

## Annotation

Книга "Введение в QNX/Neutrino 2» откроет перед вами в мельчайших подробностях все секреты ОСПВ нового поколения от компании QNX Software Systems Ltd (QSSL) — QNX/Neutrino 2. Книга написана в непринужденной манере, легким для чтения и понимания стилем, и поможет любому, от начинающих программистов до опытных системотехников, получить необходимые начальные знания для проектирования надежных систем реального времени, от встраиваемых управляющих приложений до распределенных сетевых вычислительных систем

В книге подробно описаны основные составляющие ОС QNX/Neutrino и их взаимосвязи. В частности, уделено особое внимание следующим темам:

- обмен сообщениями: принципы функционирования и основы применения;
- процессы и потоки: базовые концепции, предостережения и рекомендации;
- таймеры: организация периодических событий в программах;
- администраторы ресурсов: все, что относится к программированию драйверов устройств;
- прерывания: рекомендации по эффективной обработке.

В книге представлено множество проверенных примеров кода, подробных разъяснений и рисунков, которые помогут вам детально вникнуть в и излагаемый материал. Примеры кода и обновления к ним также можно найти на веб-сайте автора данной книги, [www.parse.com](http://www.parse.com).

- 
- [Введение в QNX/Neutrino 2](#)
    - [Предисловие](#)
    - [Введение](#)
      - 
      - [Немного истории](#)
      - [Для кого эта книга?](#)
      - [Что содержит эта книга?](#)
        - 
        - [Процессы и потоки](#)
        - [Обмен сообщениями](#)
        - [Часы, таймеры и периодические уведомления](#)

- [Прерывания](#)
- [Администраторы ресурсов](#)
- [Переход с QNX4 на QNX/Neutrino](#)
- [Скорая помощь](#)
- [Глоссарий](#)
- [Другие источники информации](#)
- [Источники информации в Интернет](#)
  - [О компании PARSE Software Devices](#)
- [Выражение признательности](#)
- [Типографские соглашения](#)
- [Глава 1](#)
  - [Основные понятия о процессах и потоках](#)
    - [Процесс как жилой дом](#)
    - [Потоки как обитатели дома](#)
    - [Назад к процессам и потокам](#)
    - [Взаимное исключение](#)
    - [Приоритеты](#)
    - [Семафоры](#)
    - [Семафор в роли мутекса](#)
  - [Роль ядра](#)
    - [Одиночный процессор](#)
    - [Несколько процессоров — симметричная мультипроцессорная система \(SMP\)](#)
    - [Ядро в роли арбитра](#)
    - [Состояния потоков](#)
  - [Процессы и потоки](#)
    - [Почему процессы?](#)
    - [Запуск процесса](#)
    - [Запуск потока](#)
  - [Дополнительно о синхронизации](#)
    - [Блокировки чтения/записи](#)
    - [Ждущие блокировки](#)
    - [Условные переменные](#)
    - [Дополнительные сервисы QNX/Neutrino](#)
    - [Пулы потоков](#)

- [Диспетчеризация и реальный мир](#)
  - 
  - [Перепланирование по аппаратному прерыванию](#)
  - [Перепланирование по системным вызовам](#)
  - [Перепланирование по исключительным ситуациям](#)
- [Резюме](#)
- [Глава 2](#)
  - [Введение в обмен сообщениями](#)
    - 
    - [Микроядро и обмен сообщениями](#)
  - [Обмен сообщениями и модель «клиент/сервер»](#)
  - [Распределенный обмен сообщениями](#)
  - [Что это означает для вас](#)
    - 
    - [Философия QNX/Neutrino](#)
  - [Обмен сообщениями и многопоточность](#)
    - 
    - [Модель «сервер/субсервер»](#)
    - [Несколько примеров](#)
  - [Применение обмена сообщениями](#)
    - 
    - [Клиент](#)
    - [Сервер](#)
    - [Иерархический принцип обмена \(send-иерархия\)](#)
    - [Идентификаторы отправителя, каналы и другие параметры](#)
    - [Составные сообщения](#)
  - [Сообщения типа «импульс» \(pulse\)](#)
    - 
    - [Прием импульса](#)
    - [Функция MsgDeliverEvent\(\)](#)
    - [Флаги канала](#)
  - [Обмен сообщениями в сети](#)
    - 
    - [Особенности обмена сообщениями в сети](#)
    - [Несколько замечаний о дескрипторах узлов](#)
  - [Наследование приоритетов](#)
    - 
    - [Так в чем тут хитрость?](#)
  - [Резюме](#)

- [Глава 3](#)
  - [Часы и таймеры](#)
    - 
    - [Периодические процессы](#)
    - [Источники прерываний таймера](#)
    - [Разрешающая способность отсчета времени](#)
    - [Флуктуации отсчета времени](#)
    - [Типы таймеров](#)
    - [Схема уведомления](#)
  - [Применение таймеров](#)
    - 
    - [Создание таймера](#)
    - [Сигнал, импульс или поток?](#)
    - [Какой таймер выбрать?](#)
    - [Сервер с периодическими импульсами](#)
    - [Таймеры, посылающие сигналы](#)
    - [Таймеры, создающие потоки](#)
    - [Опрос и установка часов реального времени, и кое-что еще](#)
  - [Тайм-ауты ядра](#)
    - 
    - [Тайм-ауты ядра и функция pthread\\_join\(\)](#)
  - [Резюме](#)
- [Глава 4](#)
  - [QNX/Neutrino и прерывания](#)
    - 
    - [Подпрограмма обработки прерывания](#)
    - [Активность прерываний по уровню и по фронту](#)
  - [Написание обработчиков прерываний](#)
    - 
    - [Подключение обработчиков прерываний](#)
    - [Отключение обработчика прерывания](#)
    - [Параметр flags](#)
    - [Обработчик прерывания](#)
    - [Функции, которые может вызывать ISR](#)
  - [Резюме](#)
- [Глава 5](#)
  - [Что такое администратор ресурсов?](#)
    - 
    - [Примеры администраторов ресурсов](#)

- [Характеристики администраторов ресурсов](#)
- [Взгляд со стороны клиента](#)
  - 
  - [Поиск сервера](#)
  - [Поиск администратора процессов](#)
  - [Обработка каталогов](#)
  - [Объединенные файловые системы](#)
  - [Резюме о клиенте](#)
- [Взгляд со стороны администратора ресурсов](#)
  - 
  - [Регистрация префикса](#)
  - [Обработка сообщений](#)
- [Библиотека администратора ресурсов](#)
  - 
  - [Реально все это за вас делает библиотека](#)
  - [За кулисами библиотеки](#)
- [Написание администратора ресурсов](#)
  - 
  - [Структуры данных](#)
  - [Структура администратора ресурсов](#)
  - [Структуры данных уровня POSIX](#)
- [Функции-обработчики](#)
  - 
  - [Общие замечания](#)
  - [Замечания о функциях установления соединения](#)
  - [Алфавитный список функций установления соединения и ввода/вывода](#)
- [Примеры](#)
  - 
  - [Простой пример функции io\\_read\(\)](#)
  - [Простой пример функции io\\_write\(\)](#)
  - [Простой пример функции io\\_devctl\(\)](#)
  - [Пример функции io\\_devctl\(\), имеющей дело с данными](#)
- [Дополнительно](#)
  - 
  - [Расширение ОСВ](#)
  - [Расширение атрибутной записи](#)
  - [Блокирование в пределах администратора ресурсов](#)
  - [Возврат элементов каталога](#)
- [Резюме](#)

- [Приложение А](#)
  - [Переход с QNX4 на QNX/Neutrino](#)
    - 
    - [Сходства](#)
    - [Улучшения](#)
  - [Философия переноса программ](#)
    - 
    - [Анализ обмена сообщениями](#)
    - [Обработчики прерываний](#)
  - [Резюме](#)
- [Приложение Б](#)
  - [Обращайтесь за помощью к профессионалам](#)
    - 
    - [Итак, у вас проблема...](#)
    - [RTFM](#)
    - [Свяжитесь с технической поддержкой](#)
  - [Другие источники информации](#)
    - 
    - [Каталог «третьих» фирм — продукты и консалтинг](#)
- [Глоссарий](#)

---

Спасибо, что скачали книгу в [бесплатной электронной библиотеке Royallib.com](#)

[Все книги автора](#)

[Эта же книга в других форматах](#)

Приятного чтения!

**Введение в QNX/Neutrino 2**  
**Руководство по программированию**  
**приложений реального времени в QNX**  
**Realtime Platform**

## Предисловие

Впервые взглянув 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 QNX/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, 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 QSSL в начале 1991 года, когда была еще только-только выпущена QNX4. Она разрабатывалась в соответствии с техническими условиями только что утвержденной спецификации POSIX 1003.1, которые должны были 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 из менее чем 15 разработчиков стала проводить локальные совещания, обсуждая всё 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 «ремонтить то, что работает».



Все фундаментальные идеи, которые Дэн Додж и Гордон Белл вложили в QNX изначально, действуют в QNX/Neutrino и по сей день — обмен сообщениями, микроядерная архитектура, предсказуемое время реакции, и т.д. Усложняла разработку QNX/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 QNX/Neutrino был разработан с учетом трех основных независимых функций: управление пространством имен путей, создание и 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 резмагтера». :-).

(«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» (Krtén) «занавеску» (curtain) — *прим. ред.* :-)

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 QNX/Neutrino, и ему был нужен минимальный пример администратора ресурсов, `/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 QNX/Neutrino, теперь каждый может писать POSIX-совместимые администраторы ресурсов с минимальными усилиями.

Однако, при том, что концепция администратора ресурсов была значительным шагом в эволюции QNX/Neutrino и обеспечивала мощный фундамент для операционной системы, новорожденная ОС требовала большего. Файловые системы, модули совместимости (например, TCP/IP) и устройства общего назначения, (последовательный интерфейс, консоли) разрабатывались параллельно. В результате огромной работы, в начале 1996 года вышла QNX/Neutrino 1.00. В течение последующих нескольких лет к работе над QNX/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, StrongARM 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 QNX/Neutrino 2.00 — как раз к моменту выхода книги Роба! :-)

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

*Питер Ван Дер Вин (Peter van der Veen),*

*С борта самолета где-то между Оттавой и Сан-Хосе,  
Сентябрь 1999 г.*

## 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 16Кб ОЗУ и не было видеокарты — 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, я 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 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 8Мб ОЗУ и винчестером на 70 Мб. На некоторых это действовало очень впечатляюще. Тем, на которых не действовало, я показывал, сколько ОЗУ и дискового пространства было еще *доступно*, при том что большую 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 улыбнулось 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 QNX 4: 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 QSSL, 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 назывался «QUNIX» («Quantum UNIX»). После вежливого письма адвокатов 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 Limited» — 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 серии. Затем, в начале 80-х, 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 QNX/Neutrino.

(Несмотря на то, что термином «Neutrino» часто называют саму ОС (сам Роб, кстати, тоже грешит этим), это не так. Neutrino — имя микроядра, а не всей ОС в целом; QNX/Neutrino была названа так, потому что является версией QNX, основанной на микроядре Neutrino. Впоследствии, после выхода пакета QNX Realtime Platform, чтобы не вносить путаницы, ОС QNX/Neutrino стали называть просто QNX6. — *прим. ред.*)

## 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 из себя и как работает QNX/Neutrino. Главы книги содержат описание состояний процессов, потоков, алгоритмов диспетчеризации, обмена сообщениями, модульной концепции построения ОС, и так далее. 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, так как именно эти концепции составляют основу QNX/Neutrino.

## Процессы и потоки

В этой главе 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 все вытекающие из этого как преимущества, так и подводные камни.

В разделе «Диспетчеризация и реальный мир» обсуждается, диспетчеризируются потоки в работающей системе, и что может вызвать перепланирование.

## Обмен сообщениями

В данной главе 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.

☞ Это одна из самых важных глав в книге!



## **Часы, таймеры и периодические уведомления**

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

## **Прерывания**

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

## **Администраторы ресурсов**

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

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

## **Переход с QNX4 на QNX/Neutrino**

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 QSSL, 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 эта глава поможет вам избежать стандартных подводных камней и написать программу так, чтобы она была переносима в обе операционные системы.

## **Скорая помощь**

Куда обращаться, если вы зашли в тупик, нашли ошибку или когда вам просто нужен добрый 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.

## Источники информации в Интернет

Веб-сайты:

<http://www.parse.com/>

Веб-сайт компании PARSE Software Devices. Информацию об опечатках в данной книге и примеры кода из нее доступны по адресу:  
[http://www.parse.com/book\\_v3/index.html](http://www.parse.com/book_v3/index.html).

<http://www.qnx.com/>

Сайт компании QSSL; здесь вы найдете всю самую свежую информацию о QNX/Neutrino. (QSSL сейчас зарегистрировала еще несколько URL в домене [qnx.com](http://www.qnx.com), см. [get.qnx.com](http://www.get.qnx.com), [qdn.qnx.com](http://www.qdn.qnx.com), [betas.qnx.com](http://www.betas.qnx.com), [partners.qnx.com](http://www.partners.qnx.com) — *прим. ред.*)

<http://search.yahoo.com/bin/search?p=QNX>

Ищите на Yahoo! Это ссылка на Интернет-каталог QNX-ресурсов.

FTP-сайты:

<ftp://ftp.parse.com>

FTP-сайт компании PARSE Software Devices. Здесь можно скачать примеры исходных текстов, приведенных в этой книге, в удобном архивном формате.

<ftp://ftp.qnx.com>

Сайт с официальными обновлениями QNX, демо-версиями программ третьих сторон и свободно-распространяемыми программами для QNX.

Телеконференции USENET:

[comp.os.qnx](http://comp.os.qnx)

Телеконференция по QNX (главным образом QNX4, но поток информации по QNX/Neutrino постоянно увеличивается).

## QUICS

QNX Users Interactive Conferencing System — интерактивная система телеконференций службы технической поддержки QSSL. Используйте клиента **telnet**, чтобы подключиться к [quics.qnx.com](http://quics.qnx.com):

**telnet quics.qnx.com**

Там вы сможете создать себе учетную запись QUICS, а затем использовать **tin** (архаичная программа чтения новостей — *прим. ред.*) для участия в телеконференциях.

Вы также можете обратиться к QUICS через Интернет, на [www.qnx.com](http://www.qnx.com).

(Справедливости ради следует отметить, что эта информация устарела — пока готовился перевод данной книги, QSSL изменила структуру технической поддержки. Старая добрая QUICS теперь — достояние истории; на ее место пришла более современная веб-ориентированная QNX Developers Network (QDN) — см. <http://qdn.qnx.com>, [nntp://inn.qnx.com](http://inn.qnx.com), [nntp://nntp.qnx.com](http://nntp.qnx.com) — прим. ред.)

## О компании PARSE Software Devices

Компания PARSE Software Devices была основана как организация, занимающаяся исследованиями и разработкой, выполняющая заказные работы и предоставляющая консультационные услуги для международного сообщества разработчиков. Наши основные направления:

- системная архитектура и проектирование;
- системы реального времени и встраиваемые системы;
- системное программирование;
- телефония/телекоммуникации/системы передачи данных;
- обучение персонала.

За информацией о заказных работах обращайтесь в компанию PARSE Software Devices по адресу [info@parse.com](mailto:info@parse.com).

Готовятся к выпуску новые книги — пожалуйста, пошлите запрос по адресу [books@parse.com](mailto:books@parse.com), чтобы подписаться на нашу информационную рассылку. Отсутствие спама гарантируется. :-)

Отметим также, что данная книга доступна для корпоративного использования компаниями класса OEM, а также он-лайн — обращайтесь по адресу [books@sparse.com](mailto:books@sparse.com) для получения дополнительной информации.

## Об авторе

Роб Кёртен выполнял (в основном контрактные) работы в области встраиваемых систем в течение более чем 13 лет, и занимался системным программированием на протяжении более 18 лет. За период работ по трехлетнему контракту с QSSL он разработал и преподавал учебные курсы «Программирование задач реального времени для ядра Neutrino» и «Написание администраторов ресурсов». Он также написал прототип администратора сети QNX/Neutrino (**npi-qnet**), а также часть учебного

пособия «Построение встраиваемых систем» («Building Embedded Systems»), поставляется в комплекте документации к QNX/Neutrino — *прим. ред.*).

Предыдущая книга Роба, «Введение в QNX 4: Руководство по программированию приложений реального времени» была удостоена Почетной премии («Award of Merit») Общества технических коммуникаций (Society for Technical Communications; <http://www.stc.org>).

Недавно он выполнял контрактную работу по заказу компании Cisco Systems Inc., в которой он разрабатывал системную архитектуру (проектирование и программирование) семейства продуктов Cisco GSR-12000 (Gigabit Switch Router — гигабитный коммутирующий маршрутизатор).

Роб имеет широкий круг интересов — от компьютерной музыки и графики до виртуальных файловых систем. Он также *заядлый* коллекционер машин серии PDP-8. Если у вас есть что-нибудь от PDP-8 — детали, документация или еще что — пожалуйста, пришлите ему весточку на [rk@parse.com](mailto:rk@parse.com)? Вы также можете посмотреть его домашнюю страничку по адресу <http://www.parse.com/~rk/>, чтобы увидеть, что он из себя представляет (на этой неделе :-).

### О Крисе Херборте

По истечении почти четырех лет работы в технической издательской группе QSSL Крис решил, что пришло время создать что-нибудь свое. Объединив свои навыки технического писателя, редактора и программиста, он создал компанию Arcane Dragon Software (что-то типа «Программного обеспечения таинственного дракона» — *прим. ред.*).

Компания Arcane Dragon Software (<http://home.beoscentral.com/chrish/ads/>) предоставляет следующие услуги:

- написание и редактирование технической литературы;
- программирование на C, C++ и Python для BeOS (см. <http://www.be.com>), QNX4, QNX/Neutrino и Linux;
- создание интерфейсов пользователя.

Крис — лауреат премии BeOS Masters Outstanding Contribution Award («За выдающиеся заслуги», одной из двух вообще когда-либо врученных) и обладатель трех высших наград Сообщества технических коммуникаций (две Почетных премии («Award of Merit») и одна Премия за мастерство («Award of Excellence»). Он также был техническим редактором книги Роба

Кёртена «*Введение в ОС QNX 4*» (издательство PARSE), книги Мартина Броуна «*BeOS: Перенос UNIX-приложений*» (издательство Morgan-Kauffman) и книги Скота Хакера «*Библия BeOS*» (издательство Peachpit).

## Выражение признательности

Появление данной книги было бы невозможным без помощи и поддержки моих коллег, которые щедро одаривали меня своими многочисленными предложениями и комментариями. Это: Люк Базинет (Luc Bazinet), Джеймс Чанг (James Chang), Дэн Додж (Dan Dodge), Дейв Донахо (Dave Donaho), Мария Годфри (Maria Godfrey), Майк Хантер (Mike Hunter), Прадип Кафейл (Pradeep Kathail), Стив Марш (Steve Marsh), Дэнни Н. Прайэри (Danny N. Priarie) и Эндрю Вернон (Andrew Vernon).

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) за уже вторую прекрасную обложку для моей книги.

Отдельное спасибо Джону Острандеру (John Olander) 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. Ниже в таблице приведены образцы принятых типографских соглашений.

Тип текста	Пример оформления
Тексты программ	<code>if (stream == NULL)</code>
Опции команд	<code>-lR</code>
Команды	<code>make</code>
Переменные окружения	<code>PATH</code>
Файлы и имена путей	<code>/dev/null</code>
Имена функций	<code>exit()</code>
Комбинации клавиш	Ctrl-Alt-Del
Клавиатурный ввод	Текст, который вы набираете
Клавиши	Enter
Вывод программ	login:
Именованные константы	<code>NULL</code>
Типы данных	<code>unsigned short</code>
Литералы	<code>0xFF</code> , "message string"
Имена переменных	<code>stdin</code>

 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 используем 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 спален, 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, суть 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, вы знаете, каково это — вы можете делать в доме все, что вы пожелаете и когда вы пожелаете, потому что в доме больше никого нет. Если вы пожелаете включить стерео, принять душ, приготовить обед, что угодно — вы просто идете и делаете это.

#### ***Многопоточность***

Ситуация в корне изменится, если вы введете 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 попытается тайком поджечь кухню, и т.д.

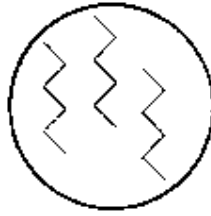
А если теперь добавить в дом несколько детей — тут все станет еще интереснее.

## **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 и 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.

На рисунке, представленном ниже, показано, как мы в дальнейшем будем представлять потоки и процессы. Процесс здесь — это круг, отображающий «контейнерную» концепцию (адресное пространство), а три ломаных линии — это потоки. Вы найдете подобные иллюстрации далее во всех разделах этой книги.



Процесс как контейнер потоков.

## 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 мутексом (сокращенно от MUTual Exclusion — взаимное исключение). Этот объект подобен замку в двери: как только поток заблокирует мутекс, никакой другой поток не сможет получить доступ к мутексу до тех пор, пока владеющий мутексом поток его не разблокирует — иными словами, мутекс будет удерживать другие потоки, подобно дверному замку.

Другая интересная параллель, которая проявляется как с мутексами, так и по аналогии с дверными замками, состоит в том, что мутекс является действительно «рекомендательной» блокировкой. Если поток не подчиняется правилам использования мутексов, 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 ожидают 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 бы был, 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 «поджигает» 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 поджигает 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 QNX/Neutrino: для рассматриваемой версии QNX/Neutrino номер 1 — самый низкий, номер 63 — 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. Если бы на момент разблокировки мутекса в ожидании находилось множество потоков, мы бы отдали этот мутекс ожидающему потоку с наивысшим приоритетом. Предположим, однако, что оба человека имеют тот же самый 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 так же, как это делает ядро в QNX/ Neutrino. В случае, когда в ожидании находится группа потоков, мы выстраиваем их сначала по приоритету, а уже в пределах каждого приоритета — по продолжительности ожидания.

Мутекс, 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, так как 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 мутекса? 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 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 мы бы 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 мутекса — это атомарная операция, и невозможна ситуация, в которой, пока вы находитесь в процессе блокировки мутекса, его заблокирует некоторый другой поток, так что в результате вы оба стали бы владельцами этого мутекса. 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 не-единичным 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. Напоминаем, 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 мутексы. Вы либо являетесь владельцем мутекса — в этом случае вы имеете доступ к ресурсу, — или нет — тогда вы не имеете доступа. Семафор, описанный выше в аналогии с доступом на кухню, является семафором со счетчиком. Такой семафор отслеживает состояние своего внутреннего счетчика обращений (т.е. число ключей, доступных потокам).

### **Семафор в роли мутекса**

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

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

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

Мутексы — это семафоры «специального назначения». Если вы пожелаете, чтобы в определенном месте программы выполнялся только один поток, эффективнее всего было бы реализовать это при помощи мутекса.

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

☞ Чтобы не возникло путаницы, необходимо также иметь в виду, что мутекс 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 в разделе «На что следует обратить

внимание при использовании SMP» 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 данный поток готов к использованию процессора. Когда мы обсуждали, например, мутексы, мы говорили о состояниях блокировки (это происходило в тех случаях, когда поток пытался завладеть мутексом, 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 мутекса. 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 мутекс).

Очевидно, это не очень-то интересная ситуация. Предположим, что готовы к выполнению сразу несколько потоков. Вспомним, не мы ли делегировали доступ к мутексу 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 весьма просто — ядро отдает процессор потоку с высшим приоритетом. Приоритеты в QNX/ Neutrino пронумерованы от единицы (самый низкий) и далее, в единичным дискретом — так же, как это было упомянуто в обсуждении получения мутекса. 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 63 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.

### *Дисциплины 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 QNX/Neutrino поддерживает две дисциплины диспетчеризации: карусельную, она же RR (Round Robin), и FIFO (First In — First Out).

### *Диспетчеризация FIFO*

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.)

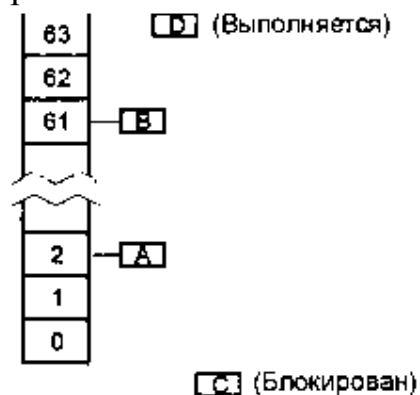
Если работающий поток завершает свою работу или добровольно уступает процессор, ядро анализирует состояние других потоков того же самого приоритета на готовность их к выполнению. Если таковых не имеется, то ядро анализирует потоки с более низким приоритетом, готовые к выполнению. Заметьте, что выражение «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 выше). Но существует также специальная функция *sched\_yield()* (базирующаяся 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 ниже, мы видим три потока, размещенных в двух различных процессах:



Три потока в двух различных процессах.

Если мы предположим, что потоки «А» и «В» находятся в состоянии READY («готов»), что поток «С» заблокирован (возможно, в ожидании мутекса), а другой поток «D» (не показан) в настоящее время выполняется, то очередь готовности, которую поддерживает ядро QNX/Neutrino, будет выглядеть следующим образом:



Два потока в очереди готовности, один заблокирован, один выполняется.

На рисунке иллюстрируется внутренняя очередь готовности, которую использует ядро при принятии решения о том, кого запланировать на выполнение следующим. Заметьте, что поток «С» не находится в очереди готовности, потому что он заблокирован, 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 predetermined кванта времени (который фиксирован и не может быть изменен). Вы можете узнать величину кванта времени, используя функцию `sched_rr_get_interval()`.

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 (то есть 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 авторитетом;
- 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).

Для систем с несколькими процессорами, приведенные выше правила остаются такими же, за исключением того, что несколько процессоров могут одновременно выполнять несколько потоков. Порядок, в котором потоки выполняются (то есть последовательность, в которой потоки ставятся на выполнение в многопроцессорной системе), определяется точно так же, как 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 так далее. 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, то есть 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 возможно, что этот порядок может измениться в будущем.

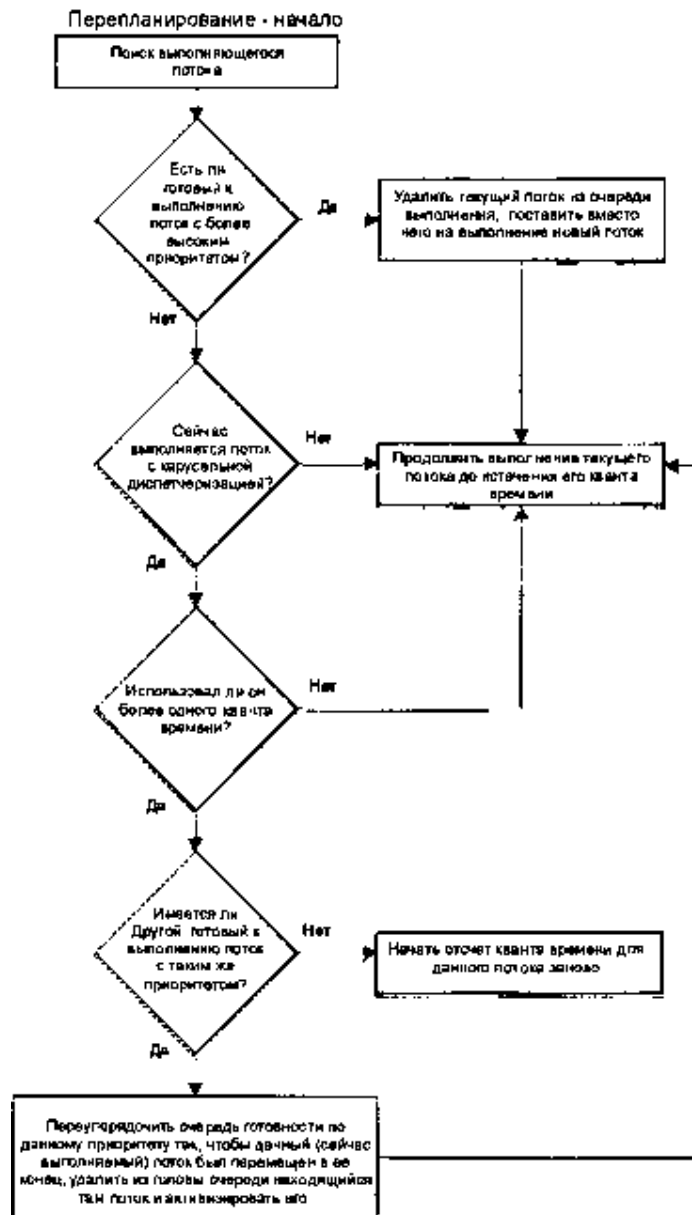


Схема алгоритма диспетчеризации.

## Состояния потоков

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

### Выполнение (RUNNING)

Состояние выполнения (RUNNING) в QNX/Neutrino означает, что поток активно использует ресурсы процессора. В системе SMP будет осуществляться выполнение множества потоков, а в системе с единственным процессором будет осуществляться 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 QNX/Neutrino имеется более дюжины 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 мутекса, этот поток находится в состоянии блокировки по мутексу (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 мутексу заблокировано несколько потоков, ядро не уделит им никакого внимания *до тех пор*, пока поток, который владеет мутексом, 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 мутекс, 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», что означает, что порядок диспетчеризации не изменяется — поток с наивысшим приоритетом продолжает работать.

### ***Полный список состояний потоков***

Ниже представлен полный список блокированных состояний с краткими пояснениями. Этот список, кстати, есть в заголовочном файле `<sys/QNX/Neutrino.h>`, только там эти состояния снабжены префиксом «STATE\_» (например, состояние READY данной таблицы там будет звучать как STATE\_READY).

#### **Если**

**состояние  
потока:**

**То это значит, что:**

DEAD	Поток «мертв», ядро ожидает освобождения занятых им ресурсов. (В классических UNIX системах это состояние также называют «zombie» — «зомби» — <i>прим. ред.</i> )
RUNNING	Поток выполняется.
READY	Поток не выполняется, но готов к работе (работает один или более потоков с более высокими или равными приоритетами).
STOPPED	Поток приостановлен (по сигналу SIGSTOP)
SEND	Поток ожидает приема своего сообщения сервером.
RECEIVE	Поток ожидает сообщение от клиента.
REPLY	Поток ожидает от сервера ответ на свое сообщение.
STACK	Поток ожидает распределения дополнительного стекового пространства.
WAITPAGE	Поток ожидает устранения администратором процессов повреждения на странице.
SIGSUSPEND	Поток ожидает сигнал.
SIGWAITINFO	Поток ожидает сигнал.
NANOSLEEP	Поток «спит» (приостановлен на определенный период времени).
MUTEX	Поток ожидает захват мутекса.
CONDVAR	Поток ожидает соблюдения условия условной переменной.
JOIN	Поток ожидает завершения другого потока.

INTR Поток ожидает прерывание.

SEM Поток ожидает захват семафора.

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

Мы рассмотрим заблокированные состояния SEND (блокировка по передаче), RECEIVE (блокировка по приему) и REPLY (блокировка по ответу) в главе «Обмен сообщениями». Состояние NANOSLEEP связано с применением функций типа *sleep()*, которые мы рассмотрим в главе «Часы, таймеры и периодические уведомления». Состояние INTR связано с использованием функции *InterruptWait()*, которую мы изучим в главе «Прерывания». 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 QNX/Neutrino допускается один или более процессов. (Аналогично — QNX/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 самые 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 QNX/Neutrino. Операционная система QNX/Neutrino состоит из множества независимых модулей, каждый из которых наделен некоторой зоной ответственности. Эти модули независимы и реализованы в отдельных процессах. Разработчики из QSSL использовали эту удобную особенность для отдельной разработки модулей, независимых друг от друга. Единственная возможная установка зависимости 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 определен, и устранять неисправности в одном таком модуле будет гораздо проще — тем более, что он не связан с другими.

Тем не менее, наиболее важным моментом является надежность. Процесс, точно так же, как и жилой дом, имеет некоторые четкие «границы». Человек, живущий в доме, точно знает, когда он в доме, а когда — нет. Поток наделен в этом смысле пониманием, что если у него есть доступ к памяти в пределах процесса, он может функционировать. Если он переступит границы адресного пространства процесса, он будет уничтожен. Это означает, что два потока, работающие в различных процессах, изолированы один от другого.



Защита памяти.

*Адресные пространства процессов* устанавливаются и поддерживаются модулем администратора процессов QNX/ Neutrino. При запуске процесса администратор процессов распределяет ему некоторый объем памяти и активирует его потоки. Отведенная данному процессу память помечается как принадлежащая ему.

Это означает, что если в данном процессе имеются несколько потоков, и ядру необходимо переключить контекст между ними, это можно сделать очень эффективно, поскольку не нужно изменять адресное пространство, достаточно просто сменить рабочий поток. Если, однако, мы должны переключиться на другой поток в другом процессе, тут уже включается в работу администратор процессов и переключает адресное пространство. Пусть вас не беспокоят возникающие при этом дополнительные издержки — под управлением QNX/Neutrino все это осуществляется очень быстро.

## Запуск процесса

Теперь обратим внимание на функции, предназначенные для работы с потоками и процессами. Любой поток может осуществить запуск процесса; единственные налагаемые здесь ограничения вытекают из основных принципов защиты (правила доступа к файлу, ограничения на привилегии и т.д.). По всей вероятности, вам уже доводилось запускать процессы — либо из системного сценария, либо из командного интерпретатора, или из программы от своего имени.

### *Запуск процесса из командной строки*

Например, при запуске процесса из командного интерпретатора вы можете ввести командную строку:

```
$ program1
```

Это предписывает командному интерпретатору запустить программу **program1** и ждать завершения ее работы. Или, вы могли набрать:

```
$ program2 &
```

Это предписывает командному интерпретатору запустить программу **program2** без ожидания ее завершения. В таком случае говорят, что программа **program2** работает в фоновом режиме.

Если вы пожелаете скорректировать приоритет программы до ее запуска, вы можете применить команду **nice** — точно так же, как в UNIX:

```
$ nice program3
```

Это предписывает командному интерпретатору запустить программу **program3** с заниженным приоритетом.

Или нет?

Если посмотреть, что происходит в действительности, то мы велели командному интерпретатору выполнить программу, называемую `nice`, с обычным приоритетом. Команда `nice` затем занизила свой собственный приоритет (отсюда и имя программы «`nice`» — «благовоспитанная») и затем запустила программу `program3` с этим заниженным приоритетом.

### *Запуск процесса из программы*

Нас обычно не заботит тот факт, что командный интерпретатор создает процессы — это просто подразумевается. В некоторых прикладных задачах можно положиться на сценарии командного интерпретатора (пакеты команд, записанные в файл), которые сделают эту работу за вас, но в ряде других случаев вы пожелаете создавать процессы самостоятельно.

Например, в большой мультипроцессорной системе вы можете пожелать, чтобы одна главная программа выполнила запуск всех других процессов вашего приложения на основании некоторого конфигурационного файла. Другим примером может служить необходимость запуска процессов по некоторому событию.

Рассмотрим некоторые из функций, которые QNX/Neutrino обеспечивает для запуска других процессов (или подмены одного процесса другим):

- `system()`;
- семейство функций `exec()`;
- семейство функций `spawn()`;
- `fork()`;
- `vfork()`.

Какую из этих функций применять, зависит от двух требований: переносимости и функциональности. Как обычно, между этими двумя требованиями возможен компромисс.

Обычно при всех запросах на создание нового процесса происходит следующее. Поток в первоначальном процессе вызывает одну из вышеприведенных функций. В конечном итоге, функция заставит администратор процессов создать адресное пространство для нового процесса. Затем ядро выполнит запуск потока в новом процессе. Этот поток выполнит несколько инструкций и вызовет функцию `main()`. (Конечно, в случае вызова функции `fork()` или `vfork()` новый поток начнет выполнение в новом процессе с возврата из `fork()` или `vfork()`, соответственно — как иметь с этим дело, мы вкратце рассмотрим ниже).

### *Запуск процесса с помощью вызова функции `system()`*

Функция `system()` — самая простая функция; она получает на вход одну командную строку, такую же, которую вы набрали бы в ответ на подсказку командного интерпретатора, и выполняет ее.

Фактически, для обработки команды, которую вы желаете выполнить, функция `system()` запускает копию командного интерпретатора.

Редактор, который я использую при написании данной книги, использует вызов `system()`. При редактировании мне может понадобиться выйти в командный интерпретатор, проверить некоторые фрагменты программ, и затем снова вернуться в редактор. Все это необходимо сделать, не потеряв позицию курсора в тексте. В этом редакторе я, к примеру, могу дать команду «`!pwd`» для отображения текущего рабочего каталога. Редактор при этом выполнит следующий код:

```
system("pwd");
```

Подходит ли функция `system()` для всех дел в Поднебесной? Конечно же, нет. Однако, ее применение может быть очень полезно для множества задач, требующих создания процессов.

### *Запуск процесса с помощью вызова функций `exec()` и `spawn()`*

Давайте рассмотрим ряд других функций создания процессов.

Следующие функции создания процессов, которые следует рассмотреть, принадлежат к семействам `exec()` и `spawn()`. Прежде, чем мы обратимся к подробностям их применения, рассмотрим суть различий между этими двумя группами функций.

Семейство функций `exec()` подменяет текущий процесс другим. Я подразумеваю под этим то, что когда процесс вызывает функцию семейства `exec()`, этот процесс прекращает выполнение текущей программы и начинает выполнять другую. Идентификатор процесса (PID) при этом не меняется, просто процесс преобразуется в другую программу. Что произойдет с потоками в данном процессе? Мы вернемся к этой теме после того, как рассмотрим функцию `fork()`.

С другой стороны, семейство функций `spawn()` так не делает. Вызов функции семейства `spawn()` создает другой процесс (с новым идентификатором), который соответствует программе, указанной в аргументах функции.

<b>Spawn</b>	<b>POSIX Exec</b>	<b>POSIX</b>
--------------	-------------------	--------------

<i>spawn()</i>	Да		
<i>spawnl()</i>	Нет	<i>execl()</i>	Да
<i>spawnle()</i>	Нет	<i>execle()</i>	Да
<i>spawnlp()</i>	Нет	<i>execlp()</i>	Да
<i>spawnlpe()</i>	Нет	<i>execlpe()</i>	Нет
<i>spawnpr()</i>	Да		
<i>spawnv()</i>	Нет	<i>execv()</i>	Да
<i>spawnve()</i>	Нет	<i>execve()</i>	Да
<i>spawnvp()</i>	Нет	<i>execvp()</i>	Да
<i>spawnvpe()</i>	Нет	<i>execvpe()</i>	Нет

Рассмотрим различные варианты функций *exec()* и *spawn()*. В таблице, представленной ниже, вы увидите, что некоторые функции из них предусмотрены POSIX, а некоторые — нет. Конечно, для максимальной переносимости, следует использовать только POSIX-совместимые функции.

При том, что названия функций могут показаться малопонятными, в их суффиксах есть логика.

Суффикс: Смысл:

*l* (нижний регистр «L») Список аргументов определяется через список параметров, заданный непосредственно в самом вызове и завершаемый нулевым аргументом NULL.

*e* Указывается окружение.

*p* Если не указано полное имя пути программы, для ее поиска используется переменная окружения PATH.

*v* Список аргументов определяется через указатель на вектор (массив) аргументов.

Список аргументов здесь — список аргументов командной строки, передаваемых программе.

Заметьте, что в библиотеке языка Си функции *spawnlp()*, *spawnvp()* и *spawnlpe()* все вызывают функцию *spawnvpe()*, которая, в свою очередь, вызывает POSIX-функцию *spawnpr()*. Функции *spawnle()*, *spawnv()* и *spawnl()* все в конечном счете вызывают функцию *spawnve()*, которая затем вызывает POSIX-функцию *spawn()*. И, наконец, POSIX-функция *spawnpr()* вызывает POSIX-функцию *spawn()*. Таким образом, в основе всех возможностей семейства *spawn()* лежит сам вызов *spawn()*.

Рассмотрим теперь различные варианты функций *spawn()* и *exec()* более подробно так, чтобы вы смогли получить навык свободного



использования различных суффиксов. Затем мы перейдем непосредственно к рассмотрению вызова функции *spawn()*.

### Суффикс «l»

Например, если я хочу вызвать команду **ls** с аргументами **-t**, **-r**, и **-l** (означает — «сортировать выходные данные по времени в обратном порядке и показывать выходные данные в длинном формате»), я мог бы определить это в программе так:

```
/* Вызвать ls и продолжить выполнение */
spawnl(P_WAIT, "/bin/ls", "/bin/ls", "-t", "-r", "-l",
      NULL);
```

```
/* Заменить себя на ls */
execl(P_WAIT, "/bin/ls", "/bin/ls", "-t", "-r", "-l",
      NULL);
```

Или, вариант с применением суффикса **v**:

```
char *argv[] = {
    "/bin/ls",
    "-t",
    "-r",
    "-l",
    NULL
};
```

```
/* Вызвать ls и продолжить выполнение */
spawnv(P_WAIT, "/bin/ls", argv);
```

```
/* Заменить себя на ls */
execv(P_WAIT, "/bin/ls", "/bin/ls", argv);
```

Почему именно такой выбор? Он дан для удобства восприятия. У вас может быть синтаксический анализатор, уже встроенный в вашу программу, и может быть удобно сразу оперировать массивами строк. В этом случае я бы рекомендовал применять варианты с суффиксом «v». Или вам может понадобиться запрограммировать вызов программы, когда вам известно, где он находится и какие имеет параметры. В этом случае, зачем вам утруждать себя созданием массива строк, когда вы знаете точно, какие нужны аргументы? Просто передайте их варианту функции с суффиксом «l».

Отметим, что мы передаем реальное имя пути программы (*/bin/ls*), а затем имя программы *еще раз* в качестве первого аргумента. Это делается для поддержки программ, которые ведут себя по-разному в зависимости от того, под каким именем они были вызваны.

Например, GNU-утилиты компрессии и декомпрессии (**gzip** и **gunzip**) фактически привязаны к одному и тому же исполняемому модулю. Когда исполняемый модуль стартует, он анализирует аргумент **argv[0]** (передаваемый функции *main()*) и принимает решение, следует ли выполнять компрессию или декомпрессию.

### Суффикс «е»

Варианты с суффиксом «е» передают программе окружение. Окружение — это только своего рода «контекст», в котором работает программа. Например, у вас может быть программа проверки орфографии, у которой есть эталонный словарь. Вместо описания каждый раз в командной строке местоположения словаря вы могли бы сделать это в окружении:

```
$ export DICTIONARY=/home/rk/.dict
$ spellcheck document.1
```

Команда **export** предписывает командному интерпретатору создать новую переменную окружения (в нашем случае **DICTIONARY**) и присвоить ей значение (*/home/rk/.dict*).

Если вы когда-либо хотели бы использовать различные словари, вы были бы должны изменить среду до выполнения программы. Это просто сделать из оболочки:

```
$ export DICTIONARY=/home/rk1.altdict
$ spellcheck document.1
```

Но как сделать это из ваших собственных программ? Для того чтобы применять «е»-версии функций *spawn()* и *exec()*, вам следует определить массив строк, представляющих собой окружение:

```
char *env[] = {
    "DICTIONARY=/home/rk/.altdict",
    NULL
};

// Запуск проверки в отдельном процессе:
spawnle(P_WAIT, "/usr/bin/spellcheck",
        "/usr/bin/spellcheck", "documents.1", NULL, env);
```

```
// Запуск проверки вместо себя:
execl("/usr/bin/spellcheck", "/usr/bin/spellcheck",
      "document.1", NULL, env);
```

### Суффикс «р»

Версии с суффиксом «р» будут искать исполняемый модуль программы в списке каталогов, приведенном в переменной окружения **PATH**. Вы, вероятно, отметили, что во всех примерах местоположение исполняемых модулей строго определено — **/bin/ls** и **/usr/bin/spellcheck**. А как быть с другими исполняемыми модулями? Если вы не хотите сразу определить точный путь к нужной программе, было бы лучше сделать так, чтобы места поиска исполняемых модулей вашей программе сообщил пользователь. Стандартная системная переменная **PATH** для этого и предназначена. Ниже приведено ее значение для минимальной системы:

```
PATH=/proc/boot:/bin
```

Это сообщает командному интерпретатору, что когда я набираю команду, он в первую очередь должен просмотреть каталог **/proc/boot/**, и если не сможет найти команду там, то должна просмотреть каталог бинарных файлов **/bin**. Переменная **PATH** представляет собой разделяемый двоеточиями список каталогов для поиска команд. К переменной **PATH** вы можете добавлять столько элементов, сколько пожелаете, но имейте в виду, что при поиске файла будут проанализированы все элементы (в приведенной последовательности).

Если вы не знаете путь к выполнимой программе, вы можете использовать варианты с суффиксом «р».

Например:

```
// Использование явного пути:
execl("/bin/ls", "/bin/ls", "-l", "-t", "-r", NULL);
```

```
// Поиск пути в PATH:
execp("ls", "ls", "-l", "-t", "-r", NULL) ;
```

Если функция *execl()* не сможет найти **ls** в **/bin**, она завершится с ошибкой. Функция *execp()* просмотрит все каталоги, указанные в **PATH**, в поисках **ls**, и завершится с ошибкой только в том случае, если не сможет найти **ls** ни в одном из этих каталогов. Это также прекрасная вещь для многоплатформенной поддержки — вашей программе не обязательно знать

имена каталогов, принятых на разных машинах, она просто выполнит поиск.

А что произойдет, если сделать так?

```
execlp("/bin/ls", "ls", "-l", "-t", "-r", NULL);
```

Выполняет ли этот вызов поиск в окружении? Нет. Вы явно указали *execlp()* имя пути, что отменяет правило поиска в **PATH**. Если **ls** не будет найдена в **/bin** (а здесь это будет именно так), то никаких других попыток поиска не выполняется — эта ситуация подобна варианту с использованием функции *execl()*.

Опасно ли смешивать явный путь с простым именем команды (например, указывать путь как **/bin/ls**, а имя — как **ls** вместо **/bin/ls**)? Обычно нет, потому что:

- значительное число программ так или иначе игнорирует **argv[0]**;
- те программы, поведение которых зависит от их имени, обычно вызывают функцию *basename()*, которая удаляет каталоговую часть **argv[0]** и возвращает только имя.

Единственная обоснованная причина использования полного имени пути в качестве первого параметра заключается в том, что программа может выводить диагностические сообщения, содержащие этот первый параметр, который немедленно укажет вам, откуда она была вызвана. Это может быть важно, если копии программы располагаются в нескольких каталогах из перечисленных в **PATH**.

Функции семейства *spawn()* имеют дополнительный параметр; во всех приведенных выше примерах я всегда указывал **P\_WAIT**. Имеются четыре флага, которые вы можете придать функции *spawn()*, чтобы изменить ее поведение:

<b>P_WAIT</b>	Вызывающий процесс (ваша программа) будет блокирован до тех пор, пока вновь созданный процесс не отработает и не завершится.
---------------	--

<b>P_NOWAIT</b>	Вызывающая программа не будет блокирована на время выполнения вновь созданной. Это позволяет вам запустить программу в фоновом режиме и продолжать выполнение, пока она делает свое дело.
-----------------	---

<b>P_NOWAITO</b>	Аналогично <b>P_NOWAIT</b> за исключением того, что устанавливается флаг <b>SPAWN_NOZOMBIE</b> . Это означает, что вы не должны беспокоить себя вызовом функции <i>waitpid()</i> для очистки кода завершения процесса.
------------------	--

<b>P_OVERLAY</b>	Этот флаг превращает вызов функции <i>spawn()</i> в соответствующий вызов <i>exec()</i> ! Ваша программа
------------------	--

преобразуется в указанную программу без изменения идентификатора процесса ID. Вообще-то, если вы хотите сделать именно так, то, наверное, будет более корректно использовать вызов *exec()*, поскольку это избавит будущих читателей ваших исходных текстов от необходимости искать *P\_OVERLAY* в справочном руководстве по библиотеке языка Си!

### *Просто spawn()*

Как мы упомянули выше, все функции семейства *spawn()*, в конечном счете, вызывают базовую функцию *spawn()*. Ниже приведен прототип функции *spawn()*:

```
#include <spawn.h>
pid_t spawn(const char *path, int fd_count,
            const int fd_map[], const struct inheritance *inherit,
            char* const argv[], char* const envp[]);
```

Мы можем не обращать внимание на параметры *path*, *argv*, и *envp* — мы уже рассмотрели их выше как местоположение исполняемого модуля (*path*), вектор параметров (*argv*) и окружение (*envp*).

Параметры *fd\_count* и *fd\_map* идут вместе. Если вы задаете нуль в *fd\_count*, тогда *fd\_map* игнорируется, и это означает, что вновь создаваемый процесс унаследует от родительского все дескрипторы файлов (кроме тех, которые модифицированы флагом *FD\_CLOEXEC* функции *fcntl()*). Если параметр *fd\_count* имеет ненулевое значение, то он задает число дескрипторов файлов, содержащихся в *fd\_map*, и будут унаследованы только они.

Параметр *inherit* — это указатель на структуру, которая содержит набор флагов, маски сигналов, и т.д. Для получения более подробной информации об этом вам следует обратиться за помощью к справочному руководству по библиотеке языка Си.

### *Запуск процесса при помощи функции fork()*

Предположим, что вы решили создать новый процесс, который был бы идентичен работающему в настоящее время процессу, и сделать это так, чтобы эти два процесса выполнялись одновременно. Вы могли бы решить

эту проблему с помощью функции *spawn()* (и параметра *P\_NOWAIT*), передав вновь создаваемому процессу достаточно информации о точном состоянии вашего процесса, чтобы новый процесс мог настроить себя сам. Однако, такой подход может оказаться чрезвычайно сложным, потому что описание «текущего состояния» процесса может потребовать большого количества данных.

Существует более простой способ — применение функции *fork()* которая просто копирует текущий процесс. У результирующего процесса как код, так и данные полностью совпадают с таковыми для родительского процесса.

Конечно же, невозможно создать процесс, который во всем был бы идентичен родительскому. Почему? Наиболее очевидное различие между этими двумя процессами должно быть в идентификаторе процесса — мы не можем создать два процесса с одним и тем же идентификатором. Если вы посмотрите документацию на функцию *fork()* в справочном руководстве по библиотеке Си, вы увидите, что между этими двумя процессами будет иметь место ряд различий. Внимательно изучите этот список, чтоб быть уверенным в корректном применении функции *fork()*.

Если после ветвления по *fork()* получаются два одинаковых процесса, то как же их различить? Когда вы вызываете *fork()*, вы тем самым создаете другой процесс, выполняющий тот же самый код и в том же самом местоположении (то есть оба процесса вернуться из вызова *fork()*), что и родительский. Рассмотрим пример программы:

```
int main (int argc, char **argv) {
    int retval;
    printf("Это определенно родительский процесс\n");
    fflush(stdout);
    retval = fork();
    printf("Кто это сказал?\n");
    return (EXIT_SUCCESS);
}
```

После вызова *fork()* оба процесса выполняют второй вызов *printf()*! Если вы запустите эту программу на выполнение, она выведет на экран примерно следующее:

```
Это определенно родительский процесс
Кто это сказал?
Кто это сказал?
```

Иными словами, оба процесса выведут вторую строку.

Существует только один способ различить эти два процесса — он заключается в использовании возвращаемого функцией *fork()* значения,

размещенного в *retval*. Во вновь созданном дочернем процессе *retval* будет иметь нулевое значение, а в родительском она будет содержать идентификатор дочернего.

Китайская грамота, да? Проясним этот момент еще одним фрагментом программы:

```
printf("PID родителя равен %d\n", getpid());
fflush(stdout);

if (child_pid = fork()) {
    printf("Это родитель, PID сына %d\n", child_pid);
} else {
    printf("Это сын, PID %d\n", getpid());
}
```

Эта программа выведет на экран примерно следующее:

```
PID родителя равен 4496
Это родитель, PID сына 8197
Это сын, PID 8197
```

Таким образом, после применения функции *fork()* вы можете определить, в каком процессе находитесь («отец» это или «сын»), анализируя значение, возвращаемое функцией *fork()*.

### ***Запуск процесса с помощью функции *vfork()****

Применение функции *vfork()* по сравнению с обычной *fork()* позволяет существенно сэкономить на ресурсах, поскольку она делает разделяемым адресное пространство родителя.

Функция *vfork()* создает «сына», но затем приостанавливает родительский поток до тех пор, пока «сын» не вызовет функцию *exec()* или не завершится (с помощью *exit()* или его друзей). В дополнение к этому, функция *vfork()* будет работать в системах с физической моделью памяти, в то время как функция *fork()* не сможет, потому что нуждается в создании такого же адресного пространства, а это в физической модели памяти просто невозможно.

### ***Создание процесса и потоки***

Предположим, что у вас есть процесс, и вы еще не создали никаких потоков (т.е., вы работаете с одним потоком — тем, который вызвал

функцию *main()*). Если вызвать функцию *fork()*, то будет создан другой процесс, и тоже с одним потоком.

Это был простейший пример.

Теперь предположим, что в вашем процессе вы вызвали *pthread\_create()* для создания другого потока. Если вы теперь вызовете функцию *fork()*, она возвратит ENOSYS (что означает, что функция не поддерживается)! Почему так?

Вы можете верить этому или нет, но это POSIX-совместимая ситуация. POSIX утверждает, что функция *fork()* может возвращать ENOSYS. На самом же деле происходит вот что: Си-библиотека QNX/Neutrino не рассчитана на ветвление процесса с потоками. Когда вы вызываете *pthread\_create()*, эта функция устанавливает флаг, сигнализирующий что-то типа «не позволяйте этому процессу применять *fork()*, потому что механизм ветвления в данном случае не определен». Затем, при вызове *fork()*, этот флаг проверяется и, если он установлен, это принуждает *fork()* вернуть значение ENOSYS.

Такая реализация была сделана преднамеренно, и причина этого кроется в потоках и мутексах. Если бы этого ограничения не было (и оно может быть снято в будущих версиях QNX/ Neutrino), то вновь созданный процесс, как и предполагается, имел бы то же самое число потоков, что и исходный. Однако, тут возникает сложность, потому что некоторые из исходных потоков могут являться владельцами мутексов. Поскольку вновь создаваемый процесс имеет ту же область данных, что и исходный, библиотека должна была бы отслеживать, какие мутексы принадлежат каким потокам в исходном процессе, и затем дублировать принадлежность мутексов в новом процессе. Это не является невозможным: есть функция, называемая *pthread\_atfork()*, которая обеспечивает процессу возможность обрабатывать такие ситуации. Однако, на момент написания этой книги функциональные возможности *pthread\_atfork()* используются не всеми мутексами в Си-библиотеке QNX/Neutrino.

### ***Так что же использовать?***

Очевидно, если вы переносите в QNX/Neutrino программу из другой ОС, вы пожелаете использовать те же механизмы, что и в исходной программе. Я бы посоветовал избегать в новом коде применения функции *fork()*, и вот почему:

- функция *fork()* не работает с несколькими потоками — см. выше;



- при работе с *fork()* в условиях многопоточности вы должны будете зарегистрировать обработчик *pthread\_atfork()* и локировать каждый мутекс по отдельности перед собственно ветвлением, а это усложнит структуру программы;

- дочерние процессы, созданные *fork()*, копируют все открытые дескрипторы файлов. Как мы увидим позже в главе «Администратор ресурсов», это требует много дополнительных усилий, которые может быть совершенно напрасными, если дочерний процесс затем сразу сделает *exec()* и тем самым закроет все открытые дескрипторы.

Выбор между семействами функций *vfork()* и *spawn()* сводится к переносимости, а также того, что должны делать родительский и дочерний процесс. Функция *vfork()* задержит выполнение до тех пор, пока дочерний процесс не вызовет *exec()* или не завершится, тогда как семейство *spawn()* может позволить работать обоим процессам одновременно. Впрочем, в разных ОС поведение функции *vfork()* может несколько отличаться.

## Запуск потока

Теперь, когда мы знаем, как запустить другой процесс, давайте рассмотрим, как осуществить запуск другого потока.

Любой поток может создать другой поток в том же самом процессе; на это не налагается никаких ограничений (за исключением объема памяти, конечно!) Наиболее общий путь реализации этого — использование вызова функций POSIX *pthread\_create()*:

```
#include <pthread.h>
int pthread_create(pthread_t *thread,
    const pthread_attr_t *attr,
    void* (*start_routine)(void*), void *arg);
```

Функция *pthread\_create()* имеет четыре аргумента :

*thread*            указатель на **pthread\_t**, где хранится идентификатор потока

*attr*             атрибутная запись

*start\_routine* подпрограмма, с которой начинается поток

*arg*              параметр, который передается подпрограмме *start\_routine*

Отметим, что указатель *thread* и атрибутная запись (*attr*) — необязательные элементы, вы можете передавать вместо них NULL.

Параметр *thread* может использоваться для хранения идентификатора вновь создаваемого потока. Обратите внимание, что в примерах, приведенных ниже, мы передадим NULL, обозначив этим, что мы не

заботимся о том, какой идентификатор будет иметь вновь создаваемый поток.

Если бы нам было до этого дело, мы бы сделали так:

```
pthread_t tid;
pthread_create(&tid, ...
printf("Новый поток имеет идентификатор %d\n", tid);
```

Такое применение совершенно типично, потому что вам часто может потребоваться знать, какой поток выполняет какой участок кода.

☞ Небольшой тонкий момент. Новый поток может начать работать еще до присвоения значения параметру *tid*. Это означает, что вы должны внимательно относиться к использованию *tid* в качестве глобальной переменной. В примере, приведенном выше, все будет корректно, потому что вызов *pthread\_create()* отработал до использования *tid*, что означает, что на момент использования *tid* имел корректное значение.

Новый поток начинает выполнение с функции *start\_routine()*, с параметром *arg*.

### *Атрибутная запись потока*

Когда вы осуществляете запуск нового потока, он может следовать ряду четко определенных установок по умолчанию, или же вы можете явно задать его характеристики.

Прежде, чем мы перейдем к обсуждению задания атрибутов потока, рассмотрим тип данных **pthread\_attr\_t**:

```
typedef struct {
    int flags;
    size_t stacksize;
    void *stackaddr;
    void (*exitfunc)(void *status);
    int policy;
    struct sched_param param;
    unsigned guardsize;
} pthread_attr_t;
```

В основном эти поля используются как:

*flags*                      Неисчисляемые (булевы) характеристики потока — например, создается поток как «обособленный» или

«синхронизирующий».

<i>stacksize</i> , <i>stackaddr</i> и <i>guardsize</i>	Параметры стека.
<i>exitfunc</i>	Функция, выполняемая перед завершением потока.
<i>policy</i> и <i>param</i>	Параметры диспетчеризации.

Доступны следующие функции:

Управление атрибутами

*pthread\_attr\_destroy()*

*pthread\_attr\_init()*

Флаги (булевы характеристики)

*pthread\_attr\_getdetachstate()*

*pthread\_attr\_setdetachstate()*

*pthread\_attr\_getinheritsched()*

*pthread\_attr\_setinheritsched()*

*pthread\_attr\_getscope()*

*pthread\_attr\_setscope()*

Параметры стека

*pthread\_attr\_getguardsize()*

*pthread\_attr\_setguardsize()*

*pthread\_attr\_getstackaddr()*

*pthread\_attr\_setstackaddr()*

*pthread\_attr\_getstacksize()*

*pthread\_attr\_setstacksize()*

Параметры диспетчеризации

*pthread\_attr\_getschedparam()*

*pthread\_attr\_setschedparam()*

*pthread\_attr\_getschedpolicy()*

*pthread\_attr\_setschedpolicy()*

Список выглядит довольно большим (18 функций), но в действительности нас будет заботить применение только примерно половины функций из этого списка, потому что все эти они сгруппированы по парам «get» — «set», т.е. в каждой паре есть функция как получения параметров (get), так и их установки (set) — за исключением функций *pthread\_attr\_init()* и *pthread\_attr\_destroy()*.

Прежде чем мы исследуем назначения атрибутов, следует отметить одно обстоятельство. Вы обязаны вызвать *pthread\_attr\_init()* для инициализации атрибутивной записи до момента ее использования,

задействовать ее с помощью соответствующей функции (функций) *pthread\_attr\_set\*()* и только затем вызвать функцию *pthread\_create()* для создания потока. Изменение атрибутной записи после того, как поток уже создан, не будет иметь никакого действия.

### ***Администрирование атрибутов потока***

Перед использованием атрибутной записи для ее инициализации следует вызвать функцию *pthread\_attr\_init()*:

```
...  
pthread_attr_t attr;  
...  
pthread_attr_init(&attr);
```

Вы можете также вызывать *pthread\_attr\_destroy()* для «деинициализации» атрибутной записи потока, но так обычно никто не делает (если не требуется жесткой POSIX-совместимости).

В приведенных ниже описаниях значения по умолчанию помечены комментарием «(по умолчанию)».

### ***Атрибут потока «flags» (флаги)***

Три функции — *pthread\_attr\_setdetachstate()*, *pthread\_attr\_setinheritsched()* и *pthread\_attr\_setscope()* — определяют, создается ли поток как «синхронизирующий» («joinable») или как «обособленный» (detached), наследует ли поток атрибуты диспетчеризации от создающего потока или использует атрибуты диспетчеризации, указанные в функциях *pthread\_attr\_setschedparam()* и *pthread\_attr\_setschedpolicy()*, и, наконец, имеет ли поток масштаб «системы» или «процесса».

Для создания «синхронизирующего» потока (это значит, что с завершением этого потока можно синхронизировать другой поток при помощи функции *pthread\_join()*), используется вызов:

*(по умолчанию)*

```
pthread_attr_setdetachstate(&attr,  
PTHREAD_CREATE_JOINABLE);
```

Чтобы создать поток, синхронизация с завершением которого невозможна (такой поток называют «обособленным»), надо было бы сделать так:

```
pthread_attr_setdetachstate(&attr,  
PTHREAD_CREATE_DETACHED);
```

Если вы желаете, чтобы поток унаследовал атрибуты диспетчеризации от потока, его создающего (то есть имел бы ту же самую дисциплину диспетчеризации и тот же самый приоритет, что и родитель), вам следует сделать так:

*(по умолчанию)*

```
pthread_attr_setinheritsched(&attr, PTHREAD_INHERIT_SCHED);
```

Для создания потока, который использует атрибуты диспетчеризации, указанные в непосредственно в атрибутной записи (это делается при помощи функций *pthread\_attr\_setschedparam()* и *pthread\_attr\_setschedpolicy()*), вызов выглядел бы следующим образом:

```
pthread_attr_setinheritsched(&attr,  
PTHREAD_EXPLICIT_SCHED);
```

И наконец, функция *pthread\_attr\_setscope()*. Вам не придется ее вызывать никогда. Почему? Потому что QNX/Neutrino поддерживает для потоков только масштаб системы, и соответствующее значение устанавливается по умолчанию, когда вы инициализируете атрибут. (Масштаб системы означает, что за обладание ресурсами все потоки в системе конкурируют друг с другом; масштаб процесса же означает, что потоки конкурируют за процессор только в пределах «своего» процесса, а диспетчеризацию процессов выполняет ядро).

Если вам необходимо вызвать эту функцию, вы можете сделать это только следующим образом:

*(по умолчанию)*

```
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
```

### ***Атрибуты потока «stack» (параметры стека)***

Прототипы функций установки параметров стека в атрибутах потока выглядят следующим образом:

```
int pthread_attr_setguardsize(pthread_attr_t *attr,  
size_t gsize);
```

```
int pthread_attr_setstackaddr(pthread_attr_t *attr,  
void *addr);
```

```
int pthread_attr_setstacksize(pthread_attr_t *attr,  
size_t ssize);
```

Все эти три функции имеют в качестве первого параметра атрибутивную запись, вторые параметры перечислены ниже:

*gsize* Размер «области защиты».

*addr* Адрес стека, если последний вами предусмотрен.

*ssize* Размер стека.

Область защиты — это область памяти, расположенная сразу после стека, которую поток не может использовать для записи. Если это происходит (а это означает, что стек вот-вот переполнится), потоку будет послан SIGSEGV. Если размер области защиты равен 0, это означает, что области защиты не предусматривается. Это также подразумевает, что проверка стека на переполнение выполняться не будет. Если размер области защиты отличен от нуля, то это устанавливает его по меньшей мере в общесистемное значение по умолчанию (которое вы можете получить по запросу *sysconf()*, указав ему константу *\_SC\_PAGESIZE*). Заметьте, что ненулевой минимально возможный размер области защиты составляет одну страницу (например, 4 Кб для процессора x86). Также отметьте, что страница защиты не занимает никакой физической памяти, это уловка с применением виртуальной адресации (MMU). Параметр *addr* представляет собой адрес стека, если вы его задаете явно. Вы можете задать вместо него NULL, что будет значить, что система будет должна сама распределить (и освободить!) стек для потока. Преимущество явного определения стека для потока состоит в том, что вы сможете делать «посмертный» (после аварийного завершения) анализ глубины стека. Это достигается распределением области стека и заполнением ее некоторой «подписью» (например, многократно повторяемой строкой «STACK»), после чего поток запускается на выполнение. По завершении работы потока вы сможете проанализировать область стека и посмотреть, на какую глубину поток затер в ней вашу «подпись», и тем самым определить максимальную глубину стека, использованную потоком в данном конкретном сеансе выполнения.

Параметр *ssize* определяет размер стека. Если вы явно задаете адрес области стека в параметре *addr*, то параметр *ssize* должен задавать размер этой области. Если вы не задаете адрес области стека в параметре *addr* (то есть передаете вместо адреса NULL), то параметр *ssize* сообщает системе, стек какого размера следует распределить. Если вы укажете для параметра *ssize* значение 0 (ноль), система выберет размер стека, заданный по умолчанию. Очевидно, что задавать 0 в качестве параметра *ssize* и при этом явно указывать адрес стека, используя параметр *addr* — порочная практика, поскольку в действительности вы тем самым заявляете: «вот указатель на

объект, который имеет некоторый заданный по умолчанию размер». Проблема здесь заключается в том, что между размером объекта и передаваемым значением нет никакой связи.

☞ Если стек назначается с помощью параметра *attr*, данный поток не будет защищен от переполнения этого стека (то есть область защиты будет отсутствовать).

### *Атрибуты потока «scheduling» (диспетчеризация)*

Наконец, если вы определяете `PTHREAD_EXPLICIT_SCHED` для функции `pthread_attr_setinheritsched()`, тогда вам необходимо будет как-то определить дисциплину диспетчеризации и приоритет для потока, который вы намерены создать.

Это выполняется с помощью двух функций:

```
int pthread_attr_setschedparam(pthread_attr_t *attr,  
    const struct sched_param *param);
```

```
int pthread_attr_setschedpolicy(pthread_attr_t *attr,  
    int policy);
```

С параметром *policy* все просто — это либо `SCHED_FIFO`, либо `SCHED_RR`, либо `SCHED_OTHER`.

☞ В рассматриваемой версии QNX/Neutrino параметр `SCHED_OTHER` интерпретируется как `SCHED_RR` (карусельная диспетчеризация).

Параметр *param* — структура, которая содержит единственный элемент: *sched\_priority*. Задайте этот параметр путем прямого присвоения ему значения желаемого приоритета.

☞ Стандартная ошибка, которой следует избегать, заключается в задании `PTHREAD_EXPLICIT_SCHED` и затем определением только дисциплины диспетчеризации. Проблема состоит в том, что в инициализированной атрибутной записи значение *param.sched\_priority* есть 0 (ноль). Это тот же самый приоритет, что и у «холостого» потока (`IDLE`), что означает, что создаваемый вами поток будет конкурировать за процессор с «ХОЛОСТЫМ» ПОТОКОМ.

Плавали, знаем. :-)

На том, что QSSL зарезервировала нулевой приоритет только для «холостого» потока, уже «прокололось» немало программистов. Поток с нулевым приоритетом просто не сможет выполняться.

### *Несколько примеров*

Давайте рассмотрим ряд примеров. Будем считать, что в обсуждаемой программе подключены нужные заголовочные файлы (`<pthread.h>` и `<sched.h>`), а также что поток, который предстоит создать, называется `new_thread()`, и для него существуют все необходимые прототипы и определения.

Самый обычный способ создания потока — просто оставить везде значения по умолчанию:

```
pthread_create(NULL, NULL, new_thread, NULL);
```

В вышеупомянутом примере мы создали наш новый поток со значениями параметров по умолчанию и передали ему `NULL` в качестве его единственного параметра (третий `NULL` в указанном выше вызове `pthread_create()`).

Вообще говоря, вы можете передавать вашему новому потоку что угодно через параметр `arg`. Например, число 123:

```
pthread_create(NULL, NULL, new_thread, (void*)123);
```

Более сложный пример — создание «обособленного» (`detached`) потока с диспетчеризацией карусельного типа (`RR`) и приоритетом 15:

```
pthread_attr_t attr;
```

```
// Инициализировать атрибутивную запись
```

```
pthread_attr_init(&attr);
```

```
// Установить «обособленность»
```

```
pthread_attr_setdetachstate(&attr,
```

```
PTHREAD_CREATE_DETACHED);
```

```
// Отменить наследование по умолчанию (INHERIT_SCHED)
```

```
pthread_attr_setinheritsched(&attr,
```

```
PTHREAD_EXPLICIT_SCHED);
```

```
pthread_attr_setschedpolicy(&attr, SCHED_RR);
```

```
attr.param.sched_priority = 15;
```



```
// И, наконец, создать поток
pthread_create(NULL, &attr, new_thread, NULL);
```

Для того чтобы увидеть, как «выглядит» многопоточная программа, можно запустить из командного интерпретатора команду `pidin`. Скажем, нашу программу зовут `spud`. Если мы выполняем `pidin` один раз до создания программой `spud` потоков, и еще раз — после того, как `spud` создала два потока (тогда всего их будет три), то вот как примерно будет выглядеть вывод (я укоротил вывод `pidin` для демонстрации только того, что относится к `spud`):

```
# pidin
pid  tid name prio STATE Blocked
12301  1 spud  10r READY
```

```
# pidin
pid  tid name prio STATE Blocked
12301  1 spud  10r READY
12301  2 spud  10r READY
12301  3 spud  10r READY
```

Вы можете видеть, что процесс `spud` (идентификатор процесса 12301) имеет три потока (столбец «tid» в таблице). Эти три поток» выполняются с приоритетом 10, с диспетчеризацией карусельного (RR) типа (обозначенной как «r» после цифры 10). Все три процесса находятся в состоянии готовности (READY), т. е. готовы использовать процессор, но в настоящее время не выполняются (поскольку в данный момент выполняется другой поток с более высоким приоритетом).

Теперь, когда мы знаем все о создании потоков, давайте рассмотрим, как и где мы можем этим воспользоваться.

### *Где хороша многопоточность*

Существует два класса задач, где можно было бы эффективно применять многопоточность.

☞ Потоки подобны перегруженным операторам в языке Си++. Поначалу может показаться хорошей идеей перегрузить каждый оператор какой-либо дополнительной интересной функцией, но это сделает программу трудной для восприятия. Аналогичная ситуация с потоками. Вы могли бы создать множество потоков, но это усложнит ваш код и сделает программу малопонятной, а значит, сложной в сопровождении. Разумное же применение потоков, наоборот, внесет в программу дополнительную функциональную ясность.

Применение потоков хорошо там, где можно выполнять операции параллельно — например, в ряде математических задач (графика, цифровая обработка сигналов, и т.д.). Потоки также прекрасны там, где программа должна выполнять несколько независимых функций, при этом использующих общие данные — например, веб-сервер, который обслуживает несколько клиентов одновременно. Эти два класса задач мы здесь и рассмотрим.

### ***Потоки в математических операциях***

Предположим, что мы имеем графическую программу, выполняющую алгоритм трассировки луча. Каждая строка раstra на экране зависит от содержимого основной базы данных (которая описывает генерируемую картинку). Ключевым моментом здесь является то, что *каждая строка раstra не зависит от остальных*. Это обстоятельство (независимость строк раstra) автоматически приводит к программированию данной задачи как многопоточной.

Ниже приведен однопоточный вариант:

```
int main (int argc, char **argv) {
    int x1;

    ... // Выполнить инициализации

    for (x1 = 0; x1 < num_x_lines; x1++) {
        do_one_line(x1);
    }

    ... // Вывести результат
```

```
}
```

Здесь мы видим, что программа итеративно по всем значениям рассчитывает необходимые растровые строки.

В многопроцессорных системах эта программа будет использовать только один из процессоров. Почему? Потому что мы не указали операционной системе выполнять что-либо параллельно. Операционная система не настолько умна, чтобы посмотреть на программу и сказать: «Эй, секундочку! У нас ее 4 процессора, и похоже, что у нас тут несколько независимых потоков управления. Запущу-ка я это на всех 4 процессорах сразу!»

Так что это дело разработчика (ваше дело!) — сообщить QNX/Neutrino, какие разделы программы следует выполнять параллельно. Проще всего это можно было бы сделать так:

```
int main (int argc, char **argv) {
    int x1;

    ... // Выполнить инициализации

    for (x1 = 0; x1 < num_x_lines; x1++) {
        pthread_create(NULL, NULL, do_one_line, (void*)x1);
    }

    ... // Вывести результат
}
```

С таким упрощением связано множество проблем. Первая из них (и самая незначительная) состоит в том, что функцию *do\_one\_line()* придется модифицировать так, чтобы она могла в качестве своего аргумента принимать значение типа **void\*** вместо **int**. Это можно легко исправить с помощью оператора приведения типа (typecast).

Вторая проблема несколько сложнее. Скажем, что разрешающая способность дисплея, для которой вы рассчитывали картинку, была равна 1280×1024. Нам пришлось бы создать 1280 потоков! В общем-то, для QNX/Neutrino это не проблема — QNX/Neutrino позволяет создавать до 32767 потоков в одном процессе! Однако, каждый поток должен иметь свой уникальный стек. Если ваш стек имеет разумный размер (скажем 8 Кб), эта программа израсходует под стек 1280×8 Кб (10 мегабайт!) ОЗУ. И ради чего? В вашей системе есть только 4 процессора. Это означает, что только 4 из этих 1280 потоков будут работать одновременно, а другие 1276 потоков будут ожидать доступа к процессору. (В действительности, в данном случае

пространство под стек будет выделяться только по мере необходимости. Но тем не менее, это все равно расходование ресурсов впустую — есть ведь еще и другие издержки.)

Более красивым способом решения этой задачи было бы разбить ее на 4 части (по одной подзадаче на каждый процессор), и обрабатывать каждую часть как отдельный поток:

```
int num_lines_per_cpu;
int num_cpus;

int main (int argc, char **argv) {
    int cpu;

    ... // Выполнить инициализации

    // Получить число процессоров
    num_cpus = _syspage_ptr->num_cpu;
    num_lines_per_cpu = num_x_lines / num_cpus;
    for (cpu = 0; cpu < num_cpus; cpu++) {
        pthread_create(NULL, NULL, do_one_batch, (void*)cpu);
    }

    ... // Вывести результат

}

void* do_one_batch(void *c) {
    int cpu = (int)c;
    int x1;
    for (x1 = 0; x1 < num_lines_per_cpu; x1++) {
        do_line_line(x1 + cpu * num_lines_per_cpu);
    }
}
```

Здесь мы запускаем только *num\_cpus* потоков. Каждый поток будет выполняться на отдельном процессоре. А поскольку мы имеем дело с небольшим числом потоков, мы тем самым не засоряем память ненужными стеками. Обратите внимание, что мы получили число процессоров путем разыменования глобальной переменной — указателя на системную страницу *\_syspage\_ptr*. (Дополнительную информацию относительно системной страницы можно найти в книге «Building Embedded Systems»

(поставляется в комплекте документации по QNX/ Neutrino — прим. ред.) или в заголовочном файле `<sys/syspage.h>`).

### ***Программирование для одного или нескольких процессоров***

Последняя программа в первую очередь интересна тем, что будет корректно функционировать в системе с одиночным процессором тоже. Просто будет создан только один поток, который и выполнит всю работу. Дополнительные издержки (один стек) с лихвой окупаются гибкостью программы, умеющей работать быстрее в многопроцессорной системе.

### ***Синхронизация по отношению к моменту завершения потока***

Я уже упоминал, что с приведенным выше упрощенным примером программы связана масса проблем. Так вот, еще одна связанная с ним проблема состоит в том, что функция *main()* сначала запускает целый букет потоков, а затем отображает результаты. Но как функция узнает, когда уже можно выводить результаты?

Заставлять *main()* заниматься опросом, закончены ли вычисления, противоречит самому замыслу ОС реального времени.

```
int main (int argc, char **argv) {  
    ...  
    // Запустить потоки, как раньше  
    while (num_lines_completed < num_x_lines) {  
        sleep(1);  
    }  
}
```

Не вздумайте писать такие программы!

Для решения этой задачи существуют два изящных решения: применение функций *pthread\_join()* и *barrier\_wait()*.

### ***«Присоединение» (joining)***

Самый простой метод синхронизации — это «присоединение» потоков. Реально это действие означает ожидание завершения.

Присоединение выполняется одним потоком, ждущим завершения другого потока. Ждущий поток вызывает *pthread\_join()*:

```
#include <pthread.h>
```

```
int pthread_join(pthread_t thread, void **value_ptr);
```

Функции *pthread\_join()* передается идентификатор потока, к которому вы желаете присоединиться, а также необязательный аргумент *value\_ptr*, который может быть использован для сохранения возвращаемого присоединяемым потоком значения (Вы можете передать вместо этого параметра NULL, если это значение для вас не представляет интереса — в данном примере мы так и сделаем).

Где нам брать идентификатор потока? Мы игнорировали его в функции *pthread\_create()*, передав NULL в качестве первого параметра. Давайте исправим нашу программу:

```
int num_lines_per_cpu;
```

```
int num_cpus;
```

```
int main(int argc, char **argv) {
```

```
    int cpu;
```

```
    pthread_t *thread_ids;
```

```
    ... // Выполнить инициализации
```

```
    thread_ids = malloc(sizeof(pthread_t) * num_cpus);
```

```
    num_lines_per_cpu = num_x_lines / num_cpus;
```

```
    for (cpu = 0; cpu < num_cpus; cpu++) {
```

```
        pthread_create(
```

```
            &thread_ids[cpu], NULL, do_one_batch, (void*)cpu);
```

```
    }
```

```
    // Синхронизироваться с завершением всех потоков
```

```
    for (cpu = 0; cpu < num_cpus; cpu++) {
```

```
        pthread_join(thread_ids[cpu], NULL);
```

```
    }
```

```
    ... // Вывести результат
```

```
}
```

Обратите внимание, что на этот раз мы передали функции *pthread\_create()* в качестве первого аргумента указатель на *pthread\_t*. Там и будет сохранен идентификатор вновь созданного потока. После того как первый цикл *for* завершится, у нас будет *num\_cpu* работающих потоков, плюс поток, выполняющий *main()*. Потребление ресурсов процессора

потоком *main()* нас мало интересует — этот поток потратит все свое время на ожидание.

Ожидание достигается применением функции *pthread\_join()* к каждому из наших потоков. Сначала мы ждем завершения потока `thread_ids[0]`. Когда он завершится, функция *pthread\_join()* разблокируется. Следующая итерация цикла `for` заставит нас ждать завершения потока `thread_ids[1]`, и так далее для всех *num\_cpus* потоков.

В этот момент возникает законный вопрос: «А что если потоки завершат работу в обратном порядке?» Другими словами, если имеются 4 процессора, и по какой-либо причине поток, выполняющийся на последнем процессоре (с номером 3), завершит работу первым, затем завершится поток, выполняющийся на процессоре с номером 2, и так далее? Вся прелесть приведенной схемы заключается в том, что ничего плохого не произойдет.

Первое, что произойдет — это то, что *pthread\_join()* блокируется на `thread_ids[0]`. Тем временем пусть завершится поток `thread_ids[3]`. Это не окажет абсолютно никакого воздействия на поток *main()*, который будет по-прежнему ждать завершения первого потока. Затем, пусть завершит работу поток `thread_ids[2]`. По-прежнему, никаких последствий. И так далее — пока не завершит работу поток `thread_ids[0]`.

В этот момент *pthread\_join()* разблокируется, и мы немедленно переходим к следующей итерации цикла `for`. Вторая итерация цикла `for` применит *pthread\_join()* к потоку `thread_ids[1]`, который не будет заблокирован, и итерация завершится немедленно. Почему? Потому что поток, идентифицированный как `thread_ids[1]`, уже завершился. Поэтому наш цикл `for` просто «проскочит» остальные потоки и завершится. В этот момент мы будем знать, что вычислительные потоки синхронизированы, и теперь мы можем выводить результаты отображение.

### *Применение барьера*

Когда мы говорили о синхронизации функции *main()* по моменту завершения рабочих потоков (в параграфе «Синхронизация по отношению к моменту завершения потока», см. выше), мы упомянули два метода синхронизации: один метод с применением функции *pthread\_join()*, который мы только что рассмотрели, и метод с применением барьера.

Возвращаясь к нашей аналогии с процессами в жилом доме, предположим, что семья пожелала где-нибудь отдохнуть на природе. Водитель садится в микроавтобус и запускает двигатель. И ждет. Водитель

будет ждать до тех пор, пока *все* члены семьи не сядут в машину, и только затем можно будет ехать — не можем же мы кого-нибудь оставить!

Точно так происходит и в нашем примере с выводом графики на дисплей. Основной поток должен дожидаться того момента, когда все рабочие потоки завершат работу, и только затем можно начинать следующую часть программы.

Однако, отметьте для себя одну важную отличительную особенность. С применением функции *pthread\_join()* мы ожидаем завершения потоков. Это означает, что на момент ее разблокирования потоков нет больше с нами; они закончили работу и завершились.

В случае с барьером, мы ждем «встречи» определенного числа потоков у барьера. Затем, когда заданное число потоков достигнуто, мы их всех разблокируем (заметьте, что потоки при этом продолжают выполнять свою работу).

Сначала барьер следует создать при помощи функции *barrier\_init()*:

```
#include <sync.h>
```

```
int barrier_init(barrier_t *barrier, const barrier_attr_t
*attr, int count);
```

Эта функция создает объект типа «барьер» по переданному ей адресу (указатель на барьер хранится в параметре *barrier*) и назначает ему атрибуты, которые определены в *attr* (мы будем использовать NULL, чтобы установить значения по умолчанию). Число потоков, которые должны вызывать функцию *barrier\_wait()*, передается в параметре *count*.

После того как барьер создан, каждый из потоков должен будет вызвать функцию *barrier\_wait()*, чтобы сообщить, что он отработал:

```
#include <sync.h>
```

```
int barrier_wait(barrier_t *barrier);
```

После того как поток вызвал *barrier\_wait()*, он будет блокирован до тех пор, пока число потоков, указанное первоначально в параметре *count* функции *barrier\_init()*, не вызовет функцию *barrier\_wait()* (они также будут блокированы). После того как нужное число потоков выполнит вызов функции *barrier\_wait()*, все эти потоки будут разблокированы «одновременно».

Вот пример:

```
/*
 * barrier1.c
 */
```



```
#include <stdio.h>
#include <time.h>
#include <sync.h>
#include <sys/neutrino.h>

barrier_t barrier; // Объект типа «барьер»

void* thread1(void *not_used) {
    time_t now;
    char buf[27];
    time(&now);
    printf("Поток 1, время старта %s", ctime_r(&now, buf));

    // Выполнить вычисления
    // (вместо этого просто сделаем sleep)
    sleep(20);

    barrier_wait(&barrier);

    // После этого момента все потоки уже завершатся
    time(&now);
    printf("Барьер в потоке 1, время срабатывания %s",
        ctime_r(&now, buf));
}

void* thread2(void *not_used) {
    time_t now;
    char buf[27];
    time(&now);
    printf("Поток 2, время старта %s", ctime_r(&now, buf));

    // Выполнить вычисления
    // (вместо этого просто сделаем sleep)
    sleep(40);

    barrier_wait(&barrier);

    // После этого момента все потоки уже завершатся
    time(&now);
    printf("Барьер в потоке 2, время срабатывания %s",
```

```

    ctime_r(&now, buf));
}

main() // Игнорировать аргументы
{
    time_t now;
    char buf[27];

    // Создать барьер со значением счетчика 3
    barrier_init(&barrier, NULL, 3);
    // Создать два потока, thread1 и thread2
    pthread_create(NULL, NULL, thread1, NULL);
    pthread_create(NULL, NULL, thread2, NULL);

    // Сейчас выполняются оба потока
    // Ждать завершения
    time(&now);
    printf("main(): ожидание у барьера, время %s",
        ctime_r(&now, buf));

    barrier_wait(&barrier);

    // После этого момента все потоки уже завершатся
    time(&now);
    printf("Барьер в main(), время срабатывания %s",
        ctime_r(&now, buf));
}

```

Основной поток создал объект типа «барьер» и инициализировал его значением счетчика, равным числу потоков (включая себя!), которые должны «встретиться» у барьера, прежде чем он «прорвется». В нашем примере этот индекс был равен 3 — один для потока *main()*, один для потока *thread1()* и один для потока *thread2()*. Затем, как и прежде, стартуют потоки вычисления графики (в нашем случае это потоки *thread1()* и *thread2()*). Для примера вместо приведения реальных алгоритмов графических вычислений мы просто временно «усыпили» потоки, указав в них **sleep(20)** и **sleep(40)**, чтобы имитировать вычисления. Для осуществления синхронизации основной поток (*main()*) просто блокирует сам себя на барьере, зная, что барьер будет разблокирован только после того, как рабочие потоки аналогично присоединятся к нему.

Как упоминалось ранее, с функцией *pthread\_join()* рабочие потоки для синхронизации главного потока с ними должны умереть. В случае же с барьером потоки живут и чувствуют себя вполне хорошо. Фактически, отработав, они просто разблокируются по функции *barrier\_wait()*. Тонкость здесь в том, что вы обязаны предусмотреть, что эти потоки должны делать дальше! В нашем примере с графикой мы не дали им никакого задания для них — просто потому что мы так придумали алгоритм. В реальной жизни вы могли бы захотеть, например, продолжить вычисления.

### ***Несколько потоков при одиночном процессоре***

Предположим, что мы слегка изменили наш пример так, чтобы можно было проиллюстрировать, почему иногда хорошо иметь несколько потоков даже в системе с одиночным процессором.

В таком модифицированном примере один узел на сети ответственен за вычисление строк раstra (как и в примере с графикой, рассмотренном выше). Однако, когда строка рассчитана, ее данные должны быть отправлены по сети другому узлу, который выполняет функцию отображения. Ниже приведена соответствующая модифицированная функция *main()* (на основе первоначального примера без потоков):

```
int main(int argc, char **argv) {
    int x1;

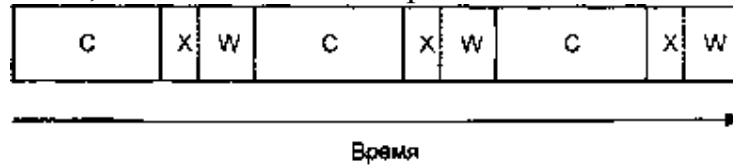
    ... // выполнить инициализации

    for (x1 = 0; x1 < num_x_lines; x1++) {
        do_one_line(x1); // Область «C» на схеме
        tx_one_line_wait_ack(x1); // Области «X» и «W» на схеме
    }
}
```

Обратите внимание на то, что мы исключили отображающую часть программы и вместо этого добавили функцию *tx\_one\_line\_wait\_ack()*. Далее предположим, что мы имеем дело с достаточно медленной сетью, но процессор в действительности не занимается передачей данных — он просто отдает их некоторым аппаратным средствам, которые уже сами позаботятся об их передаче. Функция *tx\_one\_line\_wait\_ack()* потребует немного процессорного времени на то, чтобы обеспечить передачу данных аппаратным средствам, и после этого, пока не получит подтверждения о

получении данных от удаленного узла, не будет потреблять процессорное время вообще.

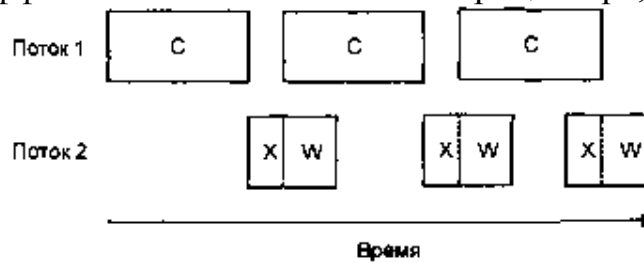
Ниже представлена диаграмма, иллюстрирующая загрузку процессора в данном случае (графические вычисления на ней обозначены как «С», передача — как «Х», а ожидание подтверждения — как «W»).



Последовательное выполнение, один процессор.

Минуточку! Мы тратим впустую драгоценные секунды, ожидая, пока аппаратура сделает свое дело!

Если мы сделали бы это в многопоточном варианте, мы смогли бы добиться более эффективного использования процессора, так?



Многопоточное выполнение, один процессор

Это уже намного лучше, потому что теперь, даже при том, второй поток затрачивает немного времени на ожидание, мы добились уменьшения суммарного времени вычислений.

Если бы в нашем примере тратилось  $T_{\text{compute}}$  единиц времени на вычисления,  $T_{\text{tx}}$  — на передачу и  $T_{\text{wait}}$  — на ожидание аппарату средств, тогда для первого случая в нашем примере общие затраты времени на обработку были бы равны:

$$(T_{\text{compute}} + T_{\text{tx}} + T_{\text{wait}}) \cdot \text{num\_x\_lines},$$

тогда как затраты времени при использовании двух потоков были бы равны:

$$(T_{\text{compute}} + T_{\text{tx}}) \cdot \text{num\_x\_lines} + T_{\text{wait}},$$

что меньше на величину:

$$T_{\text{wait}} \cdot (\text{num\_x\_lines} - 1),$$

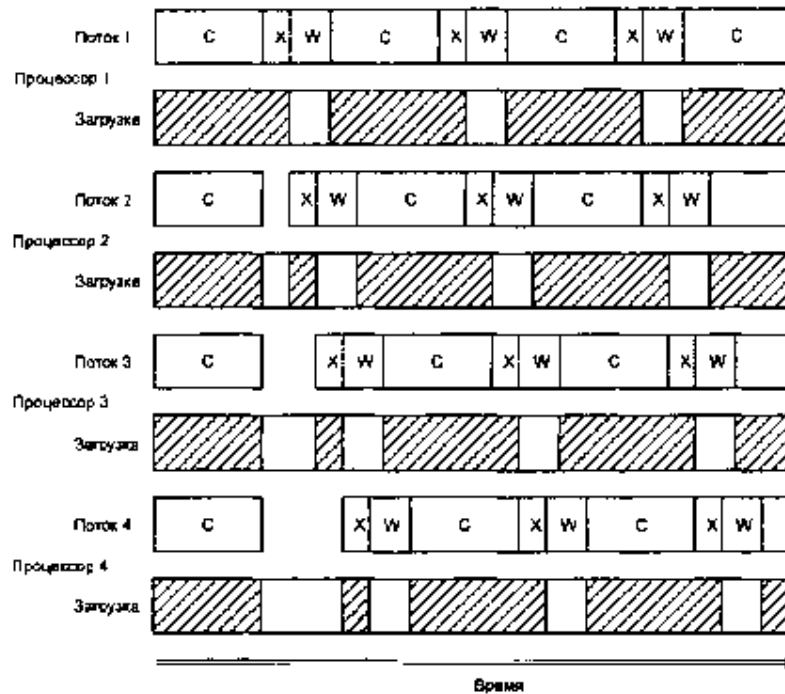
в предположении, конечно, что  $T_{\text{wait}} \leq T_{\text{compute}}$ .

☞ Отметим, что мы изначально будем ограничены интервалом времени, равным:

$$T_{\text{compute}} + T_{\text{tx}} \cdot \text{num\_x\_lines},$$

потому что мы должны будем завершить по меньшей мере одно полное вычисление, а также еще и передать данные. Иными словами, мы можем использовать многопоточность для распараллеливания вычислений, но аппаратный ресурс для передачи данных у нас все равно есть только один.

А если бы мы разработали вариант системы с четырьмя потоками и выполнили это в SMP-системе с четырьмя процессорами, это выглядело бы примерно так:



Четыре потока, четыре процессора.

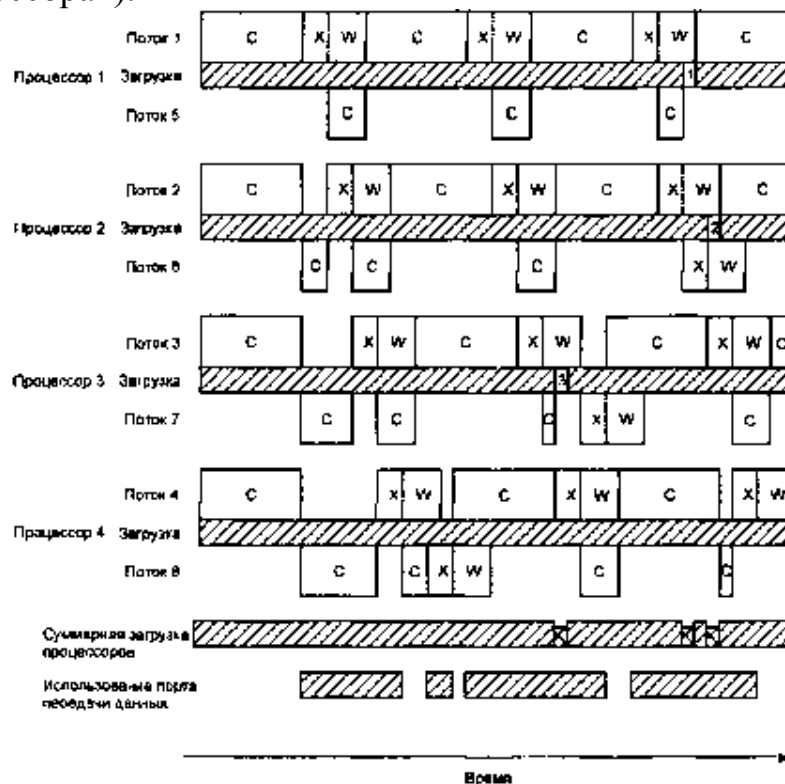
Обратите внимание, насколько каждый из этих четырех центральных процессоров недоиспользован (см. незаштрихованные прямоугольники в строках «Загрузка»). На представленном выше рисунке имеются две интересные зоны. Когда все четыре потока стартуют одновременно, все они вычисляются. К сожалению, когда потоки заканчивают вычисления, они начинают конкурировать за право обладания аппаратными средствами передачи данных (зоны «X» на диаграмме смещены одна относительно другой, поскольку, имея только один передающий ресурс, можно вести только одну передачу одновременно). Это дает нам небольшую аномалию на начальном этапе. После того как потоки отработали этот этап, они оказываются естественным образом синхронизированы по отношению к работе аппаратных средств, так как время передачи данных намного

меньше, чем  $\frac{1}{4}$  времени вычислительного цикла. Если игнорировать эту небольшую аномалию в работе системы на начальном этапе, значения временных интервалов в данной системе можно оценить по формуле:

$$(T_{\text{compute}} + T_{\text{tx}} + T_{\text{wait}}) \cdot \text{num\_x\_lines} / \text{num\_cpus}$$

Из этой формулы следует, что применение четырех потоков на четырех процессорах обеспечивает сокращение затрат времени приблизительно в 4 раза по сравнению с аналогичным временем в модели с единственным потоком, т.е. по сравнению с данным! примера, с которого мы начали обсуждение этой проблемы.

Суммируя все то, что мы узнали из анализа примера с использованием многопоточного варианта с одиночным процессором, в идеале мы желали бы иметь больше потоков, чем процессоров, чтобы дополнительные потоки могли «подобрать» время простоя процессоров, которое естественным образом возникает из интервалов ожидания подтверждения (а также из интервалов ожидания, связанных с конкуренцией за передатчик) В этом случае у нас бы получилось примерно вот что: (см. рис. «Восемь потоков, четыре процессора»).



Восемь потоков, четыре процессора.

На этом рисунке предполагается следующее:

- потоки 5, 6, 7 и 8 привязаны к процессорам 1, 2, 3, и 4 (для упрощения);

- передача данных выполняется с более высоким приоритетом, чем вычислительные операции;
- прервать передачу нельзя.

Из диаграммы видно, что хоть мы теперь и имеем в два раза больше потоков, чем процессоров, мы по-прежнему сталкиваемся с временными интервалами, в течение которых процессоры «недоиспользованы». На рисунке показаны три таких интервала времени. Эти интервалы обозначены числами, соответствующими номеру процессора, и указаны на временных диаграммах загрузки процессоров в строках «Загрузка»:

1. Поток 1 ожидает подтверждения (состояние «W»), при этом поток 5 завершил вычисления и ждет доступности передатчика.

2. Потоки 2 и 6 ожидают подтверждения.

3. Поток 3 ожидает подтверждения, при этом поток 7 завершил вычисления и ждет доступности передатчика.

Этот пример для нас — важный урок. Бессмысленно просто увеличивать количество процессоров в надежде, что все ваши дела пойдут быстрее, поскольку имеются также и ограничивающие факторы. В некоторых случаях эти ограничивающие факторы определяются просто конструкцией материнской платы мультипроцессорной системы, то есть структурой подсистемы разрешения конфликтов за устройства в память, когда несколько процессоров пытаются обратиться по одному и тому же адресу. В нашем случае обратите внимание, что строка «Использование порта передачи данных» стала все больше заполняться. Если бы мы просто увеличили число процессоров, то в конечном счете столкнулись бы с проблемами, связанными с тем, что соответствующие потоки простаивали бы в ожидании передатчика.

В любом случае, используя потоки-«мусорщики» для сбора неиспользованных ресурсов процессоров, мы сможем обеспечить намного более эффективное использование процессоров. Это время приблизительно оценивается по формуле:

$$(T_{\text{compute}} + T_{\text{tx}} + T_{\text{wait}}) \cdot \text{num\_x\_lines} / \text{num\_cpus}$$

При выполнении только вычислений мы ограничены только количеством процессоров; ни один процессор не будет простаивать в ожидании подтверждения. Впрочем, это был бы идеальный случай. Как вы видели из диаграммы, реально периодически возникают временные интервалы, когда один процессор простаивает. Также, как отмечалось ранее, мы в любом случае ограничены по скорости значением:

$$T_{\text{compute}} + T_{\text{tx}} \cdot \text{num\_x\_lines}.$$

## ***На что обратить внимание при использовании симметричного мультипроцессора (SMP)***

При том, что в общем случае вы можете запросто «игнорировать», работаете вы с SMP-архитектурой или с одиночным процессором, есть ряд обстоятельств, которые определенно добавят вам головной боли. К сожалению, это могут быть такие маловероятные события, которые могут проявиться не на этапе разработки, а на этапе его испытаний, в демонстрационных версиях или даже, что самое неприятное, на стадии эксплуатации. Так вот, следование ряду принципов «защитного программирования» избавит вас от связанной с этими проблемами нервозности.

Вот краткий перечень того, что следует четко помнить, имея дело с SMP-системой:

- Потоки действительно могут работать и работают параллельно — ни в коем случае не доверяйте при их синхронизации таким механизмам как диспетчеризация FIFO или система приоритетов.
- Потоки могут также выполняться одновременно с обработчиками прерываний (ISR) — это означает, что вам нужно будет не только защитить поток от обработчика прерываний, но и наоборот — обработчик прерываний от потока. Подробнее об этом см. в главе 4, «Прерывания».
- Некоторые операции, которые по вашему мнению должны быть атомарными, в действительности таковыми не являются — это зависит от операции и от процессора. Отметим из такого списка операции типа «чтение- модификация-запись» (например, ++, --, &=, т.д.). См. файл `<atomic.h>` для анализа возможных замен. (Заметьте, что это не проблема SMP в чистом виде; код для вышеупомянутых операции может выполняться не как атомарный на большинстве RISC-процессоров).

## ***Потоки в независимых ситуациях***

Ранее в разделе «Где хороша многопоточность» говорилось о том, что потокам также находят применение там, где имеет место обработка информации по множеству независимых алгоритмов с разделяемыми структурами данных. При этом, строго говоря, вы могли бы использовать несколько *процессов* (с одним потоком каждый), явно разделяющих данные, но в некоторых случаях вместо этого гораздо удобнее использовать один

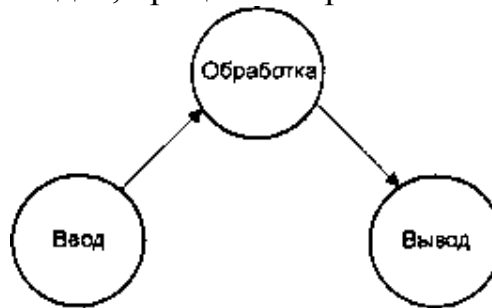


многопоточный процесс. Давайте рассмотрим, почему и где здесь можно использовать потоки.

В наших примерах будем отталкиваться от стандартной модели «ввод-обработка-вывод». В наиболее общем случае одна часть этой модели ответственна за получение откуда-либо входных данных, другая часть — за обработку этих данных и преобразование их в некоторые выходные данные (или управляющие воздействия), третья часть — за отправку полученных выходных данных куда надо.

### *Несколько процессов*

Давайте, во-первых, осмыслим, что мы будем иметь в случае нескольких однопоточных процессов. Для нашей модели у нас было бы три процесса — процесс «ввода», процесс «обработки» и процесс «вывода»:



Система 1: Несколько операций, несколько процессов.

В таком виде наша модель в высшей степени абстрактна, но и в такой же степени «слабо связана». Процесс «ввода» не имеет никакой реальной связи ни с процессом «обработки», ни с процессом «вывода» — он просто отвечает за сбор входных данных и передачу их как-нибудь на следующий этап («этап обработки»).

Мы могли бы сказать то же самое о процессах «обработки» и «вывода» — они также не имеют никакой реальной связи друг с другом. Также здесь предполагается, что обмен данными («ввод — обработка» и «обработка — вывод») осуществляется по некоторому стандартному протоколу (например, через программные каналы, очереди сообщений POSIX, обмен сообщениями QNX/Neutrino — что угодно).

### *Несколько процессов с разделяемой памятью*

В зависимости от объема потока данных, мы можем пожелать оптимизировать характер связей. Самый простой путь состоит в том, чтобы связать три процесса «теснее». Попробуем теперь вместо использования универсального протокола соединения выбрать схему с разделяемой памятью (на диаграмме толстые стрелки указывают потоки данных; тонкие стрелки — потоки управления):

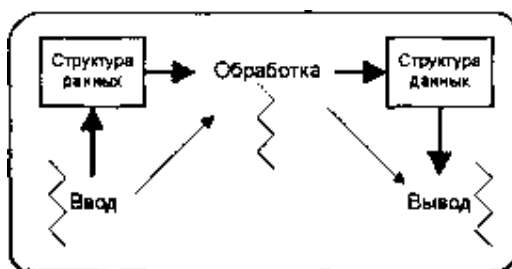


Система 2: Несколько операций, буферы разделяемой памяти между процессами.

В данной схеме мы «подтянули» связь так, чтобы в результате обеспечить более быстрый и более эффективный обмен данными. В то же время, мы здесь по-прежнему можем применять универсальный протокол для передачи «управляющей» информации, поскольку предполагается, что по сравнению с потоком данных ее не так много.

### Несколько потоков

Система с наиболее тесными связями представлена на следующей схеме:



Система 3: Несколько операций, несколько потоков.

Здесь мы наблюдаем один процесс с тремя потоками. Все три потока неявно разделяют области данных. Обмен управляющей информацией может быть реализован аналогично предыдущим примерам или с помощью ряда примитивов синхронизации потоков (мы уже имели дело с мутексами, барьерами и семафорами — скоро рассмотрим и другие).

## Сравнение

Давайте теперь сравним эти три метода по ряду критериев и взвесим все «за» и «против».

В системе 1 связь была самой слабой. Это имеет то преимущество, что каждый из трех процессов может быть легко (то есть при помощи командной строки, в противоположность перекомпиляции/переработке) заменен другим модулем. Это следует из самой природы модели, потому что «единицей модульности» здесь является сам функциональный модуль. Система 1 является также единственной, которая из всех трех может быть распределена по узлам сети QNX/Neutrino. Поскольку информационные связи здесь абстрагированы до некоторого универсального протокола, очевидно, что эти три процесса могут быть выполнены на любой машине в сети. Это может быть очень мощным фактором масштабируемости в Вашем проекте — вам может понадобиться расширить свою сеть до сотен узлов, либо разделенных географически, либо как-то иначе — например, для совместимости с другими аппаратными средствами.

Однако, как только мы переходим к применению разделяемой памяти, мы теряем способность распределять модули по сети. QNX/Neutrino не поддерживает распределенные объекты разделяемой памяти. Таким образом, в Системе 2 мы реально ограничили себя выполнением всех трех процессов на одной и той же машине. Мы не потеряли способность легкой замены или исключения модулей, потому что модули все еще представляют собой отдельные процессы, управляемые командной строкой. Но мы добавили ограничение, в соответствии с которым все заменяемые компоненты должны соответствовать модели с разделяемой памятью.

В системе 3 мы теряем все отмеченные ранее проектные возможности. Мы определенно не можем выполнять различные потоки одного процесса на различных узлах (хотя при этом мы можем выполнять их на различных процессорах в SMP-системе). Также мы потеряли наши возможности переконфигурации — теперь нам обязательно понадобится механизм явного доопределения, который из алгоритмов «ввода», «обработки» и «вывода» мы должны использовать (эту проблему можно решить с помощью разделяемых объектов, также известных как динамические библиотеки — DLL).

Так почему же я должен проектировать свою систему, используя многопоточность, как в Системе 3? Почему бы мне для обеспечения максимальной универсальности не выбрать Систему 1?

Ну, даже при том, что Система 3 является наиболее ригидной, она, скорее всего, окажется самой быстродействующей. В ней не будет переключений контекста между потоками в различных процессах, мне не придется настраивать разделяемую память, а также применять абстрактные методы синхронизации типа программных каналов, очередей сообщений POSIX или обмен сообщениями QNX/Neutrino для обеспечения доставки данных или управляющей информации — я смогу использовать базовые примитивы синхронизации потоков на уровне ядра. Другим преимуществом является то, что при запуске системы, состоящей из одного процесса (с тремя потоками), я могу быть уверен, что все, что мне понадобится далее, уже загружено с носителя (то есть потом не выяснится что-то типа «Опа! А нужного-то драйвера на диске и нету...») И, наконец, Система 3 также, скорее всего, будет наиболее компактной, потому что не придется использовать три отдельных копии информации, характерной для процессов (например, дескрипторы файлов).

Мораль: знайте, какое решение сулит какие выгоды и какие потери, и применяйте то, что будет оптимальным для вашего конкретного проекта.

## Дополнительно о синхронизации

Мы уже обсудили:

- мутексы;
- семафоры;
- барьеры.

Давайте теперь завершим нашу дискуссию о синхронизации, обсудив следующее:

- блокировки чтения/записи (reader/writer locks);
- ждущие блокировки (sleepers);
- условные переменные (condition variables);
- дополнительные сервисы QNX/QNX/Neutrino.

### Блокировки чтения/записи

Блокировки чтения/записи применяются точно в соответствии с их названием: несколько «читателей» могут использовать ресурс в отсутствие «писателей», или один «писатель» может использовать ресурс в отсутствие «читателей» и других «писателей».

Эта ситуация возникает достаточно часто для того, чтобы создать отдельный примитив синхронизации специально для этих целей.

У вас будет часто возникать ситуация разделения структуры данных группой потоков. Очевидно, что в любой момент времени только один поток может записывать данные в эту структуру. Если бы запись велась более чем одним потоком одновременно, одни потоки могли бы записать свои данные поверх данных других потоков. Для предотвращения таких ситуаций поток-«писатель» должен эксклюзивно получить блокировку чтения/записи («rwlock»), обозначив этим, что он и только он имеет доступ к структуре данных. Заметьте, что это исключительное право доступа «строго контролируется на добровольных началах» — обеспечение того, чтобы все потоки, которые пользуются указанной областью данных, синхронизировались с использованием блокировок чтения/ записи, зависит только от вас.

С «читателями» ситуация противоположная. Поскольку считывание области данных — неразрушающая операция, любое число потоков может считывать данные (даже если ту же часть данных в этот момент считывает другой поток). Сложным моментом здесь является то, что никто не должен производить запись в область данных, из которой в этот момент ведется

чтение. В противном случае, считывающие потоки могут быть «введены в заблуждение» — например, поток мог бы считать часть данных, затем быть вытесненным потоком-«писателем» затем возобновиться и продолжить считывание данных, но уже обновленных! Это может закончиться нарушением целостности данных.

Давайте рассмотрим вызовы, которые вы могли бы использовать при применении блокировок чтения/записи.

Первые два вызова используются для инициализации внутренних областей памяти для `rwlock`-блокировок (чтения/записи):

```
int pthread_rwlock_init(pthread_rwlock_t *lock,
    const pthread_rwlockattr_t *attr);
```

```
int pthread_rwlock_destroy(pthread_rwlock_t *lock);
```

Функция `pthread_rwlock_init()` принимает аргумент `lock` (типа `pthread_rwlock_t`) и инициализирует его атрибутами, указанными в параметре `attr`. В нашем примере мы применим атрибут `NULL`, что будет означать «применить значения по умолчанию». Более подробно об этом см. документацию на функции:

```
pthread_rwlockattr_init();
pthread_rwlockattr_destroy();
pthread_rwlockattr_getpshared();
pthread_rwlockattr_setpshared();
```

Когда мы закончим свои дела с блокировкой чтения/записи, её следует уничтожить функцией `pthread_rwlock_destroy()`.

Никогда не используйте блокировку, которая либо уже уничтожена, либо еще не инициализирована.

Далее, мы должны выбрать блокировку подходящего типа. Как отмечалось выше, в основном применяются два режима блокировки: «читателю» желательно иметь «неэксклюзивный» доступ, а для «писателю» — «эксклюзивный». Для упрощения имен, функции названы по именам своих пользователей:

```
int pthread_rwlock_rdlock(pthread_rwlock_t *lock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *lock);
int pthread_rwlock_wrlock(pthread_rwlock_t *lock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *lock);
```

Существует четыре функции блокировки, а не две, как вы могли бы предположить. Очевидно, «предполагаемыми» функциями были `pthread_rwlock_rdlock()` и `pthread_rwlock_wrlock()`, используемые «читателями» и «писателями», соответственно.

Это — собственно блокирующие вызовы: если блокировка для выбранной операции недоступна, поток будет заблокирован. Когда блокировка становится доступной в соответствующем режиме, поток будет разблокирован, из чего он сможет предположить, что теперь можно спокойно обращаться к защищенному блокировкой ресурсу.

Иногда, тем не менее, поток может не захотеть блокироваться, желая вместо этого просто узнать, доступна ли нужная блокировка. Для этого и существуют версии функций, содержащие в имени «try» («проверка»). Важно отметить, что «проверочные» версии получают блокировку, если она доступна, но если нет, тогда они не будут заблокированы, а только возвратят код ошибки. Причина, по которой они должны получать блокировку, если она доступна, очень проста. Предположим, что поток хочет получить блокировку на чтение, но не хочет ждать, если блокировка окажется недоступной. Поток вызывает функцию *pthread\_rwlock\_tryrdlock()*, и оказывается, что блокировка доступна. Если бы функция *pthread\_rwlock\_tryrdlock()* не захватывала доступную блокировку немедленно, могли бы произойти неприятные вещи — наш поток мог бы быть, к примеру, вытеснен другим потоком, а тот, в свою очередь, мог бы заблокировать нужный нам ресурс. Поскольку первому потоку фактически не была предоставлена блокировка, после возобновления ему придется вызывать *pthread\_rwlock\_rdlock()*, и вот теперь он будет заблокирован, поскольку ресурс более недоступен. Иными словами, в такой ситуации даже поток, не желающий блокироваться и поэтому вызывающий «проверочную» версию, по-прежнему может быть заблокирован!

Наконец, независимо от того, как блокировка нами применялась, нам необходим способ ее освобождения:

```
int pthread_rwlock_unlock(pthread_rwlock_t* lock);
```

После того как поток выполнил нужную операцию с ресурсом, он освобождает блокировку, вызывая функцию *pthread\_rwlock\_unlock()*. Если блокировка теперь становится доступной в режиме, который запрошен и ожидается другим потоком, то этот ждущий поток будет переведен в состояние готовности (READY).

Отметим, что мы не смогли бы реализовать такую форму синхронизации только с помощью мутекса. Мутекс рассчитан только на один поток, что было бы хорошо в случае записи (чтобы только один поток мог использовать ресурс в определенный момент времени), но оплошал бы в случае считывания, потому что не допустил бы к ресурсу более чем одного «читателя». Семафор также был бы бесполезен, потому что нельзя было бы отличить два режима доступа — применение семафора могло бы обеспечить доступ нескольких «читателей», но если бы семафором

попытался завладеть «писатель», его вызов ничем бы не отличался от вызова «читателей», что вызвало бы некрасивую ситуацию с множеством «читателей» и множеством же «писателей»!

## Ждущие блокировки

Другая типовая ситуация в многопоточных программах — это потребность заставить поток «ждать чего-либо». Этим «чем-либо» может являться фактически что угодно! Например, когда доступны данные от устройства, или когда конвейерная лента находится в нужной позиции, или когда данные сохранены на диск, и т.д. Еще одна хитрость этой ситуации состоит в том, что одного и того же события могут ожидать несколько потоков.

Для таких целей мы могли бы использовать либо условную переменную (*condition variable*), о которой речь ниже, либо, что гораздо проще, ждущую блокировку (*sleepon*).

Для применения ждущих блокировок надо выполнить несколько операций. Рассмотрим сначала вызовы, а затем вернемся к использованию ждущих блокировок.

```
int pthread_sleepon_lock(void);

int pthread_sleepon_unlock(void);

int pthread_sleepon_broadcast(void *addr);

int pthread_sleepon_signal(void *addr);

int pthread_sleepon_wait(void *addr);
```

⚠ Не дайте префиксу *pthread\_* себя обмануть. Эти функции не предусмотрены стандартами POSIX.

Как было отмечено ранее, потоку может быть необходимо ждать какого-нибудь события. Наиболее очевидный выбор из представленного выше списка функций — это функция *pthread\_sleepon\_wait()*. Но сначала поток должен проверить, надо ли ждать. Давайте приведем пример. Один поток представляет собой поток-«поставщик», который получает данные от неких аппаратных средств. Другой поток — поток-«потребитель» и он неким образом обрабатывает поступающие данные. Рассмотрим сначала поток-«потребитель»:



```
volatile int data_ready = 0;
```

```
consumer() {  
    while (1) {  
        while (!data_ready) {  
            // wait  
        }  
        // Обработать данные  
    }  
}
```

«Потребитель» вечно находится в своем главном обрабатывающем цикле (**while(1)**). Первое, что он проверяет — это флаг *data\_ready*. Если этот флаг равен 0, это означает, что данных нет, и их надо ждать. Впоследствии поток-«производитель» должен будет как-то «разбудить» его, и тогда поток-«потребитель» должен будет повторно проверить состояние флага *data\_ready*. Положим, что происходит именно это. Поток-«потребитель» анализирует состояние флага и определяет, что флаг равен 1, то есть данные теперь доступны. Поток-«потребитель» переходит к обработке поступивших данных, после чего он должен снова проверить, не поступили ли новые данные, и так далее.

Здесь мы можем столкнуться с новой проблемой. Как «потребителю» сбрасывать флаг *data\_ready* согласованно с «производителем»? Очевидно, нам понадобится некоторая форма монопольного доступа к флагу, чтобы в любой момент времени только один из этих потоков мог модифицировать его. Метод, который применен в данном случае, заключается в применении мутекса, но это внутренний мутекс библиотеки ждущих блокировок, так что мы сможем обращаться к нему только с помощью двух функций: *pthread\_sleepon\_lock()* и *pthread\_sleepon\_unlock()*. Давайте модифицируем наш поток-«потребитель»:

```
consumer() {  
    while (1) {  
        pthread_sleepon_lock();  
        while (!data_ready) {  
            // WAIT  
        }  
        // Обработать данные  
        data_ready = 0;  
        pthread_sleepon_unlock();  
    }  
}
```

Здесь мы добавили «потребителю» установку и снятие блокировки. Это означает, что потребитель может теперь *надежно* проверять флаг *data\_ready*, не опасаясь гонок, а также надежно его устанавливать.

Великолепно! А как насчет собственно процесса ожидания? Как мы и предполагали ранее, там действительно применяется вызов функции *pthread\_sleepon\_wait()*. Вот второй while-цикл:

```
while (!data_ready) {  
    pthread_sleepon_wait(&data_ready);  
}
```

Функция *pthread\_sleepon\_wait()* в действительности выполняет три действия:

1. Разблокирует мутекс библиотеки ждущих блокировок.
2. Выполняет собственно операцию ожидания.
3. Снова блокирует мутекс библиотеки ждущих блокировок.

Причина обязательной разблокировки/блокировки мутекса библиотеки проста: поскольку суть мутекса состоит в обеспечении взаимного исключения доступа к флагу *data\_ready*, мы хотим запретить потоку-«производителю» изменять флаг *data\_ready*, пока мы его проверяем. Но если мы не разблокируем флаг впоследствии, то поток-«производитель» не сможет его установить, чтобы сообщить нам о доступности данных! Операция повторной блокировки выполняется автоматически исключительно для удобства, чтобы вызвавший функцию *pthread\_sleepon\_wait()* поток не беспокоился о состоянии блокировки после «пробуждения».

Давайте перейдем теперь к потоку-«производителю» и рассмотрим, как он использует библиотеку ждущих блокировок. Вот его полная реализация:

```
producer() {  
    while (1) {  
        // Ждать прерывания от оборудования...  
        pthread_sleepon_lock();  
        data_ready = 1;  
        pthread_sleepon_signal(&data_ready);  
        pthread_sleepon_unlock();  
    }  
}
```

Как вы видите, поток-«производитель» также блокирует мутекс, чтобы получить монопольный доступ к флагу *data\_ready* перед его установкой.

☞ Клиента «пробуждает» не установка флага *data\_ready* в единицу (1), а вызов функции *pthread\_sleepon\_signal()*!

Давайте рассмотрим происходящее в подробностях. Определим состояния «потребителя» и «производителя» следующим образом:

**Состояние    Означает**

CONDVAR    ожидание соответствующей ждущей блокировке условной переменной  
 MUTEX      ожидание мутекса  
 READY      состояние готовности, т.е., готов выполняться или уже выполняется

INTERRUPT ожидание прерывания от аппаратных средств

Действие	Владелец мутекса	Состояние «потребителя»	Состояние «производителя»
«потребитель» блокирует мутекс	«потребитель»	READY	INTERRUPT
«потребитель» проверяет флаг <i>data_ready</i>	«потребитель»	READY	INTERRUPT
потребитель вызывает функцию <i>pthread_sleepon_wait()</i>	«потребитель»	READY	INTERRUPT
функция <i>pthread_sleepon_wait()</i> разблокирует мутекс	мутекс свободен	READY	INTERRUPT
функция <i>pthread_sleepon_wait()</i> блокируется	мутекс свободен	CONDVAR	INTERRUPT
пауза до прерывания аппаратных средств	мутекс свободен	CONDVAR	INTERRUPT
аппаратные средства генерируют данные	мутекс свободен	CONDVAR	READY
«производитель» блокирует мутекс	«производитель»	CONDVAR	READY
«производитель» устанавливает флаг <i>data_ready</i>	«производитель»	CONDVAR	READY
«производитель»	«производитель»	CONDVAR	READY

вызывает

*pthread\_sleepon\_signal()*

«потребитель»

«пробуждается»,

функция

*pthread\_sleepon\_wait()*

пытается

заблокировать мутекс

«производитель»

разблокирует мутекс

«потребитель»

получает мутекс

«потребитель»

обрабатывает данные

«производитель» ждет

новых данных от

аппаратуры

пауза («потребитель»

обрабатывает

полученные данные)

«потребитель»

завершает обработку и

разблокирует мутекс

«потребитель»

возвращается в начало

цикла и блокирует

мутекс

«производитель» MUTEX

READY

мутекс свободен MUTEX

READY

«потребитель» READY

READY

«потребитель» READY

READY

«потребитель» READY

INTERRUPT

«потребитель» READY

INTERRUPT

мутекс свободен READY

INTERRUPT

«потребитель» READY

INTERRUPT

Последняя строка в таблице повторяет первую — мы совершили один полный цикл.

Каково назначение флага *data\_ready*? Он служит для двух целей:

- Он является флагом состояния — посредником между «потребителем» и «производителем», указывающим на состояние системы. Если флаг установлен в состояние 1, это означает, что данные доступны для обработки; если этот флаг установлено в состояние 0, это означает, что данных нет, и поток-потребитель должен быть заблокирован.

- Он выполняет функцию «места, где происходит синхронизация со ждущей блокировкой». Более формально говоря, адрес переменной *data\_ready* используется как уникальный идентификатор объекта, по

которому осуществляется ждущая блокировка. Мы запросто могли бы применить «(void\*)12345» вместо «&data\_ready» — библиотеке ждущих блокировок все равно, что это за идентификатор, лишь бы он был уникален и корректно использовался. Использование же в качестве идентификатора адреса переменной есть надежный способ сгенерировать уникальный номер, поскольку не бывает же двух переменных с одинаковым адресом!

- К обсуждению различий между функциями *pthread\_sleepon\_signal()* и *pthread\_sleepon\_broadcast()* мы еще вернемся в разговоре об условных переменных.

## Условные переменные

Условные переменные (или «condvars») очень похожи на ждущие блокировки, которые мы рассматривали выше. В действительности, ждущие блокировки — это надстройка над механизмом условных переменных, и именно поэтому в таблице, иллюстрировавшей использование ждущих блокировок, у нас встречалось состояние CONDVAR. Функция *pthread\_cond\_wait()* точно так же освобождает мутекс, ждет, а затем повторно блокирует мутекс, аналогично функции *pthread\_sleepon\_wait()*.

Давайте опустим вступление и обратимся к нашему примеру о «производителе» и «потребителе» из раздела о ждущих блокировках, но вместо ждущих блокировок будем использовать условные переменные. А затем уже обсудим вызовы.

```
/*
 * cp1.c
 */

#include <stdio.h>
#include <pthread.h>

int data_ready = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t condvar = PTHREAD_COND_INITIALIZER;

void* consumer(void *notused){
    printf("Это поток-потребитель...\n");
    while (1) {
        pthread_mutex_lock(&mutex);
```

```

while (!data_ready) {
    pthread_cond_wait(&condvar, &mutex);
}
// Обработать данные
printf("Потребитель: получил данные от производителя\n");
data_ready = 0;
pthread_cond_signal(&condvar);
pthread_mutex_unlock(&mutex);
}
}

void* producer (void *notused) {
    printf("Это поток-производитель...\n");
    while (1) {
        // Получить данные от оборудования
        // (мы имитируем это при помощи sleep(1))
        sleep(1);
        printf("Производитель: получил данные от h/w\n");
        pthread_mutex_lock(&mutex);
        while (data_ready) {
            pthread_cond_wait(&condvar, &mutex);
        }
        data_ready = 1;
        pthread_cond_signal(&condvar);
        pthread_mutex_unlock(&mutex);
    }
}

main() {
    printf(
        "Начало примера с производителем и потребителем...\n");
    // Создать поток-производитель и поток-потребитель
    pthread_create(NULL, NULL, producer, NULL);
    pthread_create(NULL, NULL, consumer, NULL);
    // Дать потокам немного повывполняться
    sleep(20);
}

```

Этот пример в значительной степени похож на программу с применением ждущей блокировки, с небольшими отличиями (мы добавили несколько вызовов *printf()*, а также функцию *main()*, чтобы программа могла

работать!) Первое отличие, которое бросается в глаза, — здесь использован новый тип данных, `pthread_cond_t`. Это просто декларация для условной переменной; мы назвали нашу условную переменную *condvar*.

Следующее, что видно из примера, — это то, что структура «потребителя» идентична таковой в предыдущем примере с ждущей блокировкой. Мы заменили функции *pthread\_sleepon\_lock()* и *pthread\_sleepon\_unlock()* на стандартные мутекс-ориентированные версии (*pthread\_mutex\_lock()* и *pthread\_mutex\_unlock()*). Функция *pthread\_sleepon\_wait()* была заменена на функцию *pthread\_cond\_wait()*.

Основное различие здесь состоит в том, что библиотека ждущих блокировок имеет скрытый внутренний мутекс, а при использовании условных переменных мутекс передается явно. Последний способ дает нам больше гибкости.

И, наконец, обратите внимание на то, что мы использовали функцию *pthread\_cond\_signal()* вместо функции *pthread\_sleepon\_signal()* (опять же, с явной передачей мутекса).

#### **Функции *pthread\*\_signal()* и *pthread\*\_broadcast()***

В разделе о ждущих блокировках мы обещали обсудить различие между функциями *pthread\_sleepon\_broadcast()* и *pthread\_sleepon\_signal()*. Заодно поговорим и о различии между двумя аналогичными функциями, имеющими отношение к условным переменным: *pthread\_cond\_signal()* и *pthread\_cond\_broadcast()*.

В двух словах, функция в варианте «signal» разблокирует только один поток. Например, если бы несколько потоков находилось в ожидании по функции «wait», и некий поток вызвал бы функцию *pthread\*\_signal()*, то был бы разблокирован только один из ждущих потоков. Который из них? Тот, у которого наивысший приоритет. Если имеется два или более потоков с одинаковым приоритетом, порядок «пробуждения» будет не определен. Применение же варианта *pthread\*\_broadcast()* приведет к тому что будут разблокированы все ожидающие потоки.

Разблокировать все потоки может показаться излишним. Но с другой стороны, разблокировать только один (причем случайный поток тоже не совсем корректно).

Поэтому мы должны думать, где имеет смысл использовать какой вариант. Очевидно, что если у вас только один ждущий поток, как у нас и было во всех вариантах «потребителя», функция *pthread\*\_signal()*

прекрасно справится — будет разблокирован один поток, и как раз тот, который нужно (потому что других просто нет).

В ситуации с несколькими потоками в первую очередь следует выяснить: а почему они ждут? Обычно на этот вопрос есть два ответа:

- все потоки рассматриваются как эквивалентные и реально образуют пул доступных потоков, готовых к обработке некоторого запроса;
- все потоки являются уникальными, и каждый из них ждет соблюдения своего специфического условия.

В первом случае мы можем представить себе, что код всех потоков имеет примерно следующий вид:

```
/*
 * cv1.c
 */

#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mutex_data = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cv_data = PTHREAD_COND_INITIALIZER;
int data;

thread1() {
    for (;;) {
        pthread_mutex_lock(&mutex_data);
        while (data == 0) {
            pthread_cond_wait(&cv_data, &mutex_data);
        }
        // Сделать что-нибудь
        pthread_mutex_unlock(&mutex_data);
    }
}
```

В этом случае абсолютно неважно, который именно из потоков получит данные — главное, чтобы хотя бы один сделал это и произвел над этими данными необходимые действия.

Однако, если ваш код подобен приведенному ниже, все будет несколько по-иному:

```
/*
 * cv2.c
 */
```



```

#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mutex_xy = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cv_xy = PTHREAD_COND_INITIALIZER;
int x, y;
int isprime(int);

thread1() {
    for (;;) {
        pthread_mutex_lock(&mutex_xy);
        while ((x > 7) && (y != 15)) {
            pthread_cond_wait(&cv_xy, &mutex_xy);
        }
        // Сделать что-нибудь
        pthread_mutex_unlock(&mutex_xy);
    }
}

thread2() {
    for (;;) {
        pthread_mutex_lock(&mutex_xy);
        while (!isprime(x)) {
            pthread_cond_wait(&cv_xy, &mutex_xy);
        }
        // Сделать что-нибудь
        pthread_mutex_unlock(&mutex_xy);
    }
}

thread3() {
    for (;;) {
        pthread_mutex_lock(&mutex_xy);
        while (x != y) {
            pthread_cond_wait(&cv_xy, &mutex_xy);
        }
        // Сделать что-нибудь
        pthread_mutex_unlock(&mutex_xy);
    }
}

```

В этом случае пробуждение одного потока ничего не даст! Здесь мы обязаны «разбудить» все три потока, чтобы каждый из них проверил соблюдение своего условия.

Это в полной мере отражает второй вариант ответа на наш вопрос «а почему они ждут?» Так как все потоки все ждут соблюдения различных условий (поток *thread1()* ждет, пока значение *x* не станет меньше или равно 7, или пока значение *y* не станет равным 15, поток *thread2()* ждет, пока значение *x* не станет простым числом, а поток *thread3()* ждет, пока *x* не станет равным *y*), у нас нет никакого выбора, кроме как «разбудить» все потоки «одновременно».

### ***Ждущие блокировки в сравнении с условными переменными***

Ждущие блокировки имеют одно основное преимущество в сравнении с условными переменными. Предположим, что вам надо синхронизировать множество объектов. Используя условные переменные, вы бы ассоциировали с каждым объектом отдельную условную переменную — если бы у вас было *M* объектов, вы, скорее всего, определили бы *M* условных переменных. При применении же ждущих блокировок соответствующие им условные переменные создаются динамически по мере постановки потоков на ожидание, поэтому в этом случае на *M* объектов и *N* заблокированных потоков у вас было бы максимум *N*, а не *M* условных переменных.

Однако, условные переменные более универсальны, чем ждущие блокировки, и вот почему:

1. Ждущие блокировки в любом случае основаны на условных переменных.

2. Мутексы ждущих блокировок скрыты в библиотеке; условные переменные позволяют вам задавать его явно.

Первый пункт сам по себе достаточно убедителен. :-) Второй, однако, имеет еще и практический смысл. Когда мутекс скрыт в библиотеке, это означает, что он может быть только один на процесс, независимо от числа потоков в этом процессе или от количества переменных. Это может быть сильно ограничивающим фактором, особенно если принять во внимание, что вам придется использовать один-единственный мутекс для синхронизации доступа всех имеющихся потоков в процессе ко всем нужным им переменным!

Намного лучшая схема состоит в применении нескольких мутексов — по одному на каждый набор данных — и явно сопоставлять им условные

переменные по мере необходимости. Как мощь, так и опасность этого подхода заключаются в том, что ни на этапе компиляции, ни на этапе выполнения не будет производиться никаких проверок, и вам придется самим следить за:

- блокировкой мутексов перед доступом к соответствующим переменным;
- применением правильного мутекса для каждой переменной;
- применением правильной условной переменной для соответствующих мутекса и переменной (данных).

Самый простой путь решения этих проблем — грамотно проектировать и тщательно проверять, а также заимствовать приемы объектно-ориентированного программирования (например, встраивать мутексы в структуры данных, создавать для обращения к структурам данных специализированные подпрограммы, и т.д.). Разумеется, то, в какой степени вы примените первый, второй, или оба варианта, будет зависеть не только от вашего стиля программирования, но и от требований производительности.

Ключевыми моментами при использовании условных переменных являются:

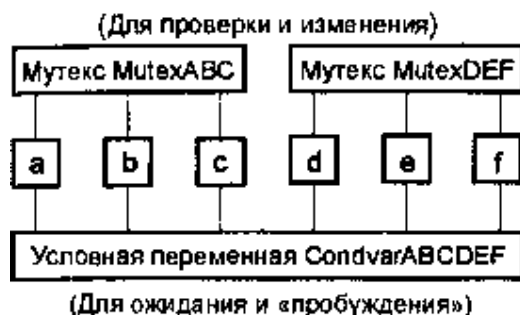
1. Мутексы следует использовать для проверки и изменения переменных.
2. Условные переменные следует использовать в качестве «точки встречи».

Ниже представлена иллюстрация этого:



Связь мутексов и условных переменных по схеме «один к одному»

Одно интересное замечание. Поскольку никаких проверок не выполняется, вы можете, например, связать один набор переменных с мутексом «MutexABC», другой — с мутексом «MutexDEF», и сопоставить обоим наборам переменных одну и ту же условную переменную «CondvarABCDEF»:



Связь мутексов и условных переменных по схеме «один ко многим».

Это весьма полезное свойство. Поскольку мутекс должен использоваться для «проверки и изменения» всегда, это подразумевает, что я должен буду выбрать правильный мутекс всякий раз, когда мне понадобится доступ к некоей переменной. Вполне логично — если я, скажем, проверяю переменную «С», то, очевидно, мне потребуется заблокировать мутекс «MutexABC». А что если я хочу изменить переменную «Е»? Хорошо, перед этим я должен буду захватить мутекс «MutexDEF». Затем я ее изменяю и сообщая об этом другим потокам через условную переменную «CondvarABCDEF», после чего освобождаю мутекс.

А теперь смотрите, что происходит. Толпа потоков, ждавших на условии «CondvarABCDEF», вдруг резко «просыпается» (по функции `pthread_cond_wait()`). Их функции ожидания немедленно пытаются повторно захватить мутекс. Критическим моментом здесь является то, что мутексов два. (В зависимости от того, изменения какой переменной поток ждал, его функция ожидания попытается захватить либо `MutexABC`, либо `MutexDEF` — *прим. ред.*) Это означает, что в SMP-системе возникли бы две *конкурирующие очереди* потоков, и в каждой потоки будут проверять как бы независимые переменные, используя при этом независимые мутексы. Круто, да?

## Дополнительные сервисы QNX/Neutrino

QNX/Neutrino позволяет делать еще ряд изящных вещей. POSIX утверждает, что с мутексом должны работать потоки одного и того же процесса, но позволяет в соответствующей реализации эту концепцию расширять. В QNX/Neutrino это расширение сводится к тому, что мутекс может использоваться потоками *различных процессов*. Чтобы понять, почему это работает, вспомните: то, что мы рассматриваем как «операционную систему», реально состоит из двух частей — ядра, которое занимается диспетчеризацией, и администратора процессов, который, наряду со всем остальным, заботится о защите памяти и «процессах».

Мутекс — всего-навсего объект синхронизации потоков. Поскольку ядро работает только с потоками, то реально ему все равно, какие потоки работают в каких процессах, это уже забота администратора.

Итак, если вы установили область разделяемой памяти между двумя процессами и разместили в ней мутекс, ничто не мешает вам с его помощью синхронизировать потоки в двух (или более!) процессах — функции `pthread_mutex_lock()` и `pthread_mutex_unlock()` будут работать точно так же.

## Пулы потоков

Другое существенное дополнение в QNX/Neutrino — это понятие пула потоков. Вы будете часто обращать внимание в ваших программах на то обстоятельство, что вам хотелось бы иметь несколько потоков и управлять их поведением в определенных пределах. Например, для сервера вы можете решить, что первоначально в ожидании сообщения от клиента должен быть заблокирован только один поток. Когда этот поток получит сообщение и пойдет обслуживать запрос, вы можете принять решение о том, что хорошо было бы создать другой поток и заблокировать его в ожидании на случай поступления другого запроса — тогда этот запрос будет кому обработать. И так далее. Через некоторое время, когда все запросы будут обслужены, у вас может оказаться большое число потоков, бездействующих в ожидании. Чтобы не расходовать ресурсы впустую, вам, возможно, захочется уничтожить некоторые из этих «лишних» потоков.

Подобные операции в жизни — обычное дело, и для задач такого рода QNX/Neutrino предоставляет для этого специальную библиотеку.

☞ В более ранних (до 2.00) версиях QNX/Neutrino была предусмотрена подобная функциональность, но она была скрыта в библиотеке администратора ресурсов. В версии 2.00 эти функции были вынесены из библиотеки администратора ресурсов в отдельную библиотеку. Мы еще вернемся к функциям работы с пулами потоков в главе «Администраторы ресурсов».

В рамках данного обсуждения важно понять, что следует различать два режима потоков в пулах:

- режим блокирования;
- режим обработки.

В режиме блокирования поток обычно вообще не использует ресурсы процессора. В типовом сервере это соответствует ситуации, когда поток

ждет сообщения. Противоположностью этого режима является режим обработки, в котором поток может как использовать, так и не использовать ресурсы процессора — это зависит от структуры процесса. Чуть позже мы рассмотрим функции работы с пулами потоков, и вы увидите, что они дают возможность управлять количеством как блокированных, так и обрабатываемых потоков.

Для работы с пулами потоков в QNX/Neutrino предусмотрены следующие функции:

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

```
thread_pool_t *thread_pool_create(  
    thread_pool_attr_t *attr, unsigned flags);
```

```
int thread_pool_destroy(thread_pool_t *pool);
```

```
int thread_pool_start(void *pool);
```

Как видно из имен функций, вы в первую очередь создаете пул потоков, используя функцию *thread\_pool\_create()*, а затем запускаете этот пул при помощи функции *thread\_pool\_start()*. Когда вы закончили свои дела с пулом потоков, вы можете использовать функцию *thread\_pool\_destroy()* для его уничтожения. Заметьте, что функция *thread\_pool\_destroy()* может вам вообще не понадобиться — например, когда ваша программа суть сервер, который работает «вечно».

Итак, первая функция, на которую следует обратить внимание — это функция *thread\_pool\_create()*. У нее два параметра: *attr* и *flags*. Параметр *attr* — атрибутная запись, которая определяет рабочие параметры пула потоков (см. **<sys/dispatch.h>**):

```
typedef struct _thread_pool_attr {  
    // Функции и дескриптор пула потоков  
    THREAD_POOL_HANDLE_T *handle;  
    THREAD_POOL_PARAM_T *(*block_func)  
        (THREAD_POOL_PARAM_T *ctp);  
    void (*unblock_func) (THREAD_POOL_PARAM_T *ctp);  
    int (*handler_func) (THREAD_POOL_PARAM_T *ctp);  
    THREAD_POOL_PARAM_T *(*context_alloc)  
        (THREAD_POOL_HANDLE_T *handle);  
    void *(*context_free) (THREAD_POOL_PARAM_T *ctp);  
    // Параметры пула потоков  
    pthread_attr_t *attr;  
    unsigned short lo_water;
```

```

unsigned short increment;
unsigned short hi_water;
unsigned short maximum;
} thread_pool_attr_t;

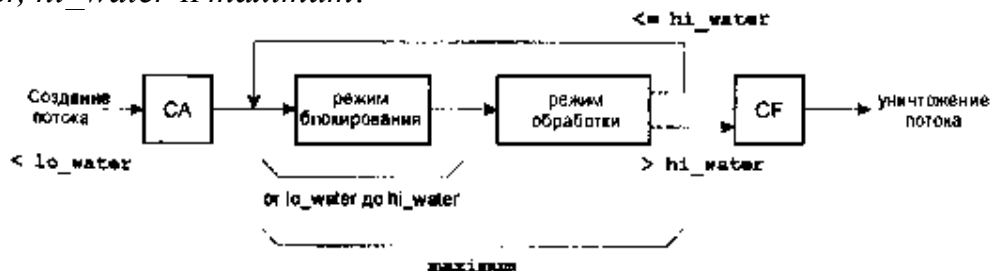
```

Я разбил определение типа `thread_pool_attr_t` на два раздела, один из которых содержит функции и дескриптор для потоков в пуле, а в другом — рабочие параметры пула.

### Управление числом потоков

Сначала проанализируем «параметры пула потоков», чтобы понять, как можно управлять числом потоков в пуле и их атрибутами. Имейте в виду, что здесь мы будем говорить о «режиме блокирования» и «режиме обработки» (далее, когда мы будем рассматривать функции исходящих вызовов (callout functions), мы увидим, как эти режимы соотносятся).

Приведенный ниже рисунок иллюстрирует связи между параметрами `lo_water`, `hi_water` и `maximum`.



Жизненный цикл потока в пуле потоков.

(Заметьте, что как «CA» здесь обозначается функция `context_alloc()`, как «CF» — функция `context_free()`, как «режим блокирования» — функция `block_func()`, а как «режим обработки» — функция `handler_func()`).

Это атрибутная запись, которая применяется при создании потока. Мы уже обсуждали эту структуру ранее (в разделе «Атрибутная запись потока»). Вспомните — это та самая структура, которая задает характеристики нового потока: приоритет, размер стека, и т.д.

**lo\_water** (От «Low watermark», буквально — «нижняя ватерлиния» — прим. ред.) Этот параметр задает минимальное количество потоков, которые должны находиться в режиме блокирования. В типовом сервере это было бы количество потоков, например, ждущих запроса. Если число ждущих потоков меньше, чем значение параметра `lo_water`, (например, потому что мы только

что приняли сообщение, и один из ждущих потоков переключился на его обработку), тогда создается дополнительно еще *increment* потоков. Это представлено на рисунке в виде первого этапа, обозначенного как «создание потока».

*increment* (Буквально — «приращение» — *прим. ред.*) Этот параметр определяет, сколько потоков должны быть созданы сразу, если число потоков, находящихся в режиме блокирования, становится меньше значения параметра *lo\_water*. В выборе значения для этого параметра вы бы наиболее вероятно начали со значения 1 (единица). Это означало бы, что если бы число потоков в режиме блокирования стало бы меньше значения параметра *lo\_water*, то пулом потоков был бы создан дополнительно ровно один поток. Для более тонкой настройки параметра *increment* можно понаблюдать за поведением процесса и определить, может ли этому параметру понадобиться принимать значения, отличные от единицы. Например, если ваш процесс периодически получает «всплески» запросов, то из того, что число потоков, находящихся в режиме блокирования, упало ниже значения *lo\_water*, можно было бы сделать вывод как раз о таком «всплеске» и принять решение о создании более чем одного резервного потока.

*hi\_water* (От «high watermark», буквально — «верхняя ватерлиния» — *прим. ред.*) Этот параметр указывает верхний предел числа потоков, которые могут быть в режиме блокирования одновременно. По мере завершения своих операций по обработке данных, потоки обычно будут возвращаться в режим блокирования. Однако, у библиотеки поддержки пулов потоков есть внутренний счетчик числа потоков, находящихся в режиме блокирования, и если его значение превышает значение параметра *hi\_water*, библиотека автоматически уничтожит поток, который вызвал переполнение (то есть тот поток, который только что завершил обработку и намеревался возвратиться в режим блокирования). Это показано на рисунке раздвоением стрелки, исходящей из блока «режим обработки» — одна стрелка ведет к «режиму блокирования», а вторая — к блоку операции «CF» и далее на уничтожение потока. Таким образом, сочетание параметров *lo\_water* и *hi\_water* позволяет вам четко определять диапазон числа потоков, одновременно находящихся в режиме блокирования.

*maximim* Параметр указывает на максимальное число потоков, которые



вообще могут работать одновременно в результате действий библиотеки поддержки пулов потоков. Например, при создании новых потоков в случае их нехватки (когда число заблокированных потоков падает ниже границы *lo\_water*) общее количество потоков было бы ограничено параметром *maximum*.

Другой ключевой параметр, предназначенный для управления потоками, — это параметр *flags*, передаваемый функции *thread\_pool\_create()*. Он может принимать одно из следующих значений:

*POOL\_FLAG\_EXIT\_SELF*

Не делать возврат из функции *thread\_pool\_start()* и не включать вызывающий поток в пул.

*POOL\_FLAG\_USE\_SELF*

Не делать возврат из функции *thread\_pool\_start()*, но включить вызывающий поток в пул.

0

Функция *thread\_pool\_start()* возвратится, новые потоки будут создаваться по мере необходимости.

Приведенное описание может показаться суховатым. Давайте рассмотрим пример.

В управляющей структуре пула потоков сконцентрируем наше внимание только на значениях параметров *lo\_water*, *increment* и *maximum*:

```
/*
 * tp1.c
 *
 * Пример с пулами потоков (1)
 *
 */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/neutrino.h>
#include <sys/dispatch.h>
```

```
char *programe = "tp1";
```

```
void tag (char *name) {
    time_t t;
    char buffer[BUFSIZ];
```

```

time(&t);
strftime(buffer, BUFSIZ, "%T ", localtime(&t));
printf("%s %3d %-20.20s: ", buffer, pthread_self(), name);
}

```

```

THREAD_POOL_PARAM_T* blockfunc(
    THREAD_POOL_PARAM_T *ctp) {
    tag("blockfunc");
    printf("ctp %p\n", ctp);
    tag("blockfunc");
    printf("sleep (%d);\n", 15 * pthread_self());
    sleep(pthread_self() * 15);
    tag("blockfunc");
    printf("Выполнили sleep\n");
    tag("blockfunc");
    printf("Возвращаем 0x%08X\n",
        0x10000000 + pthread_self());
    return((void*)(0x10000000 + pthread_self()));
    // Передано handlerfunc
}

```

```

THREAD_POOL_PARAM_T* contextalloc(
    THREAD_POOL_HANDLE_T *handle) {
    tag("contextalloc");
    printf("handle %p\n", handle);
    tag("contextalloc");
    printf("Возвращаем 0x%08X\n",
        0x20000000 + pthread_self());
    return ((void*)(0x20000000 + pthread_self()));
    // Передано blockfunc
}

```

```

void contextfree(THREAD_POOL_PARAM_T *param) {
    tag("contextfree");
    printf("param %p\n", param);
}

```

```

void unblockfunc(THREAD_POOL_PARAM_T *ctp) {
    tag("unblockfunc");
    printf("ctp %p\n", ctp);
}

```

```

}

int handlerfunc(THREAD_POOL_PARAM_T *ctp) {
    static int i = 0;
    tag("handlerfunc");
    printf("ctp %p\n", ctp);
    if (i++ > 15) {
        tag("handlerfunc");
        printf("Более 15 операций, возвращаем 0\n");
        return (0);
    }
    tag("handlerfunc");
    printf("sleep (%d)\n", pthread_self() * 25);
    sleep(pthread_self() * 25);
    tag("handlerfunc");
    printf("Выполнили sleep\n");
    tag("handlerfunc");
    printf("Возвращаем 0x%08X\n",
        0x30000000 + pthread_self());
    return (0x30000000 + pthread_self());
}

main() {
    thread_pool_attr_t tp_attr;
    void *tpp;
    memset(&tp_attr, 0, sizeof(tp_attr));
    tp_attr.handle = (void*)0x12345678;
    // Передано contextalloc
    tp_attr.block_func = blockfunc;
    tp_attr.unblock_func = unblockfunc;
    tp_attr.context_alloc = contextalloc;
    tp_attr.context_free = contextfree;
    tp_attr.handler_func = handlerfunc;
    tp_attr.lo_water = 3;
    tp_attr.hi_water = 7;
    tp_attr.increment = 2;
    tp_attr.maximum = 10;
    if ((tpp =
        thread_pool_create(&tp_attr, POOL_FLAG_USE_SELF)) ==
        NULL) {

```

```

fprintf(stderr,
    "%s: Ошибка thread_pool_create, errno %s\n",
    progname, strerror(errno));
exit(EXIT_FAILURE);
}
thread_pool_start(tpp);
fprintf(stderr,
    "%s: возврат из thread_pool_start; errno %s\n",
    progname, strerror(errno));
sleep(3000);
exit(EXIT_FAILURE);
}

```

После установки параметров мы вызываем функцию *thread\_pool\_create()* для создания пула потоков. Эта функция возвращает указатель на управляющую структуру пула потоков (*tpp*), который мы проверяем на равенство NULL (что указало бы на ошибку). И, наконец, мы вызываем функцию *thread\_pool\_start()*, передав ей эту самую управляющую структуру *tpp*.

Я указал флаг *POOL\_FLAG\_USE\_SELF*, что означает, что поток, вызвавший функцию *thread\_pool\_start()*, будет рассматриваться как доступный для ввода в пул. Таким образом, на момент старта пула в нем есть только один поток. Поскольку значение параметра *lo\_water* равно 3, библиотека немедленно создаст еще *increment* потоков (в нашем случае — 2). С этого момента в пуле будет три (3) потока, и все они будут находиться в режиме блокирования. Условие по параметру *lo\_water* удовлетворено, потому что число потоков в режиме блокирования действительно не меньше *lo\_water*, условие по параметру *hi\_water* удовлетворено, потому что число потоков в режиме блокирования действительно не больше *hi\_water*; и, наконец, также удовлетворено условие по параметру *maximum*, потому что общее число потоков не превышает его значения. Допустим теперь, что один из потоков, находящихся в режиме блокирования, разблокируется (например, в серверном приложении — при получении сообщения). Это означает, что один из трех потоков перейдет из режима блокирования в режим обработки. Счетчик заблокированных потоков уменьшится, и его значение упадет ниже значения параметра *lo\_water*. Это переключит триггер *lo\_water* и заставит библиотеку создать ещё *increment* (2) потоков. Таким образом, у нас будет всего 5 потоков (4 в режиме блокирования, и 1 — в режиме обработки).

Пусть далее разблокируется еще несколько потоков. Давайте предположим, что на этот момент еще ни один из потоков, находящихся в

режиме обработки, еще не завершил свои дела. Ниже приведена таблица, в которой иллюстрируется весь процесс, начиная с исходного состояния:

Событие	Режим обработки	Режим блокирования	Всего потоков
Исходное состояние	0	1	1
Срабатывание триггера <i>lo_water</i>	0	3	3
Разблокирование	1	2	3
Срабатывание триггера <i>lo_water</i>	1	4	5
Разблокирование	2	3	5
Разблокирование	3	2	5
Срабатывание триггера <i>lo_water</i>	3	4	7
Разблокирование	4	3	7
Разблокирование	5	2	7
Срабатывание триггера <i>lo_water</i>	5	4	9
Разблокирование	6	3	9
Разблокирование	7	2	9
Срабатывание триггера <i>lo_water</i>	7	3	10
Разблокирование	8	2	10
Разблокирование	9	1	10
Разблокирование	10	0	10

Видно, что библиотека проверяет параметр *lo\_water*, и по мере необходимости увеличивает число потоков на значение параметра *increment*, но только до тех пор, пока число потоков не достигнет предельного значения — параметра *maximum* (именно поэтому число в столбце «Всего потоков» никогда не превышает 10, даже когда условие по параметру *lo\_water* перестает выполняться).

Это означает, что однажды наступает момент, когда потоков в режиме блокирования больше не остается. Предположим теперь, что потоки, находящиеся в режиме обработки, завершают свои дела. Посмотрим, что при этом произойдет с триггером параметра *hi\_water*.

Событие	Режим	Режим	Всего
---------	-------	-------	-------

	<b>обработки</b>	<b>блокирования</b>	<b>потоков</b>
Завершение обработки	9	1	10
Завершение обработки	8	2	10
Завершение обработки	7	3	10
Завершение обработки	6	4	10
Завершение обработки	5	5	10
Завершение обработки	4	6	10
Завершение обработки	3	7	10
Завершение обработки	2	8	10
Срабатывание триггера <i>hi_water</i>	2	7	9
Завершение обработки	1	8	9
Срабатывание триггера <i>hi_water</i>	1	7	9
Завершение обработки	0	8	8
Срабатывание триггера <i>hi_water</i>	0	7	7

Обратите внимание, что с потоками ничего не происходит до тех пор, пока число заблокированных потоков не превышает значение *hi\_water*. Реализация здесь такова: как только поток завершает обработку, он проверяет число заблокированных на данный момент потоков, и если их слишком много (то есть больше, чем предусмотрено параметром *hi\_water*), то «совершает самоубийство». Удобство использования параметров *lo\_water* и *hi\_water* в управляющих структурах состоит в том, что ими вы фактически задаете «эффективный диапазон» числа потоков, в пределах которого всегда доступно достаточное число потоков, и потоки без необходимости не создаются и не уничтожаются. В нашем случае, после выполнения действий, перечисленных в вышеупомянутых таблицах, мы имеем систему, которая способна обрабатывать до 4 запросов одновременно без необходимости в создании дополнительных потоков ( $7 - 4 = 3$ , что соответствует значению параметра *lo\_water*).

### Функции работы с пулами потоков

Теперь, когда мы достаточно хорошо владеем методикой управления числом потоков в пуле, давайте обратимся к другим элементам атрибутной

записи пула потоков:

```
// Функции и дескриптор пула потоков
THREAD_POOL_HANDLE_T *handle;

THREAD_POOL_PARAM_T *(*block_func) (
    THREAD_POOL_PARAM_T *ctp);

void (*unblock_func) (THREAD_POOL_PARAM_T *ctp);

int (*handler_func) (THREAD_POOL_PARAM_T *ctp);

THREAD_POOL_PARAM_T *(*context_alloc) (
    THREAD_POOL_HANDLE_T *handle);

void (*context_free) (THREAD_POOL_PARAM_T *ctp);
```

Повторно обратимся к рисунку «Жизненный цикл пула потоков». Из рисунка видно, что при создании потока каждый раз вызывается функция *context\_alloc()*. (Аналогично, при уничтожении потока вызывается функция *context\_free()*). Элемент атрибутной записи с именем *handler* передается функции *context\_alloc()* в качестве ее единственного параметра. Функция *context\_alloc()* ответственна за индивидуальные настройки потока и возвращает указатель на контекст (списках параметров называемый *ctp*). Заметьте, что содержание этого указателя — исключительно ваша забота; библиотеке абсолютно все равно, что вы в него поместите.

Теперь, когда контекст создан функцией *context\_alloc()*, вызывается функция *block\_func()* для перевода потока в режим блокирования. Заметьте, что функция *block\_func()* получает на вход результат работы функции *context\_alloc()*. После того как функция *block\_func()* разблокируется, она возвращает указатель на контекст, который библиотека передает функции *handler\_func()*. Функция *handler\_func()* отвечает за выполнение «работы» — например, в типовом варианте именно она обрабатывает сообщение от клиента. На данный момент принято, что функция *handler\_func()* должна возвращать нуль — ненулевые значения зарезервированы QSSL для будущего функционального расширения. Функция *unblock\_func()* также в настоящее время зарезервирована, поэтому просто оставьте там NULL.

Возможно, ситуацию немного прояснит приведенный ниже пример псевдокода (он основан все на том же рисунке «Жизненный цикл потока в пуле потоков»):

```
FOREVER DO
    IF (#threads < lo_water) THEN
```

```
IF (#threads < maximum) THEN
    create new thread
    context = (*context_alloc)(handle);
ENDIF
ENDIF
retval = (*block_func)(context);
(*handler_func)(retval);
IF (#threads > hi_water) THEN
    (*context_free)(context)
    kill thread
ENDIF
DONE
```

Отметим, что приведенная выше программа излишне упрощена. Ее назначение состоит только в том, чтобы продемонстрировать вам поток данных по параметрам *ctx* и *handler* и дать вам некоторое представление об алгоритмах, которые обычно применяются для управления числом потоков.



## Диспетчеризация и реальный мир

До настоящего момента мы обсуждали дисциплины диспетчеризации и состояния потоков, но практически ничего не сказали относительно того, почему и когда происходит собственно перепланирование. Существует распространенное заблуждение, что перепланирование «просто случается», безо всяких реальных причин. И в общем-то, для проектирования это довольно полезная абстракция! Однако, очень важно понимать, почему происходит перепланирование. Вспомним рисунок «Схема алгоритма диспетчеризации» (в разделе «Роль ядра»).

Перепланирование может иметь только три причины:

- аппаратное прерывание;
- системный вызов;
- сбой (исключение).

### Перепланирование по аппаратному прерыванию

Перепланирование из-за аппаратного прерывания можно разделить на две категории:

- по прерыванию от таймеров;
- по прерыванию от других аппаратных средств.

Часы реального времени генерируют периодические прерывания для ядра, организуя перепланирование во времени.

Например, если вы производите вызов `sleep(10)`, часы реального времени сгенерируют некоторое число прерываний; по каждому прерыванию ядро увеличивает значение системных часов. Когда системные часы покажут, что 10 секунд истекли, ядро перепланирует ваш поток, переведя его в состояние готовности (READY). (Мы рассмотрим этот вопрос более подробно в главе «Часы, таймеры и периодические уведомления»).

Другие потоки могут ожидать аппаратные прерывания от внешних устройств, таких как последовательный порт, жесткий диск или аудио платы. В этом случае они блокируются в ядре, ожидающем аппаратное прерывание. Поток будет переупорядочен ядром только после того, как ядро сгенерирует «событие».

### Перепланирование по системным вызовам

Если поток делает системный вызов, перепланирование выполняется немедленно и может рассматриваться как асинхронное в отношении прерываний таймера и других прерываний.

Например, выше мы приводили пример вызова функции `sleep(10)`. Это библиотечная функция языка Си, в конечном счете она транслируется в системный вызов. В тот же самый момент ядро приняло решение о перепланировании, чтобы удалить ваш поток из очереди готовности по соответствующему приоритету и поставить на выполнение другой поток, находящийся в состоянии готовности (READY).

Системных вызовов, вызывающих процесс обязательного перепланирования, очень много. Большинство из них достаточно очевидны. Перечислим некоторые из них:

- функции таймера (например, `sleep()`);
- функции обмена сообщениями (например, `MsgSendv()`);
- примитивы работы с потоками (например, `pthread_cancel()` или `pthread_join()`).

## Перепланирование по исключительным ситуациям

Последняя из вышеперечисленных причин перепланирования — это сбой процессора (CPU fault), который является исключительной ситуацией (exception) — чем-то средним между аппаратным прерыванием и системным вызовом. Исключительные ситуации асинхронны в отношении ядра (подобно прерыванию), но синхронны с вызывающими их пользовательскими программами (подобно вызову ядра — например, такая исключительная ситуация как деление на ноль). Все рассуждения, относящиеся к перепланированию по прерываниям от аппаратных средств и по системным вызовам, относятся и к исключительным ситуациям тоже.

## Резюме

Операционная система QNX/Neutrino предлагает богатые возможности диспетчеризации потоков — минимальных диспетчеризуемых единиц. Процесс в QNX/Neutrino определяется как минимальная единица, способная обладать ресурсами (например, областями памяти), и может содержать один или более потоков.

С потоками можно применять любые из следующих методов синхронизации:

- мутексы (mutexes) — владеть мутексом в заданный момент времени может только один поток;
- семафоры (semaphores) — владеть семафором позволяет некоторому фиксированному числу потоков;
- ждущие блокировки (sleepers) — позволяют нескольким потокам блокироваться на нескольких объектах, динамически назначая заблокированным потокам соответствующие условные переменные;
- условные переменные (condvars) — подобны ждущим блокировкам, за исключением того, что за распределение условных переменных отвечает программист;
- присоединение (joining) — обеспечивает синхронизацию потока по отношению к завершению другого потока;
- барьеры (barriers) — позволяют потокам ждать, пока определенное число потоков не встретится в определенной точке.

Отметим, что мутексы, семафоры и условные переменные могут использоваться между потоками как в том же самом, так и в разных процессах, ждущие же блокировки могут применяться только между потоками одного и того же процесса (потому что системный мутекс библиотеки ждущих блокировок «скрыт» в адресном пространстве процесса).

Наряду с синхронизацией, потоки можно диспетчеризовать (используя приоритеты и различные дисциплины диспетчеризации), и они автоматически могут выполняться как в однопроцессорном блоке, так и в системе с архитектурой SMP.

Всякий раз, когда мы говорим о «создании процесса» (обычно как о средстве переноса однопоточного кода), мы действительно создаем адресное пространство с одним работающим в нем потоком — этот поток стартует по вызову функции *main()* или функций *atfork()* или *vfork()*, в зависимости от реализации.

## **Глава 2**

### **Обмен сообщениями**

## Введение в обмен сообщениями

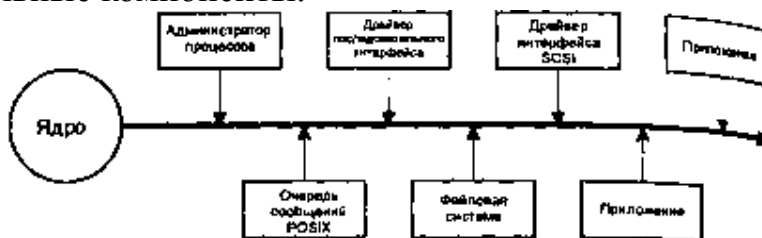
В данной главе мы рассмотрим наиболее характерную отличительную особенность QNX/Neutrino — механизм *обмена сообщениями*. Обмен сообщениями в QNX/Neutrino — ключевой механизм, глубоко интегрированный с микроядерной архитектурой этой операционной системы и обеспечивающий ей ее модульность.

## Микроядро и обмен сообщениями

Одним из основных преимуществ QNX/Neutrino является то, что данная операционная система является масштабируемой. Под «масштабируемостью» здесь подразумевается, что данная система может быть адаптирована к работе как в крошечных встраиваемых системах с ограниченными ресурсами, так и в больших сетях симметричных многопроцессорных систем (SMP), т.е. систем, ресурсы которых практически неограничены.

В операционной системе QNX/Neutrino такой уровень универсальности достигается разнесением различных сервисов по отдельным модулям. Таким образом, вы имеете возможность включить в конечную систему только те компоненты, которые вам действительно необходимы. Используя многопоточность, вы также упрощаете своему проекту «масштабируемость вверх» для использования его в SMP-системах (в данной главе мы рассмотрим еще ряд полезных применений для потоков, которые мы не обсуждали ранее).

Эта концепция была изначально заложена во все ОС семейства QNX и соблюдается по сей день. Основным принципом построения этих систем является микроядерная архитектура, когда модули, которые в традиционной операционной системе были бы включены в состав ядра, рассматриваются как необязательные компоненты.



Модульная архитектура QNX/Neutrino.

Какие из модулей применить в проекте — это уже решать проектировщику, то есть вам. В вашем проекте необходима файловая система? Если да, включите ее в проект. Если нет — можете не включать. Вам необходим драйвер последовательного порта? Каков бы ни был ваш ответ, он не повлияет на предыдущее решение касательно файловой системы, равно как и не будет от него зависеть.

В процессе работы системы вы также имеете возможность изменять ее состав. Вы можете динамически удалять любые компоненты из работающей системы или добавлять их, в любой произвольный момент времени. Вы спросите, существуют ли какие-либо ограничения относительно «драйверов»? Нет, не существуют — драйвер в QNX/Neutrino является стандартной пользовательской программой, которая разве что выполняет определенные действия с оборудованием. Мы обсудим, как писать такие программы, в главе «Администраторы ресурсов».

Ключом к реализации всего этого является обмен сообщениями. Вместо встраивания модулей ОС непосредственно в ядро и обеспечения между ними некоего «специального» взаимодействия, в QNX/Neutrino модули общаются друг с другом посредством обмена сообщениями. Ядро в основном отвечает только за служебные функции на уровне потоков (например, за диспетчеризацию потоков). На самом деле, обмен сообщениями используется не только для трюков с динамической инсталляцией и деинсталляцией компонентов — на нем основаны большинство всех остальных служебных функций (например, распределение памяти выполняется путем отправки специализированного сообщения администратору процессов). Впрочем, конечно, некоторые сервисы реализуются непосредственно через системные вызовы.

Рассмотрим открытие файла и запись в него блока данных. Это реализуется с помощью ряда сообщений, посылаемых приложением такому опциональному компоненту ОС как файловая система. Сообщение указывает файловой системе открыть файл, затем другое сообщение указывает ей записать в него некие данные. И не волнуйтесь — обмен сообщениями в QNX/Neutrino происходит очень быстро.

## Обмен сообщениями и модель «клиент/сервер»

Представьте, что приложение читает данные из файловой системы. На языке QNX это значит, что данное приложение — это *клиент*, запрашивающий данные у *сервера*.

Модель «клиент/сервер» позволяет говорить о нескольких рабочих состояниях процессов, связанных с обменом сообщениями (мы говорили о них в главе «Процессы и потоки»). Первоначально сервер ждет от кого-нибудь сообщение. В этот момент сервер, как говорят, должен быть в состоянии блокировки по приему (recieve-blocked) (оно также может обозначаться как RECV). Ниже приведен пример вывода программы `pidin`:

```
pid tid name      prio STATE  Blocked
4   1  devc-pty  10r  RECEIVE    1
```

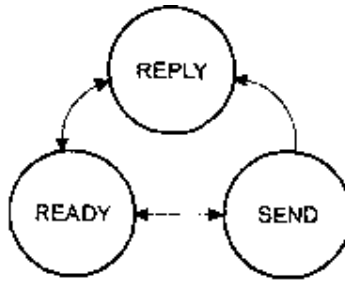
В приведенном примере сервер псевдотерминалов (называемый `devc-pty`) имеет идентификатор процесса 4, содержит один поток с идентификатором потока 1, выполняется с приоритетом 10, подчиняется диспетчеризации карусельного типа (RR) и находится в состоянии блокировки по приему, ожидая сообщения по каналу с идентификатором 1 (к «каналам» мы еще скоро вернемся).



Смена состояний сервера.

По получении сообщения сервер переходит в состояние готовности (READY) и становится способен выполнять работу. Если оказывается, что из всех процессов, находящихся в данный момент в состоянии готовности (READY), наш сервер имеет наивысший приоритет, он получит процессор и сможет выполнить какие-то действия. Поскольку это сервер, он анализирует поступившее сообщение и решает, что с ним делать. В некоторый момент времени сервер завершит обработку сообщения и затем «ответит» клиенту.

Перейдем теперь к клиенту. Изначально клиент работал самостоятельно, пока не решил послать сообщение. Клиент переключается при этом из состояния готовности (READY) в состояние либо блокировки по передаче (send-blocked), либо блокировки по приему (recieve-blocked), в зависимости от состояния сервера, которому было послано сообщение.



#### Смена состояний клиента

Скорее всего, вам чаще придется иметь дело с состоянием блокировки по приему (reply-blocked), чем с состоянием блокировки по передаче (send-blocked). Состояние блокировки по приему (reply-blocked) реально означает следующее:

Сервер принял сообщение и теперь обрабатывает его. В некоторый момент времени сервер завершит обработку и ответит клиенту. Клиент блокирован в ожидании этого ответа от сервера.

Сравните с состоянием блокировки по передаче (send-blocked):

Сервер все еще не принял сообщение — вероятно, потому что был занят обработкой другого, ранее поступившего сообщения. Когда сервер возвратится в состояние «приема» вашего (клиентского) сообщения, вы перейдете из состояния блокировки по передаче (send-blocked) в состояние блокировки по ответу (reply-blocked).

На практике, если вы наблюдаете процесс, заблокированный по передаче (send-blocked), это означает одно из двух:

1. Вы запечатлели момент, когда сервер был занят обслуживанием некоего клиента и в это время получил еще один запрос.

Это нормальная ситуация — вы можете проверить это, повторно выполнив `pidin`. На сей раз вы, вероятно, сможете увидеть, что этот процесс уже более не заблокирован по передаче.

2. В сервере проявилась какая-то внутренняя ошибка, и он больше не воспринимает запросы.

Когда это произойдет, вы сможете увидеть множество процессов, заблокированных по передаче на этом сервере. Чтобы проверить это, выполните `pidin` снова и посмотрите, есть ли изменения в состоянии клиентских процессов.



Ниже приведен пример, в котором показан клиент в состоянии блокировки по ответу (reply-blocked) и сервер, по которому он блокирован:

pid	tid	name	prio	STATE	Blocked
1	1	/nto/x86/sys/procnto	0f	READY	
1	2	/nto/x86/sys/procnto	10r	RECEIVE	1
1	3	/nto/x86/sys/procnto	10r	NANOSLEEP	
1	4	/nto/x86/sys/procnto	10r	RUNNING	
1	5	/nto/x86/sys/procnto	15r	RECEIVE	1
16426	1	esh	10r	REPLY	1

В примере показано, что программа **esh** (встраиваемый командный интерпретатор) передала сообщение процессу с номером 1 (это ядро и администратор процессов, **procnto**) и теперь ждет ответа.

Ну вот, теперь вы знаете основы обмена сообщениями в архитектуре «клиент/сервер».

Не исключено, что вы сейчас думаете: «Так что, получается, чтобы открыть файл или записать данные, мне придется писать специализированные вызовы обмена сообщениями QNX/ Neutrino?!»

Нет, вам не придется программировать обмен сообщениями непосредственно — разве что если вам будет нужно копнуть совсем вглубь (об этом несколько позже). Действительно, позвольте мне показать Вам некоторую программу клиента, который делает передачу сообщений:

```
#include <fcntl.h>
#include <unistd.h>

int main(void) {
    int fd;
    fd = open("filename", O_WRONLY);
    write(fd, "Это обмен сообщениями\n", 24);
    close(fd);
    return (EXIT_SUCCESS);
}
```

Видите? Обычная Си-программа, никаких хитростей.

Собственно обмен сообщениями реализован в Си-библиотеке QNX/Neutrino. Вы просто выдаете вызовы по стандарту POSIX 1003.1 и вызовы функций ANSI Си и Си-библиотека делает за Вас всю работу, связанную с обменом сообщениями.

В приведенном выше примере вызываются три функции, и посылаются три различных сообщения:

- *open()* — передала сообщение «open» («открыть»);
- *write()* — передала сообщение «write» («записать»);

- *close()* — передала сообщение «close» («заккрыть»).

Мы обсудим сами сообщения более подробно, когда мы будем изучать администраторы ресурсов (в главе «Администраторы ресурсов»), а пока что единственное, что нам надо знать об этом — это сам факт, что были переданы сообщения различных типов.

Давайте на мгновение отвлечемся и сравним этот подход с тем, как бы это работало в традиционной операционной системе.

Клиентская программа осталась бы такой же — различия были бы скрыты в Си-библиотеке, поставляемой производителем программного обеспечения. В такой системе функция *open()* сделала бы системный вызов, ядро затем обратилось бы непосредственно к файловой системе, которая, в свою очередь, выполнила некоторые действия и возвратила бы дескриптор файла. Вызовы функций *write()* и *close()* работали бы аналогично

Итак? Есть ли преимущества в способе, который предлагает QNX/Neutrino? «Оставайтесь с нами!»

## Распределенный обмен сообщениями

Предположим, что мы пожелаем изменить приведенный выше пример, чтобы можно было «поговорить» с другим узлом сети. Вы, наверное, думаете, что для этого придется вызывать специальные функции, чтобы «попасть в сеть». Вот сетевой вариант нашей программы:

```
#include <fcntl.h>
#include <unistd.h>

int main(void) {
    int fd;
    fd = open("/net/wintermute/home/rk/filename", O_WRONLY);
    write(fd, "Это обмен сообщениями\n", 24);
    close(fd);
    return (EXIT_SUCCESS);
}
```

Вы будете правы, если скажете, что в обеих версиях программы почти одинаковы. Так и есть.

В традиционной ОС функция *open()* библиотеки Си вызывает ядро, которое анализирует имя файла и говорит: «Опа! Это не на нашем узле...» Ядро затем вызывает сетевую файловую систему NFS, которая уже определяет, где в действительности находится файл */net/wintermute/home/rk/filename*. Затем, NFS вызывает сетевой драйвер и посылает сообщение ядру на узле *wintermute*, которое повторяет весь процесс, описанный нами в нашем первоначальном примере. Заметьте, что в этом случае оказываются вовлеченными две файловые системы, одна из которых — сетевая файловая система (NFS) клиента, а вторая — удаленная. К сожалению, в зависимости от реализации как удаленной файловой системы, так и NFS, некоторые операции (например, блокировки файлов) могут работать некорректно из-за неполной совместимости.

В QNX/Neutrino функция *open()* Си-библиотеки создает точно такое же сообщение, какое она создала бы для локальной файловой системы, и посылает его файловой системе узла *wintermute*. Локальная и удаленная файловые системы при этом абсолютно одинаковы.

Это и есть еще одна фундаментальная особенность QNX/Neutrino: распределенные операции выполняются в ней абсолютно «непринужденно», поскольку потребности клиентов изначально

абстрагированы от служебных функций, обеспечиваемых серверами, благодаря механизму обмена сообщениями.

В традиционном ядре действует «двойной стандарт», когда локальные сервисы реализуются одним способом, а удаленные (сетевые) — совершенно другим.

## Что это означает для вас

Обмен сообщениями в QNX/Neutrino элегантно реализован и распределен по сети. И что? Что с этого нам, программистам?

Ну, в первую очередь это означает, что ваши программы унаследуют от ОС те же характеристики — сделать их распределенными будет гораздо проще, чем в других ОС. Но самое полезное, на мой взгляд, преимущество заключается в том, что эта схема обеспечивает модульную структуру программного обеспечения, тем самым значительно упрощая процесс отладки и тестирования.

Вам, вероятно, доводилось участвовать в больших проектах, когда множество людей разрабатывают различные фрагменты целевого программного обеспечения. Разумеется, при этом кто-то опережает график, а кто-то запаздывает.

У таких проектов проблемы чаще всего возникают на двух этапах: на первоначальном, при распределении отдельных частей проекта между конкретными исполнителями, а также на этапе тестирования и интеграции, когда невозможно провести комплексные испытания системы из-за недоступности всех необходимых компонентов.

С использованием принципа обмена сообщениями развязать друг от друга отдельные компоненты проекта становится очень просто, что ведет к значительному упрощению как самого проекта, так и технологии тестирования. Если говорить об этом в терминах существующих парадигм, данный подход очень похож на концепции, применяемые в объектно-ориентированном программировании (ООП).

К чему все это сводится? К тому, что тестирование можно выполнять поэтапно. Вы сможете написать простенькую программку, которая посылает сообщения вашему серверному процессу, а поскольку его входы и выходы являются (или должны быть!) хорошо задокументированными, то вы сможете сразу определить, работает он или нет. Черт возьми, можно даже создать типовые тестовые наборы, включить их в комплект для регрессивного тестирования и выполнять его в автоматическом режиме!

## Философия QNX/Neutrino

Принципы обмена сообщениями лежат в самой основе философии QNX/Neutrino. Понимание смысла и приемов применения обмена

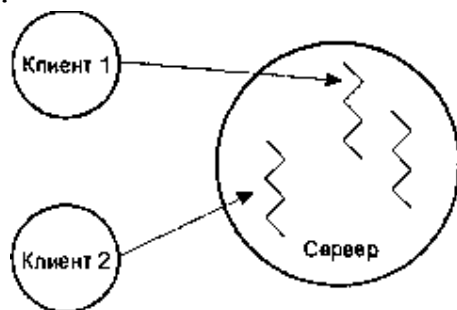
сообщениями будет ключом к наиболее эффективному использованию ОС. Прежде чем углубиться в детали, давайте рассмотрим немного теории.

## Обмен сообщениями и многопоточность

При том, что модель «клиент/сервер» проста для понимания и очень широко используется, существуют две вариации на данную тему. Первая — многопоточная реализация (об этом речь в данной главе), вторая — так называемая модель «сервер/субсервер», иногда полезная и в обычных разработках, но в полной мере раскрывающая свои преимущества при проектировании распределенных систем. Сочетание этих двух концепций предоставляет колоссальную мощь, особенно в сетях симметричных мультипроцессорных систем!

Как мы уже обсуждали в главе «Процессы и потоки», QNX/Neutrino позволяет реализовать множество потоков в одном и том же процессе. Какие преимущества это нам даст в сочетании с механизмом обмена сообщениями?

Ответ здесь довольно прост. Мы можем стартовать пул потоков (используя функции семейства *thread\_pool\_\**()), о которых мы говорили в разделе «Процессы и потоки»), каждый из которых сможет обрабатывать сообщения от клиентов:



### Обслуживание клиентов различными потоками сервера

С этой точки зрения нам абсолютно все равно, которому именно потоку достанется на обработку отправленное клиентом сообщение — главное, чтобы работа была выполнена. Это имеет ряд преимуществ. Способность обслуживать нескольких клиентов отдельными потоками обработки является очень мощной концепцией по сравнению с применением лишь одного потока. Главное преимущество состоит в том, что задача распараллеливания обработки перелagается на ядро, избавляя сам сервер от необходимости реализовывать параллельную обработку самостоятельно.

Когда несколько потоков работают на машине с единственным процессором, это значит, что все эти потоки будут конкурировать друг с другом за процессорное время.

Однако, в SMP-блоке мы можем сделать так, чтобы потоки конкурировали не за один, а за несколько процессоров, используя при этом одну и ту же общую область данных. Это означает, что здесь мы будем ограничены исключительно числом доступных процессоров.

### **Модель «сервер/субсервер»**

Давайте теперь рассмотрим модель «сервер/субсервер», а затем наложим ее на модель многопоточности.

В соответствии с моделью «сервер/субсервер», сервер по-прежнему обеспечивает обслуживание клиентуры, но поскольку обслуживание запросов может занимать слишком много времени, мы должны быть способны поставить запрос на обработку и при этом не потерять способность обрабатывать новые запросы, продолжающие поступать от других клиентов.

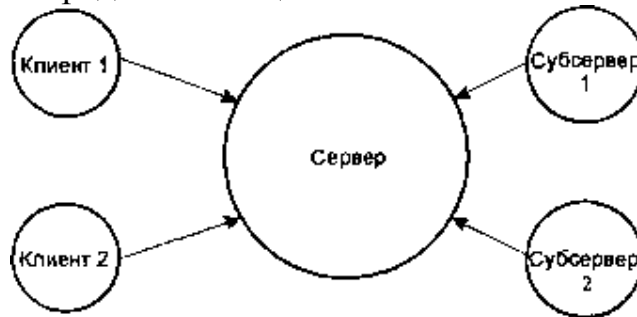
Если бы мы попытались реализовать эту задачу с применением традиционной однопоточной модели «клиент/сервер», то после получения одного запроса и начала его обработки мы потеряли бы способность воспринимать другие запросы — нам приходилось бы периодически прекращать обработку, проверять, есть ли еще запросы, помещать таковые (если они есть) в очередь заданий и затем продолжать обработку, уже распыляя внимание на обработку всевозможных заданий, находящихся в очереди. Не очень-то эффективно. Фактически мы здесь дублируем работу ядра путем реализации дополнительного квантования времени между заданиями!

Представьте себе, каково вам было бы делать все это самому. Вы сидите за своим рабочим столом в офисе, и кто-то приносит вам полную папку заданий на выполнение. Вы начинаете их выполнять. Вы по уши заняты работой, и тут в дверном проеме появляется кто-то еще, с еще одним списком не менее первоочередных заданий. Теперь у вас на рабочем столе имеется два списка неотложных дел, и вы продолжаете работать, стараясь выполнить все. Вы тратите несколько минут на работы из одного списка, потом переключаете внимание на другой, и так далее, периодически посматривая на дверной проем, на случай если кто-нибудь принесет еще.

В такой ситуации было бы гораздо разумнее как раз применить модель «сервер/субсервер». В соответствии с этой моделью, у нас есть сервер, который создает ряд других процессов (субсерверов). Каждый из этих субсерверов посылает сообщение серверу, но сервер не отвечает им, пока не получит запрос от клиента. Затем сервер передает запрос клиента



одному из субсерверов, отвечая на его сообщение заданием, которое субсервер обязан выполнить. Этот процесс показан на приведенном ниже рисунке. Отметьте для себя направления стрелок — они соответствуют направлениям передачи сообщений!



Модель «сервер/субсервер».

Если бы вы делали что-то подобное, вы бы, скорее всего, наняли дополнительно несколько служащих. Эти служащие все пришли бы к вам (как субсерверы посылают сообщение серверу — отсюда направление стрелок на рисунке) в поисках, чего бы такого сделать. Первоначально у вас могло и не быть для них никакой работы — в таком случае их запросы остались бы без ответа. Но теперь, когда кто-нибудь принесет вам кипу бумаг, вы скажете одному из ваших подчиненных: «Это тебе!» — и подчиненный пойдет заниматься делом. Аналогично, по мере поступления других заданий вы и далее будете делегировать их остальным подчиненным.

Хитрость этой модели заключается в том, что она является управляемой по ответу (reply-driven) — выполнение задания начинается с вашего ответа (reply) субсерверу. Стандартная же модель «клиент/сервер» является управляемой по запросу (send-driven), поскольку работа начинается с передачи сообщения серверу.

Но почему клиенты приходят именно к вам в офис, а не в офисы нанятых вами работников? Почему именно вы распределяете работу? Ответ довольно прост: вы — координатор, ответственный за определенную задачу, и ваша обязанность — гарантировать ее выполнение. Ваши клиенты, заказывающие вам работу, знают Вас, но не знают ни имен, ни местонахождения ваших (возможно, временных) работников.

Как вы, вероятно, и подозревали, концепцию единого многопоточного сервера и модель «сервер/субсервер» можно комбинировать. Главная хитрость при этом будет в определении того, какие части задачи лучше было бы распределить по машинам в сети (обычно это касается компонентов системы, которые не генерируют большого трафика), а какие

— по процессорам SMP-архитектур (чаще всего это элементы, требующие наличия разделяемой области данных).

Зачем можно было бы комбинировать эти два метода? Используя подход «сервер/субсервер», мы сможем распределять работу между узлами сети. Это в действительности означает, что мы ограничены только числом доступных в сети машин (ну, и полосой пропускания сети, разумеется).

Объединение этот подход с принципом распределения потоков по различным процессорам в архитектурах SMP, мы получим «вычислительный кластер», где центральный «арбитр» распределяет работу (в модели «сервер/субсервер») между блоками SMP, объединенными в сеть.

## Несколько примеров

Рассмотрим теперь несколько примеров применения каждого метода.

### *Режим с управлением по запросу (send-driven) — модель «клиент/сервер»*

Файловая система, последовательные порты, консоли и звуковые платы — все это примеры применения модели «клиент/сервер». Прикладная программа на языке Си берет на себя роль клиента и посылает запросы этим серверам. Серверы выполняют работу и отвечают клиентам.

Некоторые из этих «обычных» серверов, однако, в действительности могут быть серверами, управляемыми по ответу (reply-driven)! Это возможно, например, в случае, когда по отношению к конечному клиенту они выглядят как стандартные серверы, а вот работу выполняют по методике «сервер/ субсервер». То есть я имею в виду, что клиент по-прежнему посылает сообщение тому, кого считает «серверным процессом», а тот просто передает работу другому процессу (субсерверу).

### *Режим с управлением по ответу (reply-driven) — модель «сервер/субсервер»*

Один из наиболее популярных примеров программы, управляемой по ответу (reply-driven), — это программа фрактальной графики, распределенная по сети. Ведущая программа делит экран на несколько зон

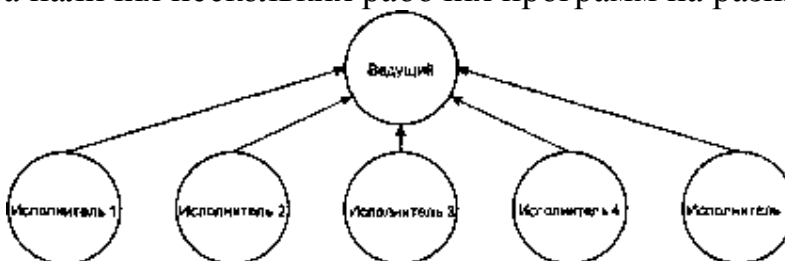
— например, на 64 зоны. При старте ведущей программе задается список узлов, которые могут участвовать в работе. Затем ведущая программа запускает рабочие программы (субсерверы), по одной на каждый узел, и ждет от них сообщений.

Затем ведущая программа по очереди берет «незаполненные» зоны (из имеющихся 64) и передает задачу фрактальных вычислений программе-исполнителю на другом узле, отвечая ей на ее сообщение. Когда рабочая программа завершит вычисления, она посылает результаты обратно ведущей, которая выводит их на экран.

Поскольку программа-исполнитель передала результаты ведущей программе путем отправки ей сообщения, она теперь снова готова получить от нее ответ с новым заданием. Ведущая программа так и делает до тех пор, пока все 64 зоны на экране не будут заполнены.

### ***Важная тонкость***

Поскольку ведущая программа отвечает за распределение работы между программами-исполнителям, она не может себе позволить быть заблокированной! При традиционном подходе с управлением по запросу (send-driven) ведущая программа должна была бы создать программу-исполнителя и послать ей сообщение. К сожалению, при этом ведущая программа не сможет получить ответ до тех пор, пока программа-исполнитель не выполнит свою работу, а значит, не сможет и передать сообщение другой программе-исполнителю. Это сразу сводит на нет все преимущества наличия нескольких рабочих программ на разных узлах.



Один ведущий, несколько исполнителей

Решение этой проблемы заключается в том, чтобы исполнители при старте запросили ведущего, есть ли для них работа, послав ему сообщение. Напомним еще раз, что направление стрелок на рисунке указывает направление передачи. Теперь исполнители ждут ответа от ведущего. Когда какой-нибудь клиент «заказывает работу» ведущему, тот отвечает одному или более из исполнителей, что указывает им выйти из ожидания и начать

выполнение. Это позволяет исполнителям заботиться о своих делах самостоятельно, а ведущий сохраняет возможность отвечать на новые запросы, поскольку не блокируется в ожидании ответа от исполнителей.

### *Многопоточный сервер*

С позиции клиента многопоточные серверы неотличимы от однопоточных. Фактически, разработчик сервера может запросто «включить многопоточность», запустив еще один или несколько потоков.

В любом случае, сервер может по-прежнему использовать несколько процессоров в SMP-системе, даже если «клиент» у него только один.

Что это означает? Давайте вернемся к примеру о фрактальной графике. Когда субсервер получает от сервера запрос на «вычисления», ему ничто не мешает запустить несколько потоков и начать обработку данного запроса на нескольких процессорах сразу. На самом деле, чтобы приложение лучше масштабировалось в сетях, в которых есть как мультипроцессоры SMP, так и однопроцессорные машины, сервер и субсервер могут сначала обмениваться информацией о том, сколько у субсервера имеется в распоряжении процессоров. Это даст серверу знать, сколько запросов субсервер может обслужить одновременно. Тогда сервер сможет перенаправлять многопроцессорным субсерверам больше запросов, чем однопроцессорным, равномерно распределяя нагрузку между вычислительными мощностями.

## Применение обмена сообщениями

Теперь, когда мы рассмотрели базовые концепции обмена сообщениями и выяснили, что он используется даже в таких обычных повседневных вещах как библиотека языка Си, давайте рассмотрим кое-какие детали.

### Архитектура и структура

Мы рассуждали о «клиентах» и «серверах». Я также использовал три ключевые выражения:

- «Клиент *посылает* (sends) сообщение серверу»;
- «Сервер *принимает* (receives) сообщение от клиента»;
- «Сервер *отвечает* (replies) клиенту».

Я преднамеренно использовал именно эти выражения, потому что они в точности соответствуют действительным именам функций, которые используются для передачи сообщений в QNX/Neutrino.

Ниже приводится (в алфавитном порядке) полный список функций QNX/Neutrino, относящихся к обмену сообщениями:

- *ChannelCreate()*, *ChannelDestroy()*;
- *ConnectAttach()*, *ConnectDetach()*;
- *MsgDeliverEvent()*;
- *MsgError()*;
- *MsgRead()*, *MsgReadv()*;
- *MsgRecieve()*, *MsgRecievePulse()*, *MsgRecievev()*;
- *MsgReply()*, *MsgReplyv()*;
- *MsgSend()*, *MsgSendc()*, *MsgSendsv()*, *MsgSendsvnc()*, *MsgSendvs()*, *MsgSendvsnc()*, *MsgSendv()*, *MsgSendvnc()*;
- *MsgWrite()*, *MsgWritev()*.

Пусть вас не приводит в замешательство размер списка функций. Вы запросто сможете писать приложения «клиент/сервер», используя лишь небольшое подмножество этого списка, просто по мере углубления в детали вы поймете, что некоторые из вышеперечисленных функций могут оказаться очень полезными в определенных случаях.

☞ Минимальный полезный набор функций включает в себя функции *ChannelCreate()*, *ConnectAttach()*, *MsgReply()*, *MsgSend()*

и *MsgRecieve()*.

Разобьем обсуждение на две части: отдельно обсудим функции, которые применяются на стороне клиента, и отдельно — те, что применяются на стороне сервера.

## Клиент

Клиент, который желает послать запрос серверу, блокируется до тех пор, пока сервер не завершит обработку запроса. Затем, после завершения сервером обработки запроса, клиент разблокируется, чтобы принять «ответ».

Это подразумевает обеспечение двух условий: клиент должен «уметь» сначала установить соединение с сервером, а потом обмениваться с ним данными с помощью сообщений — как в одну сторону (запрос — «send»), так и в другую (ответ — «reply»).

## Установление соединения

Итак, рассмотрим теперь функции по порядку. Первое, что мы должны сделать — это установить соединение. Это мы сделаем с помощью функции *ConnectAttach()*, описанной следующим образом:

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

```
int ConnectAttach(int nd, pid_t pid, int chid,  
    unsigned index, int flags);
```

Функции *ConnectAttach()* передаются три идентификатора: идентификатор *nd* — дескриптор узла (Node Descriptor), идентификатор *pid* — идентификатор процесса (process ID) и идентификатор *chid* — идентификатор канала (channel ID).

Вместе эти три идентификатора, которые обычно записываются в виде «ND/PID/CHID», однозначно идентифицируют сервер, с которым клиент желает соединиться. Аргументы *index* и *flags* мы здесь просто проигнорируем (установим их в ноль).

Итак, предположим, что мы хотим подсоединиться к процессу, находящемуся на нашем узле и имеющего идентификатор 77, по каналу с идентификатором 1. Ниже приведен пример программы для выполнения этого:

```
int coid;
coid = ConnectAttach(0, 77, 1, 0, 0);
```

Можно видеть, что присвоением идентификатору узла (*nd*) нулевого значения мы сообщаем ядру о том, что мы желаем установить соединение на локальном узле.

☞ Как я узнал, что соединиться надо с процессом 77 и по каналу 1? К этому мы скоро вернемся (см. ниже «Поиск сервера по ND/PID/CHID»).

С этого момента у меня есть идентификатор соединения — небольшое целое число, которое однозначно идентифицирует соединение моего клиента с конкретным сервером по заданному каналу.

Я смогу применять этот идентификатор для отправки запросов серверу сколько угодно раз. Выполнив все, для чего предназначалось соединение, я смогу уничтожить его с помощью функции:

```
ConnectDetach(coid);
```

Итак, давайте рассмотрим, как я воспользуюсь этим на практике.

### *Передача сообщений (sending)*

Передача сообщения со стороны клиента осуществляется применением какой-либо функции из семейства *MsgSend\**().

Мы рассмотрим это на примере простейшей из них — *MsgSend()*:

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

```
int MsgSend(int coid, const void *smsg, int sbytes,
            void *rmsg, int rbytes);
```

Аргументами функции *MsgSend()* являются :

- идентификатор соединения с целевым сервером (*coid*);
- указатель на передаваемое сообщение (*smsg*);
- размер передаваемого сообщения (*sbytes*);
- указатель на буфер для ответного сообщения (*rmsg*);
- размер ответного сообщения (*rbytes*);

Что может быть проще!

Передадим сообщение процессу с идентификатором 77 по каналу 1:

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

```
char *smsg = "Это буфер вывода";
```

```

char rmsg[200];
int coid;

// Установить соединение
coid = ConnectAttach(0, 77, 1, 0, 0);
if (coid == -1) {
    fprintf(stderr, "Ошибка ConnectAttach к 0/77/1!\n");
    perror(NULL);
    exit(EXIT_FAILURE);
}

// Послать сообщение
if (MsgSend(
    coid, smsg, strlen(smsg) + 1, rmsg, sizeof(rmsg)) == -1) {
    fprintf(stderr, "Ошибка MsgSend\n");
    perror(NULL);
    exit(EXIT_FAILURE);
}
if (strlen(rmsg) > 0) {
    printf("Процесс с ID 77 возвратил \"%s\"\n", rmsg);
}

```

Предположим, что процесс с идентификатором 77 был действительно активным сервером, ожидающим сообщение именно такого формата по каналу с идентификатором 1. После приема сообщения сервер обрабатывает его и в некоторый момент времени выдает ответ с результатами обработки. В этот момент функция *MsgSend()* должна вернуть ноль (0), указывая этим, что все прошло успешно. Если бы сервер послал нам в ответ какие-то данные, мы смогли бы вывести их на экран с помощью последней строки в программе (с тем предположением, что обратно мы получаем корректную ASCIIZ-строку).

## Сервер

Теперь, когда мы рассмотрели клиента, перейдем к серверу. Клиент использовал функцию *ConnectAttach()* для создания соединения с сервером, а затем использовал функцию *MsgSend()* для передачи сообщений.

## Создание канала



Под этим подразумевается, что сервер должен создать канал — то, к чему присоединялся клиент, когда вызывал функцию *ConnectAttach()*. Обычно сервер, однажды создав канал, приберегает его «впрок».

Канал создается с помощью функции *ChannelCreate()* и уничтожается с помощью функции *ChannelDestroy()*:

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

```
int ChannelCreate(unsigned flags);
```

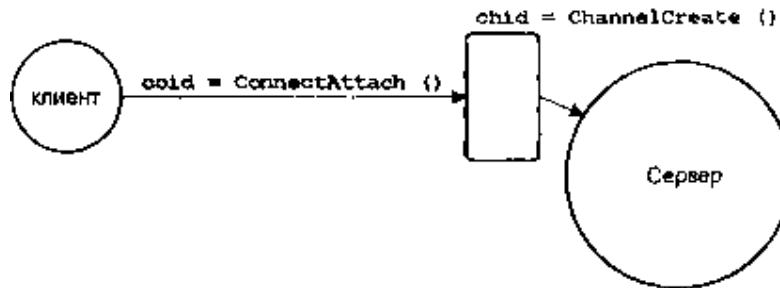
```
int ChannelDestroy(int chid);
```

Мы еще вернемся к обсуждению аргумента *flags* (в разделе «Флаги каналов», см. ниже), а покамест будем использовать для него значение 0 (ноль). Таким образом, для создания канала сервер должен сделать так:

```
int chid;
```

```
chid = ChannelCreate(0);
```

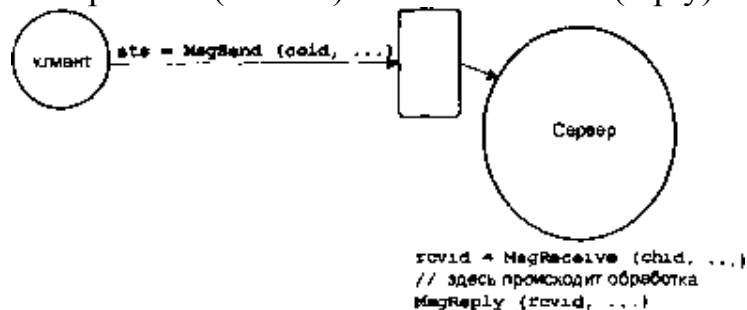
Теперь у нас есть канал. В этом пункте клиенты могут подсоединиться (с помощью функции *ConnectAttach()*) к этому каналу и начать передачу сообщений:



Связь между каналом сервера и клиентским соединением.

### Обработка сообщений

В терминах обмена сообщениями, сервер отрабатывает схему обмена в два этапа — этап «приема» (receive) и этап «ответа» (reply).



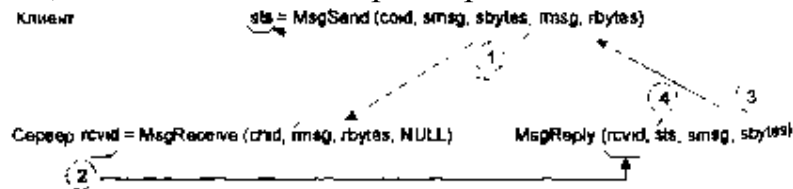
Взаимосвязь функций клиента и сервера при обмене сообщениями.

Обсудим сначала два простейших варианта соответствующих функций, *MsgReceive()* и *MsgReply()*, а далее посмотрим, какие есть варианты.

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

```
int MsgReceive(int chid, void *rmsg, int rbytes,
    struct _msg_info *info);
int MsgReply(int rcvid, int status, const void *msg,
    int nbytes);
```

Посмотрим, как соотносятся параметры:



Поток данных при обмене сообщениями.

Как видно из рисунка, имеются четыре элемента, которые мы должны обсудить:

1. Клиент вызывает функцию *MsgSend()* и указывает ей на буфер передачи (указателем *smsg* и длиной *sbytes*). Данные передаются в буфер функции *MsgReceive()* на стороне сервера, по адресу *rmsg* и длиной *rbytes*. Клиент блокируется.

2. Функция *MsgReceive()* сервера разблокируется и возвращает идентификатор отправителя *rcvid*, который будет впоследствии использован для ответа. Теперь сервер может использовать полученные от клиента данные.

3. Сервер завершил обработку сообщения и теперь использует идентификатор отправителя *rcvid*, полученный от функции *MsgReceive()*, передавая его функции *MsgReply()*. Заметьте, что местоположение данных для передачи функции *MsgReply()* задается как указатель на буфер (*smsg*) определенного размера (*sbytes*). Ядро передает данные клиенту.

4. Наконец, ядро передает параметр *sts*, который используется функцией *MsgSend()* клиента как возвращаемое значение. После этого клиент разблокируется.

Вы, возможно, заметили, что для каждой буферной передачи указываются два размера (в случае запроса от клиента клиента это *sbytes* на стороне клиента и *rbytes* на стороне сервера; в случае ответа сервера это *sbytes* на стороне сервера и *rbytes* на стороне клиента). Это сделано для того, чтобы разработчики каждого компонента смогли определить размеры своих буферов — из соображений дополнительной безопасности.

В нашем примере размер буфера функции *MsgSend()* совпадал с длиной строки сообщения. Давайте теперь рассмотрим, что происходит в сервере и как размер используется там.

### Структура сервера

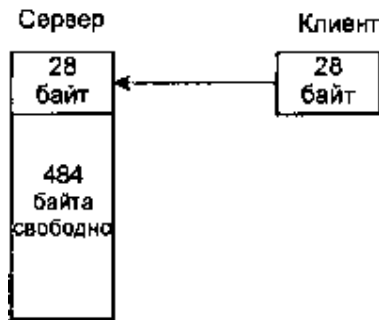
Вот общая структура сервера:

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

...

void server(void) {
    int rcvid; // Указывает, кому надо отвечать
    int chid; // Идентификатор канала
    char message[512]; // Достаточно велик
    // Создать канал
    chid = ChannelCreate(0);
    // Выполняться вечно — для сервера это обычное дело
    while (1) {
        // Получить и вывести сообщение
        rcvid = MsgReceive(chid, message, sizeof(message), NULL);
        printf("Получил сообщение, rcvid %X\n", rcvid);
        printf("Сообщение такое: \"%s\".\n", message);
        // Подготовить ответ — используем тот же буфер
        strcpy(message, "Это ответ");
        MsgReply(rcvid, EOK, message, sizeof(message));
    }
}
```

Как видно из программы, функция *MsgReceive()* сообщает ядру о том, что она может обрабатывать сообщения размером вплоть до **sizeof(message)** (или 512 байт). Наш клиент (представленный выше) передал только 28 байт (длина строки). На приведенном ниже рисунке это и показано:



Передача меньшего объема данных, чем предполагается.

Ядро реально передает *минимум* из двух указанных размеров. В нашем случае ядро передало бы 28 байт, сервер бы разблокировался и отобразил сообщение клиента. Оставшиеся 484 байта (из буфера длиной 512 байт) остались бы нетронутыми.

Аналогичная ситуация с функцией *MsgReply()*. Функция *MsgReply()* информирует, что собирается передать 512 байт, но функция *MsgSend()* определила, что может принять максимум 200 байт. Ядро опять передает минимум. В этом случае 200 байтов, которые клиент может принять, ограничивают размер передачи. (Один интересный аспект здесь состоит в том, что когда сервер передаст данные, то если клиент не примет их целиком, как в нашем примере, их уже нельзя будет вернуть — они будут потеряны.).

☞ Имейте в виду, что такое «урезание» является стандартным и ожидаемым поведением.

Когда мы будем обсуждать обмен сообщениями по сети, вы увидите, что в количестве передаваемых данных есть кое-какое «ага». Мы проанализируем это далее в разделе «Особенности обмена сообщениями в сети».

## Иерархический принцип обмена (send-иерархия)

В обмене сообщениями есть одна вещь, которая, возможно, не является очевидной — это необходимость следовать строгой иерархии обмена. Означает это то, что два потока никогда не должны посылать сообщения друг другу; наоборот, они должны быть организованы так, что каждый поток занимал свой «уровень иерархии», и все потоки данного уровня должны посылать сообщения только потокам более низкого уровня, а не своего или высшего. Проблема с наличием двух потоков, которые посылают сообщения друг другу, заключается в том, что в конечном счете вы столкнетесь с проблемой взаимной блокировки (deadlock), когда оба

потока ожидают друг от друга ответ на соответствующие сообщения. Поскольку эти потоки заблокированы, то они никогда не будут поставлены на выполнение, а значит, не смогут дать друг другу ответ, и вы в результате получите два (а то и более!) зависших потока.

Способ назначения потокам уровней иерархии заключается в том, чтобы разместить наиболее удаленную клиентуру на самом верхнем уровне и работать оттуда. Например, если у вас есть графический интерфейс пользователя, который использует некоторый сервер баз данных, который, в свою очередь, использует файловую систему, а файловая система использует блок-ориентированный драйвер файловой системы, то у вас получается естественная иерархия процессов.

Передачи (sends) при обмене сообщениями будут направлены от клиента (графического интерфейса пользователя) вниз к серверам нижнего уровня; ответы на сообщения (replies) будут иметь встречное направление.

При том, что это работает в большинстве случаев, вы можете столкнуться и с ситуацией, когда вам придется нарушить иерархию обмена. Это никогда не следует выполнять простым нарушением иерархии, направляя сообщения «против течения» — для этого существует функция *MsgDeliverEvent()*, о которой речь несколько позже.

## **Идентификаторы отправителя, каналы и другие параметры**

Мы с вами пока не обсуждали различные параметры, используемые в ранее рассмотренных примерах, чтобы можно было сконцентрировать внимание на самих принципах обмена сообщениями. Теперь поговорим об этих параметрах более подробно.

### *Дополнительно о каналах*

В приведенном выше примере с сервером мы видели, что сервер создал один-единственный канал. Конечно, можно было создать больше, но обычно серверы так не делают. (Наиболее очевидный пример сервера с двумя каналами — это администратор штатной сети **qnet** — вот уж определенно эксцентричный образец программного обеспечения !)

Оказывается, что в действительности нет большой необходимости в создании нескольких каналов. Главное назначение канала состоит в том, чтобы четко указать серверу, где «слушать» на предмет входящих сообщений, и четко указать клиентам, куда передавать сообщения (через

соответствующие соединения). Единственная ситуация, когда вам могло бы понадобиться использовать несколько каналов в сервере, — это если бы хотели реализовать сервер, предоставляющий различные услуги (или различные классы услуг) в зависимости от того, по какому каналу было принято сообщение. Второй канал мог бы применяться, например, для отправки сообщений типа «импульс», пробуждающих субсерверы — это гарантировало бы развязку этого сервиса от служебных функций, предоставляемых обычными сообщениями по первому каналу.

В предыдущем параграфе я утверждал, что вы могли бы использовать в сервере пул потоков, готовый принимать сообщения от клиентов, и что реально не имеет значения, который именно из потоков в пуле получит запрос. Это еще один аспект «канальной абстракции». В предыдущих версиях QNX (особенно в QNX4), клиент мог передать сообщение серверу, определяя его идентификатором узла (node ID) и идентификатором процесса (process ID) на этом узле. Поскольку QNX4 — однопоточная ОС, никакого беспорядка с тем, кому передается сообщение, в ней быть не могло. Однако, стоит ввести понятие потока, и встает дополнительная проблема адресации потоков в процессе (ведь именно потоки собственно предоставляют сервисы). Поскольку поток — вещь преходящая, в действительности для клиента не имеет смысла подключаться к четко определенному потоку в четко определенном процессе на четко определенном узле. К тому же, а что если нужный поток занят? Мы тогда должны были бы обеспечить клиенту возможность выбрать «незанятый поток из некоторого пула потоков, предоставляющих нужный сервис».

Так вот, для этого и существуют каналы. Канал — это «адрес» некоторого «пула потоков, предоставляющих нужный сервис». Суть здесь заключается в том, что вызвать функцию *MsgReceive()* по одному и тому же каналу могут несколько потоков одновременно. Все они будут блокированы, но входящее сообщение будет передано только одному из них.

### ***Кто послал сообщение?***

Довольно часто серверу необходимо знать, кто послал ему сообщение. Для этого есть ряд причин, например:

- учет клиентов;
- управление доступом;
- определение контекстных связей;
- выбор типа сервиса;

- и т.д.

Сделать так, чтобы клиент передавал серверу эту информацию с каждым сообщением, было бы излишне громоздким (да и давало бы лишние лазейки в системе защиты). Поэтому существует специальная структура, заполняемая ядром всякий раз, когда функция *MsgReceive()* разблокируется, приняв сообщение. Эта структура имеет тип **struct \_msg\_info** и содержит в себе следующее:

```
struct _msg_info {  
    int nd;  
    int srcnd;  
    pid_t pid;  
    int32_t chid;  
    int32_t scoid;  
    int32_t coid;  
    int32_t msglen;  
    int32_t tid;  
    int16_t priority;  
    int16_t flags;  
    int32_t srcmsglen;  
};
```

Вы передаете все это функции *MsgReceive()* в качестве последнего параметра. Если вы передаете NULL, то не произойдет ничего. (Информацию все равно можно будет потом получить с помощью вызова функции *MsgInfo()* — она не теряется!)

Давайте взглянем на поля этой структуры:

Это дескриптор узла, идентификатор процесса и идентификатор потока клиента. (Заметьте, что *nd* — это дескриптор *nd*, *srcnd*, принимающего узла для режима передачи, а *srcnd* — это *pid* и *tid* дескриптор передающего узла для режима приема. Для этого имеется очень серьезное основание ;-), которое мы рассмотрим ниже в разделе «Несколько замечаний о дескрипторах узлов»).

*priority* Приоритет потока, пославшего сообщение.

*chid*, *coid* Идентификатор канала, по которому сообщение было передано, и идентификатор использованного при этом соединения.

*scoid* Идентификатор соединения с сервером. Это внутренний идентификатор, который применяется ядром для маршрутизации сообщения от сервера назад к клиенту. Вам не нужно ничего знать об этом идентификаторе, кроме одного любопытного факта,

что это будет небольшое целое число, которое уникально идентифицирует клиента.

*flags* Содержит различные битовые флаги: `_NTO_MI_ENDIAN_BIG`, `_NTO_MI_ENDIAN_DIFF`, `_NTO_MI_NET_CRED_DIRTY` и `_NTO_MI_UNBLOCK_REQ`. Биты `_NTO_MI_ENDIAN_BIG` и `_NTO_MI_ENDIAN_DIFF` сообщают вам о порядке байт в слове для отправившей сообщение машины (в случае, если сообщение пришло через сеть от машины с другим порядком байт), бит `_NTO_MI_NET_CRED_DIRTY` зарезервирован для внутреннего использования, значение бита `_NTO_MI_UNBLOCK_REQ` мы рассмотрим в разделе «Использование бита `_NTO_MI_UNBLOCK_REQ`», см. ниже.

*msglen* Число принятых байт.

Длина исходного сообщения в байтах, как оно было отправлено клиентом. Это число может превышать значение *msglen* — например, в случае приема меньшего количества данных, чем *srcmsglen* было послано. Заметьте, что это поле действительно только в том случае, если установлен бит `_NTO_CHF_SENDER_LEN` в переданном функции *ChannelCreate()* (для канала, по которому было получено данное сообщение) параметре *flags*.

<b><i>Идентификатор отправителя (receive ID), он же клиентский жетон (client cookie)</i></b>
--

В примере программы, представленном выше, отметьте следующее:

```
rcvid = MsgReceive(...);  
...  
MsgReply(rcvid, ...);
```

Это — ключевой фрагмент, потому что именно в нем иллюстрируется привязка приема сообщения от клиента к последующему ответу этому конкретному клиенту. Идентификатор отправителя — это целое число, которое действует как жетон («magic cookie»), который вы получаете от клиента и обязаны хранить, если вы желаете впоследствии взаимодействовать с этим клиентом. Что произойдет, если вы его потеряете? Его больше нет. Функция *MsgSend()* клиента не разблокируется, пока вы (конкретный сервер) живы, или пока не произошел тайм-аут обмена сообщениями (и даже в этом случае все не так просто; см. функцию *TimerTimeout()* в справочном руководстве по библиотеке Си и обсуждение о



применения в главе «Часы, таймеры и периодические уведомления», раздел «Тайм-ауты ядра»).

☞ Не пытайтесь извлечь из значения идентификатора отправителя какой-либо конкретный смысл — он может измениться в будущих версиях операционной системы. Единственное, что нужно знать — что он уникален, то есть у вас никогда не будет двух различных клиентов с одним и тем же идентификатором отправителя (иначе ядро просто не сможет их различить, когда вы вызовете *MsgReply()*).

Отметим также, что за исключением одного частного случая (с применением функции *MsgDeliverEvent()*, которую мы рассмотрим позже), после вызова функции *MsgReply()* соответствующий идентификатор отправителя перестает иметь смысл.

Таким образом, мы плавно переходим к функции *MsgReply()*.

### ***Ответ клиенту***

Функция *MsgReply()* принимает в качестве параметров идентификатор отправителя, код возврата, указатель на сообщение и размер этого сообщения. Мы только что обсудили идентификатор отправителя — он уникально идентифицирует того, кому должно быть отправлено ответное сообщение. Код возврата указывает, какой код должна вернуть функция *MsgSend()* клиента. Наконец, указатель на сообщение и размер указывают на местоположение и размер (необязательного!) ответного сообщения, которое следует отправить.

Функция *MsgReply()* может показаться очень простой (и так оно и есть), но рассмотреть ее применение было бы полезно.

### ***А можно и не отвечать***

Однако, вы вовсе не обязаны обязательно ответить клиенту перед приемом новых сообщений от других клиентов с помощью функции *MsgReceive()*! Это положение можно с успехом использовать в множестве различных сценариев.

В типовом драйвере устройства клиент может выдать запросом, который не будет обслужен в течение продолжительного времени. Например, клиент может запросить драйвер аналого-цифрового преобразователя (АЦП): «Сходи-ка принеси мне данные за следующие 45 секунд.» Драйвер АЦП не может себе позволить вывесить табличку «Закрыто» на целых 45 секунд, потому что другим клиентам тоже может срочно что-нибудь понадобиться — например, данные по другому каналу, информация о состоянии, и т.п.

В соответствии со своей архитектурой, драйвер АЦП просто поставит в очередь полученный от функции *MsgReceive()* идентификатор отправителя, осуществит запуск 45-секундного процесса накопления данных и снова вернется к обработке клиентских запросов. По истечении этого 45-секундного интервала, когда данные накоплены, драйвер АЦП сможет найти идентификатор отправителя, связанный с данным запросом, и ответить нужному клиенту.

Вам также может понадобиться задержаться с ответом клиенту в случае модели «сервер/субсервер» (то есть некоторые клиенты — на самом деле субсерверы). Вы можете просто запомнить идентификаторы ищущих работу субсерверов и сохранить их до поры до времени. Когда работа для субсерверов появится, тогда и только тогда вы ответите субсерверу, указав, что именно он должен сделать.

### ***Ответ без данных или с кодом ошибки (errno)***

Когда дело наконец доходит до ответа клиенту, вы совершенно не обязаны передавать ему какие-либо данные. Это может использоваться в двух случаях.

Вы можете отправить клиенту ответ без данных, если единственная цель ответа — разблокировать клиента. Скажем, клиент желает быть заблокированным до некоторого события, а до какого именно — ему знать не обязательно. В этом случае функции *MsgReply()* не потребуется никаких данных, достаточно будет только идентификатора отправителя:

```
MsgReply(rcvid, EOK, NULL, 0);
```

Такой вызов разблокирует клиента (но не передаст ему никаких данных) и возвратит код ЕОК («успешное завершение»).

Как вариант, вы можете при желании вернуть клиенту код ошибки. Вы не сможете сделать это с помощью функции *MsgReply()*, вместо нее для этого используется функция *MsgError()*:

```
MsgError(rcvid, EROFS);
```

В приведенном выше примере сервер обнаруживает, что клиент пытается записать данные в файловую систему, предназначенную только для чтения, и вместо данных возвращает клиенту код ошибки (*errno*) EROFS.

Еще одним поводом ответить клиенту без данных (и соответствующие вызовы мы вскоре рассмотрим) может быть то, что данные уже переданы ранее (с помощью функции *MsgWrite()*), и больше никаких данных нет.

Почему применяются два типа вызовов? Они немного различны. В то время как обе функции *MsgError()* и *MsgReply()* разблокируют клиента, функция *MsgError()* при этом не передаст никаких данных, заставит функцию *MsgSend()* клиента вернуть -1 и установит переменную *errno* на стороне клиента в значение, переданное функции *MsgError()* в качестве второго аргумента.

С другой стороны, функция *MsgReply()* может передавать данные (как видно из ее третьего и четвертого параметров) и заставляет функцию *MsgSend()* клиента вернуть значение, переданное *MsgReply()* в качестве второго аргумента. Переменная *errno* клиента остается нетронутой.

В общем случае, если вам нужно только сообщить о результатах действия («прошло/не прошло»), лучше применять функцию *MsgError()*. Если бы вы возвращали данные, здесь была бы необходима функция *MsgReply()*. Обычно, когда вы возвращаете данные, вторым параметром функции *MsgReply()* будет положительное целое число, указывающее на число возвращаемых байт.

**Определение идентификаторов узла, процесса и канала (ND/PID/CHID) нужного сервера**

Ранее мы отметили, что для соединения с сервером функции *ConnectAttach()* необходимо указать дескриптор узла (Node Descriptor — ND), идентификатор процесса (process ID — PID), а также идентификатор канала (Channel ID — CHID). До настоящего момента мы не обсуждали, как именно клиент находит эту информацию.

Если один процесс создает другой процесс, тогда это просто — вызов создания процесса возвращает идентификатор вновь созданного процесса. Создающий процесс может либо передать собственные PID и CHID вновь созданному процессу в командной строке, либо вновь созданный процесс может вызвать функцию *getppid()* для получения идентификатора родительского процесса, и использовать некоторый «известный» идентификатор канала.

А что если у нас два совершенно чужих процесса? Это возможно, например, в том случае, если сервер создан некоей третьей стороной, а вашему приложению нужно уметь общаться с этим сервером. Реально мы должны найти ответ на вопрос: «Как сервер объявляет о своем местонахождении?»

Существует множество способов сделать это; мы рассмотрим только три из них, в порядке возрастания «элегантности»:

1. Открыть файла с известным именем и сохранить в нем ND/PID/CHID. Такой метод является традиционным для серверов UNIX, когда сервер открывает файл (например, `/etc/httpd.pid`), записывает туда свой идентификатор процесса в виде строки ASCII и предполагают, что клиенты откроют этот файл прочитают из него идентификатор.

2. Использовать для объявления идентификаторов ND/PID/CHID глобальные переменные. Такой способ обычно применяется в многопоточных серверах, которые могут посылать сообщение сами себе. Этот вариант по самой своей природе является очень редким.

3. Занять часть пространства имен путей и стать администратором ресурсов. Мы поговорим об этом в главе «Администраторы ресурсов».

Первый подход относительно прост, но он чреват «загрязнением файловой системы», когда в каталоге `/etc` лежит куча файлов `*.pid`. Поскольку файлы устойчивы (имеется в виду, что они выживают после смерти создающего их процесса и перезагрузки машины), очевидного способа стереть эти файлы не существует — разве что использовать такую программную «старуху с косой», постоянно проверяющую, не пора ли прибрать кого-то из них.

Имеется и другая связанная с этим подходом проблема. Поскольку процесс, который создал файл, может умереть, не удалив этот файл, то вы не сможете узнать, жив ли еще этот процесс, пока не попытаете передать ему сообщение. И это ещё не самое страшное — еще хуже, если комбинация ND/PID/CHID указанная в файле, оказывается настолько старой, что может быть повторно использована другой программой! Получив «чужое» сообщение, эта программа в лучшем случае его проигнорирует его, а ведь может и предпринять некорректные действия. Так что такой подход исключается.

Второй подход, где мы используем глобальные переменные для объявления значений ND/PID/CHID, не является общим решением проблемы, поскольку в нем предполагается способность клиента обратиться к этим глобальным переменным. А поскольку для этого требуется использование разделяемой памяти, это не будет работать в сети! Так что этот метод обычно используется либо в небольших тестовых

программах, либо в очень специфичных случаях, но всегда в контексте многопоточной программы.

Что реально происходит, так это то, что один поток в программе является клиентом, а другой поток — сервером. Поток-сервер создает канал и затем размещает идентификатор канала в глобальной переменной (идентификаторы узла и процесса являются одинаковыми для всех потоков в процессе, так что объявлять их не обязательно). Поток-клиент затем берет этот идентификатор канала и выполняет по нему функцию *ConnectAttach()*.

Третий подход — сделать сервер администратором ресурса — является определенно самым прозрачным и поэтому рекомендуемым общим решением. Механизм того, как это делается, изложен в главе «Администраторы ресурсов», а пока все, что вы должны об этом знать — это то, что сервер регистрирует некое имя пути как свою «область ответственности», а клиенты обращаются к нему обычным вызовом функции *open()*.

☞ Не сочту лишним подчеркнуть:

Файловые дескрипторы POSIX в QNX/Neutrino реализованы через идентификаторы соединений, то есть дескриптор файла уже является идентификатором соединения! Органичность этой схемы в том, что поскольку дескриптор файла, возвращаемый функцией *open()*, фактически является идентификатором соединения, клиенту не нужно выполнять какие-либо дополнительные действия, чтобы использовать это соединение. Например, когда клиент после вызова *open()* вызывает функцию *read()*, передавая ей полученный дескриптор, это с минимальными накладными расходами транслируется в функцию *MsgSend()*.

### ***А что насчет приоритетов?***

А что произойдет, если сообщение серверу передадут одновременно два процесса с разными приоритетами?

☞ Сообщения всегда доставляются в порядке приоритетов.

Если два процесса посылают сообщения «одновременно», первым доставляется сообщение от процесса с высшим приоритетом.

Если оба процесса имеют одинаковый приоритет, то сообщения будут доставлены в порядке отправки (поскольку в машине с одним процессором не бывает ничего одновременного, и даже в SMP-блоке будет присутствовать некий порядок, поскольку процессоры будут конкурировать между собой за доступ к ядру).

Мы еще вернемся к анализу других тонкостей этой проблемы чуть позже в этой главе, когда будем говорить о проблеме инверсии приоритетов.

### *Чтение и запись данных*

До настоящего времени мы обсуждали основные примитивы обмена сообщениями. Как я и упоминал ранее, это минимум, который необходимо знать. Однако существует еще несколько дополнительных функций, которые делают нашу жизнь значительно проще.

Рассмотрим пример, в котором для обеспечения обмена сообщениями между клиентом и сервером нам понадобились бы и другие функции.

Клиент вызывает *MsgSend()* для передачи неких данных серверу. После вызова *MsgSend()* клиент блокируется. Теперь он ждет, чтобы сервер ему ответил.

Интересные события разворачиваются на стороне сервера. Сервер вызывает функцию *MsgReceive()* для приема сообщения от клиента. В зависимости от того, как вы спроектировали вашу систему сообщений, сервер может знать, а может и не знать, насколько велико сообщение клиента. Как сервер может не знать, каков реальный размер сообщения? Возьмем наш пример с файловой системой. Предположим, что клиент делает так:

```
write(fd, buf, 16);
```

Это сработает так, как и ожидается, если сервер вызовет *MsgReceive()* с размером буфера, скажем, 1024 байта. Так как наш клиент послал небольшое сообщение (28 байт), никаких проблем не будет.

А что если клиент отправит сообщение, превышающее по размеру 1024 байт — скажем, 1 мегабайт? Например, так:

```
write(fd, buf, 1000000);
```

Как сервер мог бы обработать это сообщение поизыщнее? Мы могли, к примеру, сказать, что клиенту не позволено записывать более чем *n* байт. Тогда функции *write()* в клиентской Си-библиотеке пришлось бы разбивать

каждый «длинный» запрос на несколько запросов по  $n$  байт каждый. Неуклюже.

Другая проблема в этом примере заключается в вопросе «А каково должно быть  $n$ ?»

Как вы видите, этот подход имеет следующие основные недостатки:

- Все функции, которые применяются для обмена сообщениями ограниченного размера, должны быть модифицированы в Си-библиотеке так, чтобы функция передавала запросы в виде серии пакетов. Это само по себе немалый объем работы. Также это может иметь ряд неожиданных побочных эффектов при работе в многопоточной среде — что если первая часть сообщения от одного потока передана, и тут его вытесняет другой поток клиента и посылает свое собственное сообщение. Что будет с прерванным потоком тогда?

- Все серверы должны быть готовы к обработке сообщения максимально возможного размера. Это означает, что все серверы должны будут иметь значительные области данных, или Си-библиотека будет должна разделять большие запросы на несколько меньших, ухудшая тем самым быстродействие.

К счастью, эта проблема довольно просто обходится, причем даже с дополнительным выигрышем.

Здесь будут особенно полезны функции *MsgRead()* и *MsgWrite()*. Важно при этом помнить, что клиент заблокирован — это означает, что он не собирается изменять данные, пока сервер их анализирует.

☞ В многопоточном клиенте теоретически возможно, что в область данных заблокированного по серверу клиентского потока залезет другой поток. Такая ситуация рассматривается как некорректная (ошибка проектирования), поскольку серверный поток предполагает, что он имеет монополярный доступ к области данных клиента, пока тот заблокирован.

Функция *MsgRead()* описана так:

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

```
int MsgRead(int rcvid, void *msg, int nbytes, int offset);
```

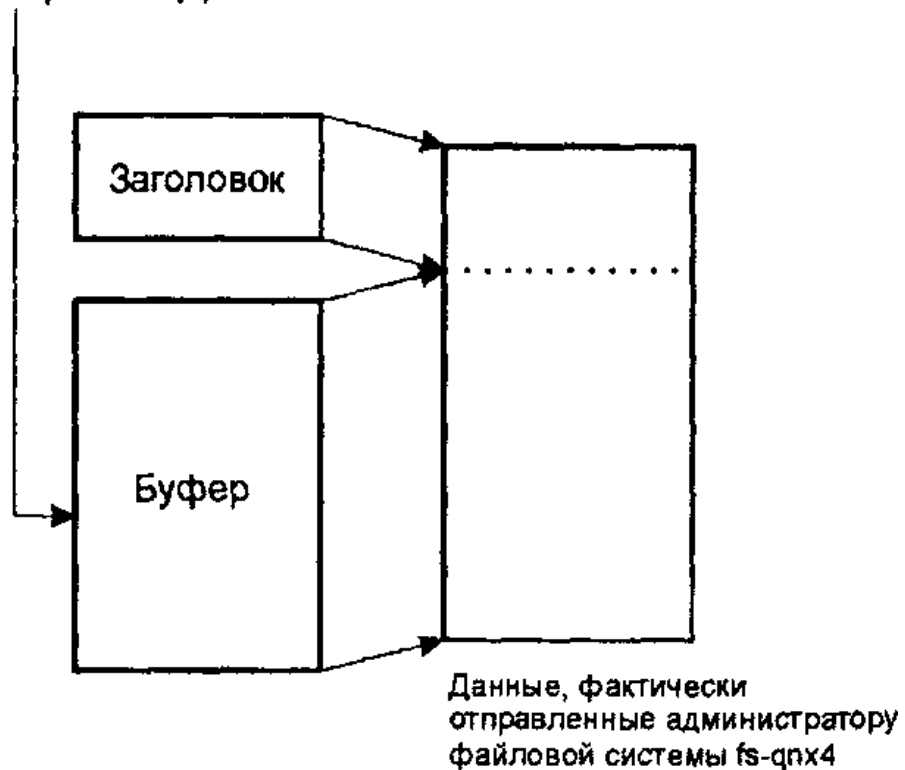
Функция *MsgRead()* позволяет Вашему серверу считать *nbytes* байт данных из адресного пространства заблокированного клиента, начиная со смещения *offset* от начала клиентского буфера, в буфер, указанный параметром *msg*. Сервер не блокируется, а клиент не разблокируется.

Функция *MsgRead()* возвращает число байтов, которые были фактически считаны, или возвращает -1, если произошла ошибка.

Итак, давайте подумаем, как бы мы использовали эти возможности в нашем примере с вызовом *write()*. Библиотечная функция *write()* создает сообщение с заголовком и посылает его серверу файловой системы **fs-qnx4**. Сервер принимает небольшую часть сообщения с помощью *MsgReceive()*, анализирует его и принимает решение, где разместить остальную часть сообщения — например, где-то в уже выделенном буфере дискового кэша.

Давайте рассмотрим пример.

```
write (fd, buf, 4096);
```



Пример отправки сообщения серверу **fs-qnx4** с непрерывным представлением данных.

Итак, клиент решил переслать файловой системе 4Кб данных. (Отметьте для себя, что Си-библиотека добавила к сообщению перед данными небольшой заголовок — чтобы потом можно было узнать, к какому типу принадлежал этот запрос. Мы еще вернемся к этому вопросу, когда будем говорить о составных сообщениях, а также — еще более детально — когда будем анализировать работу администраторов ресурсов.) Файловая система считывает только те данные (заголовок), которые будут ей необходимы для того, чтобы выяснить тип принятого сообщения:

```
// Часть заголовков, вымышлены для примера
```



```

struct _io_write {
    uint16_t type;
    uint16_t combine_len;
    int32_t nbytes;
    uint32_t xtype;
};

typedef union {
    uint16_t type;
    struct _io_read io_read;
    struct _io_write io_write;
    ...
} header_t;

header_t header; // Объявить заголовок

rcvid = MsgReceive(chid, &header, sizeof(header), NULL);
switch (header.type) {
    ...
case _IO_WRITE:
    number_of_bytes = header.io_write.nbytes;
    ...

```

Теперь сервер **fs-qnx4** знает, что в адресном пространстве клиента находится 4Кб данных (сообщение известило его об этом через элемент структуры *nbytes*), и что эти данные надо передать в буфер кэша. Теперь сервер **fs-qnx4** может сделать так:

```

MsgRead(rcvid, cache_buffer[index].data,
        cache_buffer[index].size, sizeof(header.io_write));

```

Обратите внимание, что операции приема сообщения задано смещение **sizeof(header.io\_write)** — это сделано для того, чтобы пропустить заголовок, добавленный клиентской библиотекой. Мы предполагаем здесь, что **cache\_buffer[index].size** (размер буфера кэша) равен 4096 (или более) байт.

Для записи данных в адресное пространство клиента есть аналогичная функция:

```

#include <sys/neutrino.h>

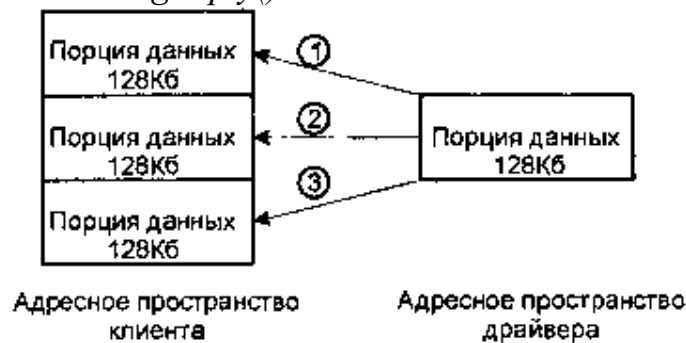
int MsgWrite(int rcvid, const void *msg, int nbytes,
             int offset);

```

Применение функции *MsgWrite()* позволяет серверу записать данные в адресное пространство клиента, начиная со смещения *offset* байт от начала указанного клиентом приемного буфера. Эта функция наиболее полезна в случаях, где сервер ограничен в ресурсах, а клиент желает получить от него значительное количество информации.

Например, в системе сбора данных клиент может выделить 4-мегабайтный буфер и приказать драйверу собрать 4 мегабайта данных. Драйверу вовсе не обязательно держать под боком здоровенный буфер просто так, на случай если кто-то вдруг неожиданно запросит передачу большого массива данных.

Драйвер может иметь буфер размером 128Кб для обмена с аппаратурой посредством DMA, а сообщение пересылать в адресное пространство клиента по частям, используя функцию *MsgWrite()* (разумеется, каждый раз увеличивая смещение на 128Кб). Когда будет передан последний фрагмент, можно будет вызывать *MsgReply()*.



Передача нескольких фрагментов сообщения с помощью функции *MsgWrite()*

Отметим, что функция *MsgWrite()* позволяет вам записать различные компоненты данных в различные места, а затем либо просто разбудить клиента вызовом *MsgReply()*:

```
MsgReply(rcvid, EOK, NULL, 0);
```

либо сделать это после записи заголовка в начало клиентского буфера:

```
MsgReply(rcvid, EOK, &header, sizeof(header));
```

Это довольно изящный трюк для записи неизвестного количества данных, когда вы узнаете, сколько данных нужно было записать, только когда запись уже закончена. Главное — если вы будете использовать второй метод, с записью заголовка после записи данных, не забудьте зарезервировать место под заголовок в начале клиентского буфера!

## Составные сообщения

До сих пор мы демонстрировали только обмен сообщениями, когда данные передаются из одного буфера в адресном пространстве клиента в другой буфер в адресном пространстве сервера (и наоборот — в случае ответа на сообщение).

При том, что данный подход вполне приемлем для большинства приложений, его применение далеко не всегда эффективно. Вспомните: наша функция *write()* из Си-библиотеки берет переданный ей буфер и добавляет в его начало небольшой заголовок. Используя то, что мы уже изучили ранее, вы могли бы ожидать, что реализация *write()* в Си-библиотеке может выглядеть примерно так (это не реальный код!):

```
ssize_t write(int fd, const void *buf, size_t nbytes) {
    char *newbuf;
    io_write_t *wptr;
    int nwritten;
    newbuf = malloc(nbytes + sizeof(io_write_t));
    // Заполнить write_header
    wptr = (io_write_t*)newbuf;
    wptr->type = _IO_WRITE;
    wptr->nbytes = nbytes;
    // Сохранить данные от клиента
    memcpy(newbuf + sizeof(io_write_t), buf, nbytes);
    // Отправить сообщение серверу
    nwritten =
        MsgSend(fd, newbuf, nbytes + sizeof(io_write_t),
                newbuf, sizeof(io_write_t));
    free(newbuf);
    return(nwritten);
}
```

Понимаете, что произошло? Несколько неприятных вещей:

- Функция *write()* теперь должна быть способна выделить память под буфер достаточно большого размера как для данных клиента (которые могут быть довольно значительными по объему), так и для заголовка. Размер заголовка не имеет значения — в этом случае он был равен 12 байтам.

- Мы были должны скопировать данные дважды: в первый раз — при использовании функции *memcpy()*, и затем еще раз, снова — уже при осуществлении передачи сообщения.

- Мы должны были предусмотреть указатель на тип *io\_write\_t* и установить его на начало буфера, вместо использования обычных механизмов доступа (впрочем, это незначительный недостаток).

Поскольку ядро намерено копировать данные в любом случае, было бы хорошо, если бы мы смогли сообщить ему о том, что одна часть данных (заголовок) фиксирована по некоторому адресу, а другая часть (собственно данные) фиксирована где-нибудь еще, без необходимости самим вручную собирать буферы из частей и копировать данные.

На наше счастье, в QNX/Neutrino реализован механизм, который позволяет нам сделать именно так! Механизм этот называется IOV (i/o vector), или «вектор ввода/вывода».

Давайте для начала рассмотрим некоторую программу, а затем обсудим, что происходит с применением такого вектора.

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

ssize_t write(int fd, const void *buf, size_t nbytes) {
    io_write_t whdr;
    iov_t iov[2];
    // Установить IOV на обе части:
    SETIOV(iov + 0, &whdr, sizeof(whdr));
    SETIOV(iov + 1, buf, nbytes);
    // Заполнить io_write_t
    whdr.type = _IO_WRITE;
    whdr.nbytes = nbytes;
    // Отправить сообщение серверу
    return (MsgSendv(coid, iov, 2, iov, 1));
}
```

Прежде всего, обратите внимание на то, что не применяется никакой функции *malloc()* и никакой функции *memset()*. Затем обратим внимание на тип применяемого вектора IOV — *iov\_t*. Это структура, которая содержит два элемента — адрес и длину. Мы определили массив из двух таких структур и назвали его *iov*.

Определение типа вектора *iov\_t* содержится в *<sys/neutrino.h>* и выглядит так:

```
typedef struct iovec {
    void *iov_base;
    size_t iov_len;
} iov_t;
```

Мы заполняем в этой структуре пары «адрес — длина» для заголовка операции записи (первая часть) и для данных клиента (вторая часть). Существует удобная макрокоманда, *SETIOV()*, которая выполняет за нас необходимые присвоения. Она формально определена следующим образом:

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

```
#define SETIOV(_iov, _addr, _len) \
    ((_iov)->iov_base = (void *) (_addr), \
     (_iov)->iov_len = (_len))
```

Макрос *SETIOV()* принимает вектор *iov\_t*, а также адрес и данные о длине, которые подлежат записи в вектор IOV.

Также отметим, что как только мы создаем IOV для указания на заголовок, мы сможем выделить стек для заголовка без использования *malloc()*. Это может быть и хорошо, и плохо — это хорошо, когда заголовок невелик, потому что вы хотите исключить головные боли, связанные с динамическим распределением памяти, но это может быть плохо, когда заголовок очень велик, потому что тогда он займет слишком много стекового пространства. Впрочем, заголовки обычно невелики.

В любом случае, вся важная работа выполняется функцией *MsgSendv()*, которая принимает почти те же самые аргументы, что и функция *MsgSend()*, которую мы использовали в предыдущем примере:

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

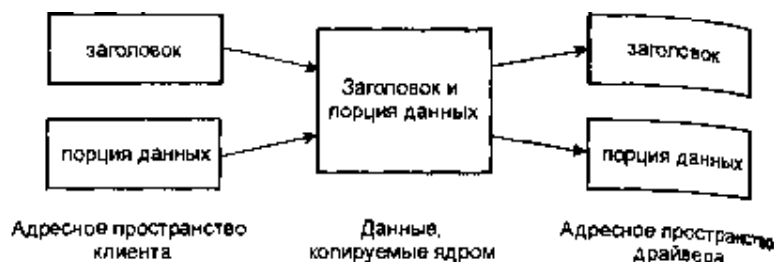
```
int MsgSendv(int coid, const iov_t *siov, int sparts,
             const iov_t *riov, int rparts);
```

Давайте посмотрим на ее аргументы:

*coid* Идентификатор соединения, по которому мы передаем — как и при использовании функции *MsgSend()*.

Число пересылаемых и принимаемых частей, указанных *sparts* параметрами вектора *iov\_t*; в нашем примере мы присваиваем и аргументу *sparts* значение 2, указывая этим, что пересылаем *rparts* сообщение из двух частей, а аргументу *rparts* — значение 1, указывая этим, что мы принимаем ответ из одной части.

Эти массивы значений типа *iov\_t* указывают на пары «адрес — длина», которые мы желаем переслать. В вышеупомянутом примере *siov* и *riov* мы выделяем *siov* из двух частей, указывая ими на заголовок и данные клиента, и *riov* из одной части, указывая им только на заголовок.



Как ядро видит составное сообщение.

Ядро просто прозрачно копирует данные из каждой части вектора IOV из адресного пространства клиента в адресное пространство сервера (и обратно, при ответе на сообщение). Фактически, при этом ядро выполняет операцию фрагментации/дефрагментации сообщения (scatter/gather).

Несколько моментов, которые необходимо запомнить:

- Число фрагментов ограничено значением  $2^{31}$  (больше, чем вам придется использовать!); число 2 в нашем примере — типовое значение.

- Ядро просто копирует данные, указанные вектором IOV, из одного адресного пространства в другое.

- *Вектор-источник и вектор-приемник не должны совпадать.*

Почему последний пункт так важен? Для того чтобы ответить, рассмотрим все подробнее. Со стороны клиента, скажем, мы выдали:

```
write(fd, buf, 12000);
```

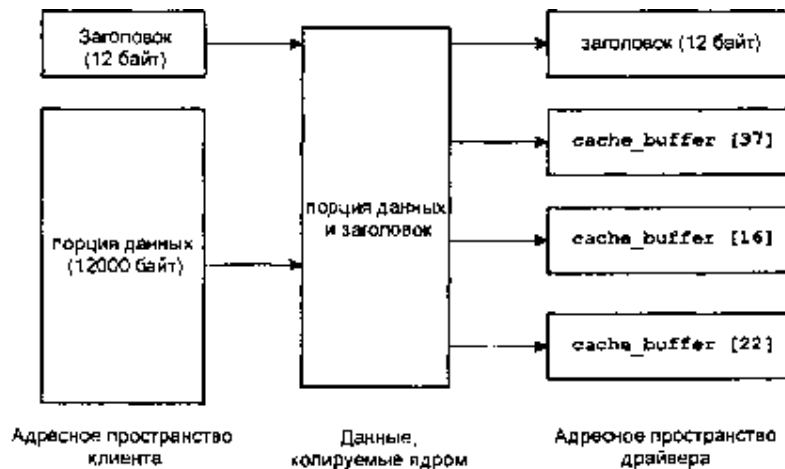
в результате чего был создан вектор IOV из двух частей:

- заголовок (12 байт);
- данные (12000 байт);

На стороне сервера (скажем, сервера файловой системы **fs-qnx4**) мы имеем блоки памяти кэша до 4Кб каждый, и мы хотели бы эффективно принять сообщение непосредственно в эти блоки. В идеале мы бы написали что-то типа:

```
// Настроить структуру IOV для приема:
SETIOV(iov + 0, &header, sizeof(header.io_write));
SETIOV(iov + 1, &cache_buffer[37], 4096);
SETIOV(iov + 2, &cache_buffer[16], 4096);
SETIOV(iov + 3, &cache_buffer[22], 4096);
rcvid = MsgReceivev(chid, iov, 4, NULL);
```

Эта программа делает в значительной степени то, что вы и предполагаете: она задает вектор IOV из 4 частей, первая из которых указывает на заголовок, а следующие три части — на блоки кэш-памяти с номерами 37, 16 и 22. (Предположим, что именно эти блоки случайно оказались доступными в данный момент.) Ниже это иллюстрируется графически.



Распределение непрерывных данных по отдельным буферам.

Затем осуществляется вызов функции *MsgReceivev()*, и ей указывается, что мы намерены принять сообщение по указанному каналу (параметр *chid*), и что вектор IOV для этой операции состоит из 4 частей.

(Кроме возможности работать с векторами IOV, функция *MsgReceivev()* действует аналогично функции *MsgReceive()*.)

Опа! Мы сделали ту же самую ошибку, которую уже делали к раньше, когда познакомились с функцией *MsgReceive()*. Как мы узнаем, сообщение какого типа мы собираемся принять и сколько в нем данных, пока не примем все сообщение целиком?

Мы сможем решить эту проблему тем же способом, что и прежде:

```
rcvid = MsgReceive(chid, &header, sizeof(header), NULL);
switch (header.message_type) {
    ...
case _IO_WRITE:
    number_of_bytes = header.io_write.nbytes;
    // Выделить / найти элемент кэша
    // Заполнить элементами кэша 3-элементный IOV
    MsgReadv(rcvid, iov, 3, sizeof(header.io_write));
```

Здесь мы вызываем «предварительную» *MsgReceive()* (отметьте, что тут мы не используем ее векторную форму, поскольку для сообщения, состоящего из одной части, в ней просто нет необходимости), определяем тип сообщения и затем продолжаем считывать данные из адресного пространства клиента (начиная со смещения `sizeof(header.io_write)`) в кэш-буферы, определенные трехэлементным вектором IOV.

Обратите внимание, что мы перешли от вектора IOV, состоящего из 4 частей (как в первом примере), к вектору IOV из 3 частей. Дело в том, что в первом примере первый из четырех элементов вектора IOV отводился под заголовок, который на этот раз мы считали непосредственно при помощи

функции *MsgReceive()*, а последние три элемента аналогичны трехэлементному вектору из второго примера — они определяют место, куда мы хотим записать данные.

Можно представить, как мы ответили бы на запрос чтения:

1. Найти элементы кэша, которые соответствуют запрашиваемым данным.
2. Заполнить вектора IOV ссылками на них.
3. Применить функцию *MsgWritev()* (или *MsgReplyv()*) для передачи данных клиенту.

Отметим, что если данные начинаются не непосредственно с начала блока кэша (или другой структуры данных), то в этом нет никакой проблемы. Просто сместите первый вектор IOV на точку начала данных и соответственно откорректируйте поле размера.

### *Как насчет других версий?*

Все функции обмена сообщениями, кроме функций семейства *MsgSend\*()*, имеют одинаковую общую форму: если имя функции имеет суффикс «v», значит, она принимает в качестве аргументов вектор IOV и число его частей; в противном случае, она принимает указатель и длину.

Семейство *MsgSend\*()* содержит четыре основных варианта реализации функций с точки зрения буферов источника и адресата, плюс два варианта собственно системного вызова — итого восемь.

В нижеприведенной таблице сведены данные о вариантах функций семейства *MsgSend\*()*.

Функция	Буфер передачи	Буфер приема
<i>MsgSend()</i>	линейный	линейный
<i>MsgSendnc()</i>	линейный	линейный
<i>MsgSendsv()</i>	линейный	IOV
<i>MsgSendsvnc()</i>	линейный	IOV
<i>MsgSendvs()</i>	IOV	линейный
<i>MsgSendvsnc()</i>	IOV	линейный
<i>MsgSendv()</i>	IOV	IOV
<i>MsgSendvnc()</i>	IOV	IOV

Под линейным буфером я подразумеваю, что передается единый буфер типа `void*` вместе с его длиной. Это легко запомнить: суффикс «v» означает «вектор», и он находится на том же самом месте, что и соответствующий



параметр — первым или вторым, в зависимости от того, какой буфер — передачи или приема — объявляется векторным.

Хмм. Получается, что функции *MsgSendsv()* и *MsgSendsvnc()* идентичны? Да, по части параметров именно так оно и есть. Различие заключается в том, является функция точкой завершения (cancellation point) или нет. Версии с суффиксом «nc» («no cancellation» — прим. ред.) не являются точками завершения, в то время как версии без этого суффикса — являются. (Дополнительную информацию относительно точек завершения и завершаемости (cancelability) вообще можно найти в справочном руководстве по Си-библиотеке в главе, посвященной *pthread\_cancel()*.)

### Реализация

Вероятно, вы уже подозревали, что все варианты функций *MsgRead()*, *MsgReceive()*, *MsgSend()* и функций *MsgWrite()* тесно связаны между собой. (Единственное исключение — функция *MsgReceivePulse()*; мы ее вкратце рассмотрим.)

Какие из этих функций следует применять? В общем-то вопрос этот является чисто философским. Что до меня лично, то я предпочитаю комбинировать.

Если мы посылаем или принимаем только одноэлементные сообщения, то зачем нам все эти проблемы с настройкой векторов IOV?

Накладные расходы (кстати, незначительные) по загрузке процессора обычно не зависят от того, настраиваете ли вы все сами или оставляете это ядру или библиотеке. Подход с использованием одноэлементных сообщений избавляет ядро от необходимости манипуляций с адресным пространством и поэтому работает несколько быстрее.

Следует ли вам применять функции, использующие IOV? Конечно! Используйте их всегда, когда вам приходится самостоятельно программировать обмен многоэлементными сообщениями. *Никогда* непосредственно не копируйте данные при передаче многоэлементных сообщений, даже если для этого потребуется всего несколько строк программы. Это перегрузит систему попытками минимизировать число реальных операций копирования данных туда-сюда; передача указателей происходит намного быстрее, чем копирование данных из буфера в буфер.

## Сообщения типа «импульс» (pulse)

Все сообщения, которые мы обсуждали до настоящего времени, блокируют клиента. Как только клиент вызывает функцию *MsgSend()*, для него наступает тихий час. Клиент отдыхает до тех пор, пока сервер не ответит на сообщение.

Однако есть ситуации, где отправитель сообщения не может себе позволить блокироваться. Мы рассмотрим некоторые из них в главах «Прерывания» и «Часы, таймеры и периодические уведомления», а сейчас мы должны понять концепцию данной проблемы.

Механизм, который обеспечивает отправку сообщения без блокирования, называют «импульсом» (pulse). Импульс — это миниатюрное сообщение, которое:

- может перенести 40 бит полезной информации (8-битный код и 32 бита данных);
- является неблокирующим для отправителя;
- может быть получено точно так же, как и сообщение другого типа;
- ставится в очередь, если получатель не заблокирован в ожидании сообщения.

## Прием импульса

Прием импульса выполняется очень просто: короткое, четко определенное сообщение передается функции *MsgReceive()*, как будто поток отправил обычное стандартное сообщение. Единственное различие состоит в том, что вы не сможете применить функцию *MsgReply()* к такому сообщению, поскольку, кроме всего прочего, общая идея импульса состоит в том, что это сообщение по своей сути является асинхронным. В данном разделе мы рассмотрим другую функцию, *MsgReceivePulse()*, применение которой полезно при обработке импульсов.

Единственно что забавляет при работе с импульсами — это то, что идентификатор отправителя, который возвращается функцией *MsgReceive()* при их приеме, имеет нулевое значение. Это верный индикатор того, что принятое сообщение является импульсом, а не стандартным сообщением клиента. В коде серверов вы будете часто видеть фрагменты, подобные представленному ниже:

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

```

rcvid = MsgReceive(chid, ...);
if (rcvid == 0) { // Это импульс
    // Определить тип импульса

    // Обработать его
} else { // Это обычное сообщение
    // Определить тип сообщения

    // Обработать его
}

```

### Что внутри импульса?

Итак, вы принимаете сообщение с нулевым идентификатором отправителя. Что у него внутри? Вот фрагмент заголовочного файла **<sys/neutrino.h>**:

```

struct _pulse {
    _uint16    type;
    _uint16    subtype;
    _int8      code;
    _uint8     zero[3];
    union sigval value;
    _int32     scoid;
};

```

Элементы *type* и *subtype* равны нулю (это еще один признак того, что перед нами импульс); содержимое элементов *code* и *value* определяется отправителем. В общем случае элемент *code* будет указывать на причину, по которой был отправлен импульс, а параметр *value* будет содержать 32 бита данных, ассоциируемых с данным импульсом. Эти два поля и есть те самые 40 бит контента; другие поля пользователем не настраиваются.

Ядро резервирует отрицательные значения параметра *code*, оставляя 127 значений для программистов — для использования по своему усмотрению.

Элемент *value* в действительности является элементом типа **union**:

```

union sigval {
    int sival_int;
    void *sival_ptr;
};

```

Поэтому (в развитие примера с сервером, представленного выше) вы часто будете видеть программу, подобную этой:

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

rcvid = MsgReceive(chid, ...
if (rcvid == 0) { // Импульс
    // Определить тип импульса
    switch (msg.pulse.code) {
    case MY_PULSE_TIMER:
        // Сработал один из наших таймеров,
        // надо что-то делать...
        break;
    case MY_PULSE_HWINT:
        // Импульс получен от обработчика прерывания.
        // Надо заглянуть в поле «value»...
        val = msg.pulse.value.sival_int;
        // Сделать что-нибудь по этому поводу...
        break;
    case _PULSE_CODE_UNBLOCK:
        // Это импульс от ядра, разблокирующий клиента
        // Сделать что-нибудь по этому поводу...
        break;
        //и так далее...
    }
} else { // Обычное сообщение
    // Определить тип сообщения
    // Обработать его
}
```

В этой программе предполагается, конечно, что вы описали структуру *msg* так, чтобы она содержала элемент «**struct \_pulse pulse;**», и что определены константы **MY\_PULSE\_TIMER** и **MY\_PULSE\_HWINT**. Код импульса **\_PULSE\_CODE\_UNBLOCK** — один из тех самых отрицательных кодов, зарезервированных для ядра, как это было упомянуто выше. Вы можете найти полный список этих кодов (а также краткое описание поля *value*) в **<sys/neutrino.h>**.

<b>Функция <i>MsgReceivePulse()</i></b>
---

Функции *MsgReceive()* и *MsgReceivev()* могут принимать либо стандартное сообщение, либо импульс. Однако, возможны ситуации, когда вы пожелаете принимать только импульсы. Лучшим примером этого является ситуация с сервером, когда вы приняли запрос от клиента на выполнение чего-нибудь, но не можете выполнить этот запрос сразу (возможно, из-за длительной операции, связанной с аппаратными средствами). В таких случаях следует, как правило, настроить аппаратные средства (или таймер, или что-нибудь еще) на передачу вам импульса всякий раз, когда происходит некое значительное событие.

Если вы напишете ваш сервер по стандартной схеме «ждать сообщения в бесконечном цикле», вы можете оказаться в ситуации, когда один клиент посылает вам запрос, а потом, пока вы ожидаете импульса, который должен сигнализировать об отработке запроса, приходит запрос от другого клиента. Вообще говоря, это как раз то что нужно — в конце концов, мы хотели иметь способность одновременно обслуживать множество клиентов. Однако, у вас могут быть веские основания отказать клиенту в обслуживании — например, если обслуживание клиента слишком ресурсоемко, и надо ограничить численность одновременно обрабатываемой клиентуры.

В таком случае вам потребуется обеспечить возможность «выборочного» приема только импульсов. Тут-то и становится актуальной функция *MsgReceivePulse()*:

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

```
int MsgReceivePulse(int chid, void *rmsg, int rbytes,  
    struct _msg_info *info);
```

Видно, что ее параметры те же, что и у функции *MsgReceive()* — идентификатор канала, буфер (и его размер), и параметр *info* — мы обсуждали его в параграфе «Кто послал сообщение?» Заметьте, что параметр *info* не применяется в импульсах. Вы можете спросить, почему он представлен в списке параметров. Ответ незамысловат: так было проще сделать. Просто передайте NULL!

Функция *MsgReceivePulse()* способна принимать только импульсы. Так, если бы у вас был канал с множеством потоков, заблокированных на нем с помощью функции *MsgReceivePulse()* (и ни одного потока, заблокированного на нем с помощью функции *MsgReceive()*), и некий клиент попытался бы отправить вашему серверу сообщение, то этот клиент остался бы заблокированным по передаче (Send-blocked) до тех пор, пока какой-либо поток сервера не вызовет *MsgReceive()*. Тем временем функция *MsgReceivePulse()* будет спокойно принимать импульсы.

Единственное, что можно гарантировать при совместном применении функций *MsgReceivePulse()* и *MsgReceive()*, — что функция *MsgReceivePulse()* обеспечит прием исключительно импульсов. Функция *MsgReceive()* сможет принимать как импульсы, так и обычные сообщения! Это происходит потому, что применение функции *MsgReceivePulse()* зарезервировано специально для случаев, где нужно исключить получение сервером обычных сообщений.

Это немного вводит в замешательство. Так как функция *MsgReceive()* может принимать и обычные сообщения, и импульсы, а функция *MsgReceivePulse()* может принимать только импульсы, то как быть с сервером, в котором применяются обе функции? Общий ответ такой. У вас есть пул потоков, выполняющих *MsgReceive()*. Этот пул потоков (один или более потоков — это зависит от числа клиентов, которое вы хотели бы обслуживать одновременно) отвечает за обработку запросов от клиентов.

Поскольку вы пытаетесь управлять численностью потоков-обработчиков, и некоторым из этих потоков может понадобиться блокироваться в ожидании импульса (например, от оборудования или от другого потока), вы блокируете поток-обработчик при помощи функции *MsgReceivePulse()*. Функция *MsgReceivePulse()* принимает только импульсы, а значит, ее применение гарантирует, что пока вы ждете импульса, к вам ненароком не просочится никакой клиентский запрос.

### **Функция *MsgDeliverEvent()***

Как было упомянуто выше в параграфе «Иерархический принцип обмена», существуют ситуации, когда приходится нарушать естественное направление передач.

Такой случай возможен, когда у вас есть клиент, который посылает серверу сообщение и при этом не хочет блокироваться, а результат может быть доступен только через некоторое время. Конечно, вы могли бы частично решить эту проблему путем применения многопоточных клиентов, выделяя в клиенте отдельный поток для блокирующих вызовов сервера, но это не всегда с успехом работает в больших системах, поскольку при большом количестве серверов количество ждущих потоков было бы слишком велико. Но допустим, вы не хотите здесь использовать многопоточность, а вместо этого вам нужно, чтобы сервер ответил клиенту сразу, и чем-то вроде «Заказ принят; я скоро вернусь». Здесь, поскольку сервер ответил, клиент теперь свободен продолжать свою работу. После того как сервер обработает запрос клиента, ему потребуется как-то сказать

клиенту «Проснись, вот твой заказ.» Очевидно, как мы это уже видели при анализе иерархического принципа обмена, сервер не должен передавать сообщения клиенту, потому что если клиент в это же время отправит сообщение серверу, это может вызывать взаимную блокировку. Так как же сервер может послать сообщение клиенту без нарушения иерархического принципа?

В действительности это составная операция. Вот как это работает:

1. Клиент создает структуру типа **struct sigevent** и заполняет ее.
2. Клиент посылает сообщение серверу, в котором запрашивает: «Сделай для меня то-то, ответ дай сразу же, а по окончании работы уведоми меня об этом при помощи структуры **struct sigevent** — структуру прилагаю».
3. Сервер принимает сообщение (которое включает в себя структуру **struct sigevent**), сохраняет структуру **struct sigevent** и идентификатор отправителя и немедленно отвечает клиенту.
4. Теперь клиент выполняется — как и сервер.
5. Когда сервер завершает работу, он использует функцию *MsgDeliverEvent()*, чтобы сообщить об этом клиенту.

Мы рассмотрим более подробно структуру **struct sigevent** в главе «Часы, таймеры и периодические уведомления», в параграфе «Как заполнить структуру **struct sigevent**», а здесь мы только предположим, что структура **struct sigevent** — это «черный ящик», который содержит некоторое событие, используемое сервером для уведомления клиента.

Поскольку сервер хранит клиентские **struct sigevent** и идентификатор отправителя, он теперь сервер может вызвать функцию *MsgDeliverEvent()*, чтобы доставить событие клиенту, как клиент того и желал:

```
int MsgDeliverEvent(int rcvid, const struct sigevent *event);
```

Обратите внимание, что функция *MsgDeliverEvent()* принимает два параметра — идентификатор отправителя (*rcvid*) и доставляемое событие (*event*). Сервер никогда не изменяет и даже не читает событие! Этот момент важен, потому что это позволяет серверу доставлять события вне зависимости от их выбранного клиентом типа, без какой бы то ни было специальной обработки на стороне сервера.

Идентификатор *rcvid* — это идентификатор отправителя, который сервер получил от клиента. Заметьте, что это определенно особый случай. Обычно, после того как сервер ответил клиенту, идентификатор отправителя прекращает иметь значение (потому что клиент уже разблокирован, и сервер не может разблокировать его заново или считать/записать данные, и т.п.). Но в нашем случае, идентификатор отправителя

содержит только информацию для ядра, какому клиенту должно быть доставлено событие. Вызывая *MsgDeliverEvent()*, сервер не блокируется — для сервера это неблокирующий вызов. Ядро доставляет событие клиенту, после чего тот выполняет какие бы то ни было соответствующие действия.

## Флаги канала

Когда мы вначале книги изучали сервер (в параграфе «Сервер»), мы упомянули, что функция *ChannelCreate()* принимает параметр *flags* (флаги); правда, тогда мы вместо этого параметра передавали ноль.

Теперь пришло время более подробно изучить назначение параметра *flags*. Рассмотрим только некоторые из возможных его значений:

### `_NTO_CHF_FIXED_PRIORITY`

Принимающий поток не изменит приоритет в зависимости от приоритета отправителя. (Мы поговорим о проблемах приоритетов более подробно в разделе «Наследование приоритетов»). Обычно (то есть если этот флаг не установлен) приоритет принимающего сообщения потока изменяется на приоритет потока-отправителя.

### `_NTO_CHF_UNBLOCK`

Ядро посылает импульс всякий раз, когда поток клиента пытается разблокироваться. Чтобы клиент мог разблокироваться, сервер должен ему ответить. Мы обсудим это ниже, потому что это имеет некоторые интересные последствия — как для клиента, так и для сервера.

### `_NTO_CHF_THREAD_DEATH`

Ядро посылает импульс всякий раз, когда заблокированный на этом канале поток «умирает». Это полезно для серверов, которые желают поддерживать фиксированную популяцию потоков в пуле, т. е. количество потоков, доступных для обслуживания запросов.

### `_NTO_CHF_DISCONNECT`

Ядро посылает импульс всякий раз после того, как уничтожается последнее из имевшихся соединений сервера с некоторым клиентом.

### `_NTO_CHF_SENDER_LEN`

Ядро доставляет серверу, наряду с остальной информацией, размер клиентского сообщения.

## Флаг `_NTO_CHF_UNBLOCK`



Присмотримся к флагу `_NTO_CHF_UNBLOCK`. Этот флаг имеет несколько особенностей при его применении, интересных и для клиента, и для сервера.

Обычно (то есть когда сервер не устанавливает флаг `_NTO_CHF_UNBLOCK`), когда клиент хочет разблокироваться от `MsgSend()` (или `MsgSendv()`, `MsgSendvs()` или другой функции этого семейства), клиент просто берет и разблокируется. Клиент может пожелать разблокироваться по приему сигнала или по тайм-ауту ядра (см. функцию `TimerTimeout()` в Справочном руководстве по Си-библиотеке, а также главу «Часы, таймеры и периодические уведомления»). Неприятный аспект этого заключается в том, что сервер понятия не имеет, что клиент уже разблокирован и больше не ожидает ответа.

Давайте предположим, что у вас многопоточный сервер, и все потоки заблокированы с помощью функции `MsgReceive()`. Клиент посылает сообщение серверу, и один из потоков сервера принимает его. Клиент блокируется, поток же сервера активно обрабатывает запрос. Но прежде, чем поток сервера сможет ответить клиенту, клиент разблокируется из своего `MsgSend()` (предположим, что по причине приема сигнала).

Не забывайте: поток сервера по-прежнему обрабатывает поступивший от клиента запрос. Но так как клиент теперь разблокирован (например, его вызов `MsgSend()` возвратил `EINTR`), он теперь может послать серверу другой запрос. Вследствие особенности архитектуры серверов в QNX/Neutrino, очередное сообщение от этого клиента принял бы другой поток сервера, но идентификатор отправителя-то остается тем же самым! Сервер не сумеет различить эти два запроса, и когда первый поток сервера завершает обработку первого запроса и отвечает клиенту, фактически он отвечает на второе сообщение, а не на первое. Итак, первый поток сервера отвечает на второе сообщение клиента.

Это плохо уже само по себе; но давайте заглянем еще на шаг вперед. Теперь второй поток завершает свою работу по обработке запроса и пробует ответить клиенту. Но поскольку первый поток сервера уже ответил этому клиенту, а значит, этот клиент уже разблокирован, то попытка второго потока сервера ответить клиенту возвратится с ошибкой.

Эта проблема встречается только в многопоточных серверах, потому что в однопоточном сервере его единственный поток был бы по-прежнему занят обработкой первого запроса клиента. Это означает, что даже если бы клиент разблокировался и снова послал сообщение серверу, он перешел бы в `SEND`-блокированное состояние (а не в `REPLY`-блокированное состояние), позволив тем самым серверу закончить обработку первого запроса, ответить клиенту (что привело бы к ошибке, потому что клиент

более не находится в REPLY-блокированном состоянии) и лишь затем принять второе сообщение. Здесь реальная проблема состоит в том, что сервер выполняет лишнюю операцию — обработку первого запроса. Операция же эта является абсолютно бесполезной, поскольку клиент больше не ожидает ее результатов.

Решение данной проблемы (в случае многопоточного сервера) заключается в том, что сервер должен при создании канала указать вызову *ChannelCreate()* флаг *\_NTO\_CHF\_UNBLOCK*. Этот флаг скажет ядру: «Сообщи мне импульсом, когда клиент попытается разблокироваться, но не позволяй ему это делать! Я разблокирую клиента сам».

Ключевым моментом здесь является то, что этот флаг сервера изменяет поведение клиентов, не позволяя им разблокироваться до тех пор, пока им это не разрешит сервер.

В однопоточном сервере происходит следующее:

Действие	Состояние клиента	Состояние сервера
Клиент посылает запрос серверу	Блокирован	Обработка
Клиент получает сигнал	Блокирован	Обработка
Ядро передает импульс серверу	Блокирован	Обработка (первого сообщения)
Сервер завершает обработку первого запроса и отвечает клиенту	Разблокирован, получены корректные данные	Обработка (импульса)

Это не помогло клиенту разблокироваться, когда он должен был это сделать, но зато обеспечило, чтобы сервер не запутался. В подобном примере сервер мог вообще проигнорировать импульс, отправленный ему ядром. Это нормально — поскольку сделано предположение, что позволить клиенту быть заблокированным до тех пор, пока сервер не подготовит данные для него, безопасно.

Если вы хотите, чтобы сервер среагировал каким-то действием на посланный ядром импульс, то существует два способа реализации этого:

- Создать еще один поток в сервере, который «слушал» бы канал на предмет импульсов от ядра. Этот второй поток будет отвечать за отмену операции, выполняемой первым потоком. Отвечать клиенту может любой из этих двух потоков.

- Не выполнять задание клиента в потоке непосредственно, а поставить его в очередь заданий. Это обычно делается в приложениях, где сервер целенаправленно направляет задания клиента в очередь, и сервер

является управляемым по событиям. Обычно одно из получаемых сервером сообщений указывает на то, что работа клиента завершена, и что пора отвечать. Когда в этом случае приходит импульс от ядра, сервер отменяет выполняемую для данного клиента работу и отвечает.

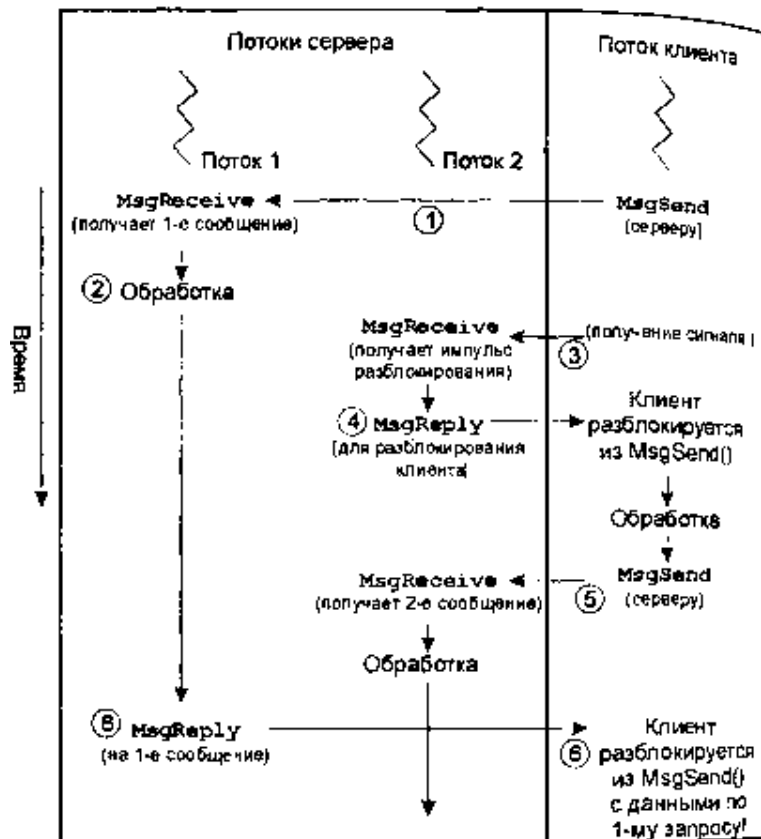
Какой из методов вам выбирать — это будет зависеть от типа работы, которую выполняет сервер. В первом случае сервер активно выполняет работу для клиента, так что у вас просто не будет иного выбора, чем применить второй поток, который слушал бы импульсы от ядра, сообщающие о разблокировании (далее — «импульсы разблокирования» — *прим. ред.*). Конечно, вы могли бы также организовать программный опрос в пределах потока для проверки, не пришел ли импульс, но программный опрос обычно удручает.

Во втором случае работу делает не сам сервер, а кто-то другой — возможно, оборудование, которому приказано «сходи и набери данных». При таком варианте поток сервера будет в любом случае блокирован по функции *MsgReceive()*, ожидая от оборудования признака завершения операции.

В обоих случаях сервер *обязан* ответить клиенту, иначе клиент останется заблокированным.

### ***Проблема синхронизации***

Но даже если вы используете флаг `_NTO_CHF_UNBLOCK`, как это описано выше, остается еще одна проблема синхронизации. Предположим, что несколько потоков вашего сервера заблокированы по функции *MsgReceive()* в ожиданий сообщения или импульса, и клиент посылает серверу сообщение. Один из потоков разблокируется и начнет обрабатывать запрос клиента. В процессе этого клиент вдруг пожелает разблокироваться, и ядро сгенерирует предупреждающий об этом импульс (импульс разблокирования). Другой поток сервера примет этот импульс. Фактически здесь мы имеем гонки потоков — первый поток на момент получения вторым импульса мог быть уже почти готов ответить клиенту. Если ответит второй поток (тот, который получил импульс), то есть шанс, что клиент разблокируется и передаст серверу еще одно сообщение. При этом первый поток сервера получает шанс завершить работу по первому запросу и ответить полученными данными на второй запрос:



Путаница в многопоточном сервере.

Также возможен такой вариант: поток, получивший импульс, готовится ответить клиенту, а в это время отвечает первый поток. Получается то же самое, что и раньше — первый поток разблокирует клиента, клиент передает второй запрос, второй поток (тот, который получил импульс) разблокирует клиента по второму запросу.

Здесь мы имеем ситуацию с двумя параллельными потоками обработки (один вызван сообщением клиента и один — импульсом). Обычно в таких ситуациях применяются мутексы.

К сожалению, это привело бы к проблеме — мутекс нужно было бы захватить немедленно после вызова `MsgReceive()` и освободить перед вызовом `MsgReply()`. Это, конечно, будет работать, но это войдет в противоречие с самим предназначением импульса разблокирования! (Сервер мог бы либо получить сообщение и игнорировать импульс разблокирования, пока не ответит клиенту, либо получить импульс разблокирования и отменить второй запрос клиента.)

Многообещающим решением (но в конечном счете все равно обреченным на провал) выглядит применение «мелкозернистого» мутекса, то есть мутекса, который захватывается и освобождается только в небольших областях потока управления (как, собственно, и предполагается

использовать мутекс — вместо блокирования целого раздела, как это было предложено выше). Можно было бы организовать в сервере флаг «Мы уже ответили?», сбрасывая его при приеме сообщения и снова устанавливая после ответа на него. Непосредственно перед ответом на сообщение проверяется состояние этого флага. Если флаг указывает на то, что ответ уже произведен, то отвечать не надо. Мутекс при этом следовало бы захватывать и освобождать только в областях проверки и установки/сброса флага.

К сожалению, это не будет работать, потому что мы далеко не всегда имеем дело с двумя потоками управления — не всегда же клиент будет получать сигнал в процессе обработки запроса, порождая тем самым импульс разблокирования. Ниже приведен сценарий, где предложенная схема не сработает:

- Клиент передает сообщение серверу; после этого клиент блокируется, а сервер переключается в режим обработки.
- Поскольку сервер принял запрос от клиента, флаг ответа сбрасывается в 0, указывая этим, что мы все еще должны ответить.
- Сервер отвечает клиенту в нормальном режиме (потому что флаг был установлен в 0) и устанавливает флаг в 1, указывая этим, что если придет импульс разблокирования, то его следует игнорировать.
- (Проблемы начинаются здесь.) Клиент посылает серверу второе сообщение и почти немедленно после этого получает сигнал; ядро передает серверу импульс разблокирования.
- Поток сервера, который принял сообщение, намеревался захватить мутекс для проверки состояния флага, но еще не успел это сделать, поскольку был вытеснен.
- Другой поток сервера получает импульс разблокирования, и поскольку флаг по-прежнему находится в состоянии 1 с момента последней установки, игнорирует этот импульс.
- Первый поток сервера захватывает мутекс и сбрасывает флаг.
- К этому моменту событие разблокирования клиента потеряно.

Даже если вы усовершенствуете флаг, задав для него большее количество состояний (таких как «импульс получен», «дан ответ на импульс», «сообщение получено», «дан ответ на сообщение»), у вас по-прежнему останется проблема гонок при синхронизации, потому что не существует атомарной операции, связывавшей бы флаг ответа и функции приема сообщения и ответа на него. (Именно это и определяет суть проблемы — небольшие окна во времени, одно после *MsgReceive()*, но до сброса флага, и второе — после того, как флаг установлен, но до вызова

*MsgReply()*.) Единственный способ обойти проблему состоит в том, чтобы переложить работу по отслеживанию состояния флага на ядро.

### ***Применение флага `_NTO_MI_UNBLOCK_REQ`***

К счастью, ядро отслеживает для нас состояние этого флага — под это отведен один бит в информационной структуре сообщения (это структура типа `struct _msg_info`, которая передается функции *MsgReceive()* в качестве последнего параметра, и которую можно также впоследствии получить по идентификатору отправителя, вызвав функцию *MsgInfo()*).

Этот флаг называется `_NTO_MI_UNBLOCK_REQ` и устанавливается в 1, если клиент желает разблокироваться (например, получив сигнал).

Это означает, что в многопоточном сервере у вас будет поток-обработчик, выполняющий работу для клиента, и еще один поток, который будет получать сообщения разблокирования (или другие; но сконцентрируемся пока только на сообщениях разблокирования). Когда от клиента приходит сообщение разблокирования, установите для себя флаг, чтобы дать вашей программе знать о желании потока разблокироваться.

Существуют две ситуации, которые необходимо рассмотреть:

- поток-обработчик заблокирован;
- поток-обработчик активен (выполняется).

Если поток-обработчик заблокирован (например, в ожидании ресурса), то поток, получивший сообщение разблокирования, должен его разбудить. Когда поток-обработчик активизируется, он должен проверить состояние флага `_NTO_MI_UNBLOCK_REQ` и, если флаг установлен, дать ответ о ненормальном завершении. Если флаг сброшен, то поток может спокойно выполнять все, что ему необходимо для нормальной обработки запроса.

В противном случае, если поток-обработчик активен, он должен периодически проверять «флаг, выставляемый в его отношении» потоком, принимающим сообщение разблокирования, и если флаг установлен в 1, он должен ответить клиенту с кодом ошибки. Заметьте, что это всего-навсего оптимизация: в неоптимизированном случае поток-обработчик постоянно вызывал бы функцию *MsgInfo()* по идентификатору отправителя и проверял бит `_NTO_MI_UNBLOCK_REQ` самостоятельно.

## Обмен сообщениями в сети

☞ Прозрачный обмен сообщениями в сети не поддерживается в версии QNX/Neutrino 2.00, но это намечено к реализации в более поздних версиях данной ОС. (Поддержка этого механизма реализована в QNX/Neutrino, начиная с версии 2.11, и присутствует в QNX Realtime Platform, начиная с релиза 6.1.0 — *прим. ред.*) Я привожу рассмотрение этого вопроса в данной книге по двум причинам: 1) Когда этот механизм будет реализован, им можно будет воспользоваться. 2) Это настолько изящно, что грех не рассказать об этом!

Чтобы не вносить излишней путаницы, до сих пор я избегал вопроса о применении обмена сообщениями в сети, хотя реально это основополагающий фактор гибкости QNX/Neutrino!

Все, что вы узнали из книги до этого момента, применимо и к передаче сообщений по сети.

Ранее в этой главе я демонстрировал пример:

```
#include <fcntl.h>
#include <unistd.h>

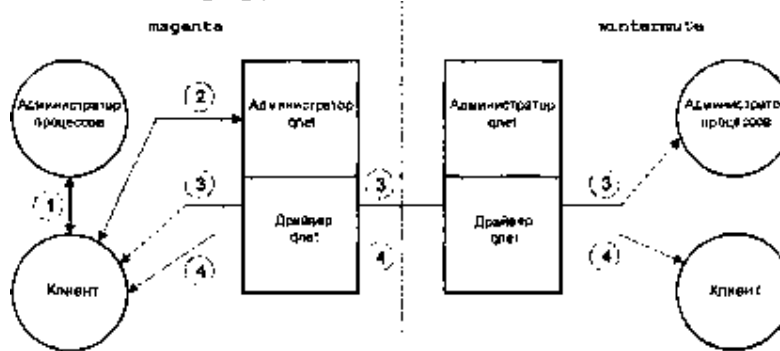
int main (void) {
    int fd;
    fd = open("/net/wintermute/home/rk/filename", O_WRONLY);
    write(fd, "Это обмен сообщениями\n", 24);
    close(fd);
    return(EXIT_SUCCESS);
}
```

В то время, я говорил, что это был пример «обмена сообщениями в сети». Клиент соединяется с сервером, определяемым тройкой ND/PID/CHID (и который оказывается на другом узле), а сервер выполняет на своем канале *MsgReceive()*. Клиент и сервер в данном случае абсолютно аналогичны клиенту и серверу в варианте с локальным узлом. Собственно, прекратить читать книгу можно прямо здесь — в передаче сообщений по сети нет ничего хитрого. На вам, наверное, любопытно как все это происходит? Читайте дальше!

Теперь, когда мы уже рассмотрели в подробностях особенности локального обмена сообщениями, мы можем более углубленно обсудить,

как осуществляется передача сообщений в сети. И хотя это обсуждение может показаться сложным, на самом деле все сводится к двум этапам: этапу разрешения имен и этапу собственно передачи сообщений.

Вот рисунок, иллюстрирующий эти этапы:



Обмен сообщениями в сети. Отметьте, что модуль **qnet** разделен на две части.

На данном рисунке наш узел называется **magenta**, а целевой узел по аналогии с примером называется **wintermute**.

Рассмотрим взаимодействия, которые происходят, когда программа-клиент использует **qnet**, чтобы обратиться к серверу через сеть:

1. Функции *open()* клиента было предписано открыть файл с именем, которое начинается с **/net**. (Имя **/net** — имя по умолчанию, объявляемое администратором **qnet** — см. документацию по QNX/Neutrino, раздел **npi-qnet**). Клиент понятия не имеет, кто именно отвечает за конкретное имя пути, поэтому он соединяется с администратором процессов (шаг 1), чтобы выяснить, кому принадлежит ресурс. Это выполняется автоматически и не зависит от того, передаем ли мы сообщения по сети или нет. Поскольку все ресурсы, имена которых начинаются с **/net**, принадлежат администратору **qnet**, администратор процессов отвечает клиенту, что относительно этого имени пути надо спросить администратора **qnet**.

2. Клиент теперь посылает сообщение потоку администратора **qnet**, надеясь, что тот будет способен обработать запрос. Однако администратор **qnet** на этом узле не может предоставить клиенту конечный сервис, поэтому он сообщает клиенту, что тот должен обратиться к администратору процессов на узле **wintermute**. (Это делается специальным перенаправляющим сообщением, в котором содержатся ND/PID/CHID сервера, к которому надо обратиться взамен.) Это перенаправление также автоматически обрабатывается клиентской библиотекой.

3. Клиент соединяется с администратором процессов на узле **wintermute**. Это включает в себя отправку сообщения другому узлу с помощью драйверного потока **qnet**. Процесс **qnet** клиентского узла



получает сообщение и транспортирует его через сетевую среду удаленному **qnet**, который, в свою очередь, доставляет его администратору процессов на узле **wintermute**. Администратор процессов этого узла разрешает остальную часть имени пути (в нашем примере это **/home/rk/filename**) и отвечает перенаправляющим сообщением. Это сообщение передается обратно — через **qnet** сервера по сетевой среде к **qnet** клиента и, наконец, самому клиенту. Поскольку в этом сообщении содержатся ND/PID/CHID нужного сервера, предоставляющего конечный сервис, клиент теперь знает, к кому обращаться (в нашем примере это администратор удаленной файловой системы).

4. Теперь клиент посылает запрос непосредственно нужному серверу. Маршрут следования сообщения здесь идентичен описанному в предыдущем пункте, за исключением того, что на этот раз связь с сервером осуществляется напрямую, а не через администратор процессов.

После того как пройдены этапы 1 и 3, все дальнейшие коммуникации осуществляются аналогично этапу 4. В вышеприведенном примере все сообщения типа *open()*, *read()* и *close()* идут по маршруту, описанном в этапе 4. Заметьте, что вся последовательность рассмотренных событий была запущена вызовом *open()*, но само сообщение *open()* все равно дошло до сервера-адресата так, как это описано этапом 4.

☞ Для особо любопытных: на самом деле я пропустил в изложении один этап. На этапе 2, когда клиент спрашивает **qnet** об узле **wintermute**, **qnet** должен сначала выяснить, кто такой **wintermute**. Это может привести к еще одной сетевой транзакции для разрешения имени узла. Приведенный выше рисунок корректен, если предположить, что **qnet** заранее знал про узел с именем **wintermute**.

Мы еще вернемся к сообщениям, используемым функциями *open()*, *read()* и *close()* (а также другими функциями) в главе «Администраторы ресурсов».

## Особенности обмена сообщениями в сети

Итак, как только соединение установлено, все дальнейшие операции обмена сообщениями осуществляются в соответствии с этапом 4, как указано на рисунке. Это может привести вас к ошибочному представлению, что передача сообщений по сети идентична локальной. К сожалению, это не так. Вот список отличий:

- более длительные задержки;
- функция *ConnectAttach()* возвращает признак успешного соединения независимо от того, является ли узел доступным или нет — реальный признак ошибки проявляется только при первой попытке передать сообщение;
- функция *MsgDeliverEvent()* не обеспечивает достаточной надежности;
- функции *MsgReply()*, *MsgRead()*, *MsgWrite()* являются блокирующими вызовами (в локальном варианте они не являлись таковыми);
- функция *MsgReceive()* не будет принимать все данные, посланные клиентом; сервер будет должен вызывать функцию *MsgRead()* для получения окончательных остальных данных.

### ***Более длительные времена задержки***

Поскольку передача сообщений теперь выполняется в некоторой среде, а не прямым копированием «память-память» под управлением ядра, можно смело ожидать, что затраты времени на передачу сообщения будут существенно выше (100-Мбитный Ethernet в сравнении с 128-битным динамическим ОЗУ с тактированием 100 МГц будет ориентировочно на один или два порядка медленнее). В дополнение к этому также будут сказываться накладные расходы протокола и повторные попытки передачи в сбойных сетях.

### ***Воздействие на функцию *ConnectAttach()****

Когда вы вызываете функцию *ConnectAttach()*, вы задаете ей идентификаторы ND, PID и CHID. Все, что при этом происходит в QNX/Neutrino, заключается в возврате ядром идентификатора соединения драйверному потоку **qnet**, изображенному выше на рисунке. Поскольку никакого сообщения еще не отправлено, вы не имеете информации о том, доступен ли узел, к которому вы только что подсоединились, или нет. В обычном случае это не проблема, потому что большинство клиентов не будет самостоятельно вызывать *ConnectAttach()* и скорее воспользуется библиотечной функцией *open()*, которая перед передачей сообщения «open» сама вызывает *ConnectAttach()*. Это практически немедленно дает информацию о доступности удаленного узла.

### ***Воздействие на функцию `MsgDeliverEvent()`***

Когда сервер вызывает функцию `MsgDeliverEvent()` локально, ответственность за доставку события целевому потоку ложится на ядро. В сетевом варианте сервер также может вызывать функцию `MsgDeliverEvent()`, но на этот раз ядро доставит «заготовку» этого события администратору `qnet`, возлагая на него ответственность за доставку этой «заготовки» удаленному `qnet`, который уже доставит реальное событие клиенту. Так вот, на стороне сервера с этим вызовом могут возникнуть проблемы, потому что он не является блокирующим. Это означает, после вызова `MsgDeliverEvent()` сервер продолжает выполняться, и поздно уже оглядываться и говорить «Знаете, очень не хочется вас огорчать, но помните тот вызов `MsgDeliverEvent()`? Так вот, он не сработал...»

### ***Воздействие на функции `MsgReply()`, `MsgRead()` и `MsgWrite()`***

Чтобы уберечь функции `MsgReply()`, `MsgRead()` и `MsgWrite()` от вышеупомянутой проблемы `MsgDeliverEvent()`, эти функции при использовании их в сети преобразуются в блокирующие вызовы. В локальном случае они бы просто передали данные и разблокировались; в сети же мы должны либо удостовериться, что данные были доставлены клиенту (в случае `MsgReply()`), либо собственно передать данные по сети клиенту или от него (в случае двух других функций).

### ***Воздействие на функцию `MsgReceive()`***

Функция `MsgReceive()` (при использовании в сети) тоже оказывается под влиянием. На момент разблокирования функции `MsgReceive()` на стороне сервера `qnet` может еще не успеть передать все данные клиента. Это делается из соображений производительности.

В структуре `struct _msg_info`, передаваемой функции `MsgReceive()` в качестве последнего параметра (мы подробно рассматривали эту структуру в параграфе «Кто послал сообщение?»), есть два флага:

`msglen` Указывает на фактическое количество данных, переданное функцией `MsgReceive()` (`qnet` любит передавать по 8Кб за один раз).

`srcmsglen` Указывает на количество данных, которое клиент хотел передать

(определяется клиентом).

Таким образом, если бы клиент желал передать 1 мегабайт данных по сети, *MsgReceive()* сервера разблокировалась бы, установив параметр *msglen* в значение 8192 (указывая, что 8192 байта доступны в буфере); параметр *srcmsglen* при этом равнялся бы 1048576 (указывая, что клиент пытался переслать 1 мегабайт данных).

Затем сервер использует *MsgRead()* для получения остальной части данных из адресного пространства клиента.

### Несколько замечаний о дескрипторах узлов

Еще одна любопытная вещь, которой мы еще не касались в обсуждениях обмена сообщениями, — это дескрипторы узлов, для краткости обозначаемые «ND» (сокр. от Node Descriptor — *прим. ред.*).

Вспомните: в наших примерах мы использовали символьные имена узлов, например, */net/wintermute*. В QNX4 (предыдущая версия QNX до появления QNX/Neutrino) вся работа в сети была основана на концепции идентификатора узла, небольшого целого числа, уникально определяющего узел сети. Таким образом, в терминах QNX4 мы говорили бы что-то вроде «узел 61», или «узел 1», и это отражалось бы и на вызовах функций тоже.

При работе в QNX/Neutrino все узлы внутренне представляются 32-разрядными числами, но эти числа не являются уникальными в сети! Я имею в виду, что узел *wintermute* может думать об узле *spud* как об узле с дескриптором 7, в то время как сам узел *spud* может думать, что дескриптор 7 соответствует узлу *magenta*. Поясню подробнее, чтобы дать полную картину происходящего. В приведенной ниже таблице сведены примерные дескрипторы узлов, которые могли бы использоваться для описания трех узлов: *wintermute*, *spud* и *foobar* (не путать с аббревиатурой FUBAR — *прим. ред.* :-):

Узел	<i>wintermute</i>	<i>spud</i>	<i>foobar</i>
<i>wintermute</i>	0	7	4
<i>spud</i>	4	0	6
<i>foobar</i>	5	7	0

Обратите внимание, что каждый узел считает свой собственный дескриптор нулевым. Также отметьте, что для узла *spud* оба узла *wintermute* и *foobar* имеют дескриптор 7. Однако, для узла *foobar* узел *wintermute* имеет дескриптор 4, а узел *spud* — 6. Как я и упоминал раньше, эти номера не уникальны в сети, но они уникальны на каждом узле. Вы

можете относиться к ним, как к файловым дескрипторам — два процесса, когда обращаются к одному и тому же файлу, могут иметь для него как одинаковый дескриптор, так и нет — все зависит от того, кто, когда и который файл открывает.

К счастью, вам не надо беспокоиться о дескрипторах узлов по ряду причин:

- Большинство осуществляемых вами операций обмена сообщениями «с внешним миром» будут реализовываться с помощью вызовов функций высокого уровня (таких как функция *open()*, приведенная в примере выше).
- Дескрипторы узлов не кэшируются — предполагается, что получив дескриптор, вы используете немедленно и забудете про него.
- Существует ряд библиотечных функций, предназначенных для преобразования имени пути (например, */net/magenta*) в дескриптор узла.

Чтобы работать с дескрипторами узлов, вам понадобится подключить файл *<sys/netmgr.h>*, потому что он содержит прототипы семейства функций *netmgr\_\**().

Для преобразования строки в дескриптор узла используется функция *netmgr\_strtond()*. После получения дескриптора узла его следует сразу же применить в вызове функции *ConnectAttach()*. Не пытайтесь сохранять его какой-либо структуре данных! Веским основанием для этого является то, что администратор сети может решить повторно использовать дескриптор после отключения всех соединений с узлом.

Так что если вы получили дескриптор «7» для узла */net/magenta*, подсоединились к нему, передали сообщение и затем отсоединились, то существует возможность того, что администратор сети заново назначит дескриптор «7» другому узлу.

Поскольку дескрипторы узлов в сети не уникальны, возникает вопрос: «А как передавать эти штуки по сети?» Очевидно, взгляды узла *magenta* и узла *wintermute* на дескриптор «7» будут радикально отличаться. Существуют два способа решения этой проблемы:

- Не передавать по сети дескрипторы узлов и пользоваться символьными именами (например, */net/wintermute*).
- Применять функцию *netmgr\_remote\_nd()*.

Первый метод хорош как универсальное решение. Второй метод достаточно удобен на практике.

```
int netmgr_remote_nd(int remote_nd, int local_nd);
```

Эта функция принимает два параметра, где *remote\_nd* — дескриптор узла целевой машины, а *local\_nd* — дескриптор узла, который нужно преобразовать из точки зрения локальной машины в точку зрения целевой.

Результатом является дескриптор узла, корректный с точки зрения заданной удаленной машины.

Например, пусть **wintermute** — имя нашей локальной машины. У нас есть дескриптор узла «7», который является корректным на нашей локальной машине и указывает на узел **magenta**. Мы хотели бы выяснить, какой дескриптор узла использует узел **magenta** для связи с нашим узлом:

```
int remote_nd;
int magenta_nd;
magenta_nd = netmgr_strtond("/net/magenta", NULL);
printf("ND узла magenta - %d\n", magenta_nd);
remote_nd = netmgr_remote_nd(magenta_nd, ND_LOCAL_NODE);
printf("С точки зрения узла magenta, наш ND - %d\n",
      remote_nd);
```

Эта программа могла бы вывести что-то вроде следующего:

```
ND узла magenta - 7
```

```
С точки зрения узла magenta, наш ND - 4
```

Это говорит о том, что на узле **magenta** нашему узлу соответствует дескриптор «4». (Обратите внимание на использование специальной константы **ND\_LOCAL\_NODE**, которая в действительности равна нулю, для указания на «локальный узел»).

Теперь вернемся к тому, о чем мы говорили в разделе «Кто послал сообщение?»). Параметр **struct \_msg\_info** содержит, среди всего прочего, два дескриптора узлов:

```
struct _msg_info {
    int nd;
    int srcnd;
    ...
};
```

Мы определили в описании для этих двух полей, что:

- *nd* — дескриптор принимающего узла с точки зрения передающего;
- *srcnd* — дескриптор передающего узла с точки зрения принимающего.

Так, для приведенного выше примера, где узел **wintermute** — локальный, а узел **magenta** — удаленный, когда узел **magenta** посылает нам (узлу **wintermute**) сообщение, эти поля заполняются следующим образом:

- *nd* равен 7;
- *srcnd* равен 4.

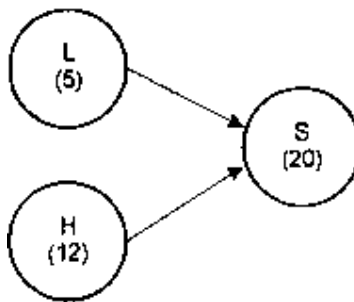
## Наследование приоритетов

Одним из интересных моментов в операционных системах реального времени является феномен инверсии приоритетов.

Инверсия приоритетов наблюдается, например, в случае, когда поток с низким приоритетом потребляет все процессорное время, в то время как в состоянии готовности находится поток с более высоким приоритетом.

Вы, наверное, сейчас думаете: «Минуточку! Вы утверждали ранее, что поток с более высоким приоритетом будет всегда вытеснять поток с более низким приоритетом! Как же такое может быть?»

Вы абсолютно правы. Поток с более высоким приоритетом всегда будет вытеснять поток с более низким приоритетом. Но при этом все-таки может произойти кое-что интересное. Давайте рассмотрим сценарий с тремя потоками (в трех различных процессах, для простоты рассмотрения), где «L» — поток с низким приоритетом, «H» — поток с высоким приоритетом, и «S» — сервер. На рисунке показаны все три потока со своими приоритетами.



Три потока с различными приоритетами.

В данный момент выполняется поток H. Поток сервера S, имеющему наивысший приоритет, пока делать нечего, так что он находится в режиме ожидания и блокирован на функции *MsgReceive()*. Поток L и хотел бы работать, но его приоритет ниже, чем у потока H, который выполняется в данный момент. Все как вы и предполагали, да?

А теперь представьте себе, что поток H принял решение «прикорнуть» на 100 миллисекунд — возможно, чтобы подождать медленное оборудование. Теперь выполняется поток L.

Вот тут-то все интересное и начинается.

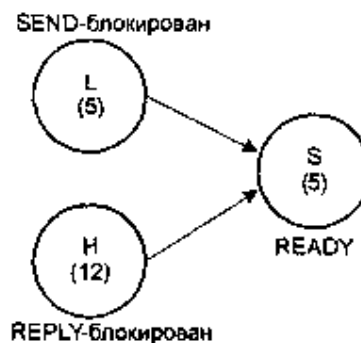
В пределах своего нормального функционирования поток L посылает сообщение потоку сервера S, принуждая этим сервер S перейти в состояние READY и (поскольку поток S имеет высший приоритет из всех готовых к

выполнению потоков) начать выполняться. К великому сожалению, сообщение, которое поток L направил к потоку сервера S, было сформулировано так: «Вычислить значение  $\Pi$  с точностью до 50 знаков после запятой».

Очевидно, это займет более чем 100 миллисекунд. Поэтому, когда 100 миллисекунд сна потока H истекут, поток H перейдет в состояние READY — угадайте, что дальше? Поток H не активизируется, постольку в состоянии READY находится поток S, имеющий более высокий приоритет!

Что здесь произошло? Произошло то, что поток с низким приоритетом «отстранил» от работы поток с более высоким приоритетом путем передачи процессора потоку с еще более высоким приоритетом. Это явление называется инверсией приоритетов.

Чтобы научиться не допускать таких вещей, мы должны поговорить о наследовании приоритетов. Простой вариант реализации наследования приоритета — заставить сервер S унаследовать приоритет клиентского потока:



Блокированные потоки.

При таком сценарии по истечении 100-миллисекундного интервала бездействия потока H этот поток переходит в состояние READY и немедленно ставится на выполнение как имеющий наивысший приоритет.

Неплохо; однако, здесь есть еще один тонкий момент.

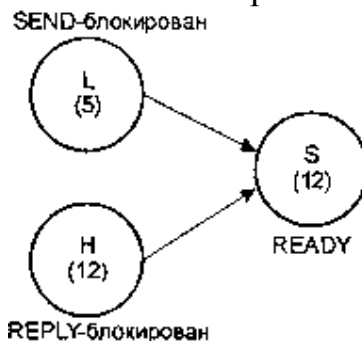
Предположим, что потоку H вдруг становится нужно выполнить какие-то вычисления — например, найти 5034-е по порядку простое число. Он посылает сообщение потоку сервера S и блокируется.

Однако, в данный момент S по-прежнему вычисляет значение  $\Pi$ , находясь на приоритете 5! В нашей выбранной для примера системе наверняка достаточно других потоков, имеющих приоритет выше, чем 5, которым тоже нужен процессор. Это автоматически значит, что процессорного времени на вычисление значения  $\Pi$  у S остается не так уж и много.



Это еще одна форма инверсии приоритетов. В этом случае поток с низким приоритетом помешал потоку с более высоким приоритетом получить доступ к ресурсу. Сравните это с первой формой инверсии приоритета, где поток с низким приоритетом реально потреблял ресурсы процессора — в рассматриваемом сейчас случае этот поток не дает более приоритетному потоку доступа к ресурсам процессора, но сам при этом непосредственно их не потребляет.

К счастью, данная проблема решается тоже достаточно просто. Достаточно увеличить приоритет сервера так, чтобы он был равен наивысшему из приоритетов всех заблокированных клиентов:



Повышение приоритета сервера.

Здесь мы немного «обделяем» другие потоки, позволяя заданию потока L выполняться с приоритетом выше, чем он сам, но зато гарантируем, что поток H получит свою заслуженную порцию процессорного времени.

### Так в чем тут хитрость?

Никакой хитрости нет, QNX/Neutrino делает все для вас автоматически.

☞ В QNX/Neutrino этот механизм реализован только на один уровень вглубь, так что если клиент посылает серверу сообщение, а этот сервер, в свою очередь, передает это сообщение другому серверу, то второй сервер наследует *нормальный* приоритет первого, а не приоритет, который первый унаследовал от клиента. Это означает, что если на первом сервере блокируется поток с более высоким приоритетом, то соответственно повышен будет приоритет только первого сервера (а поскольку первый сервер в этот момент заблокирован на втором, приоритет которого уже не повышается и остается прежним, толку от этого не будет абсолютно никакого). Будьте внимательны!

Однако, и здесь есть еще одна тонкость. Как обеспечить возврат приоритета на тот уровень, который был до изменения?

Ваш сервер работает, обслуживает запросы клиентуры и автоматически регулирует свой приоритет каждый раз, когда ему приходится разблокироваться из функции *MsgReceive()*. Но когда он должен восстанавливать прежнее значение приоритета, которое было до вызова *MsgReceive()*?

Рассмотрим два варианта развития событий.

- После обслуживания клиента сервер выполняет еще какие-то дополнительные действия. Это он должен сделать на своем приоритете, а не на приоритете клиента.

- После обслуживания клиента сервер немедленно вызывает *MsgReceive()* снова для обработки следующего запроса.

В первом случае для сервера было бы некорректно работать на приоритете клиента, поскольку он больше не делает для этого клиента никакой работы. Решение здесь очень простое. Используйте функцию *pthread\_setschedparam()* (мы ее обсуждали в главе «Процессы и потоки») для возврата приоритету нужного значения.

Что касательно второго случая, то ответ достаточно прост. Кому какое дело?

Подумайте об этом. Какая разница, станет сервер RECEIVE-блокированным на приоритете 29 или на приоритете 2?

Главное — что он RECEIVE-блокирован! А коль скоро в этом состоянии он не расходует процессорное время, его приоритет является несущественным. Как только функция *MsgReceive()* разблокирует сервер, сервером будет унаследован приоритет нового клиента, и все будет работать как полагается.

## Резюме

Обмен сообщениями представляет собой чрезвычайно мощную концепцию и является одним из основополагающих принципов, на которых построена QNX/Neutrino (как и все предыдущие версии QNX).

С помощью механизма обмена сообщениями клиент и сервер обмениваются информацией (между потоками в пределах одного процесса, между потоками в различных процессах на том же самом узле или между потоками в различных процессах на различных узлах сети). Клиент посылает сообщение и блокируется до тех пор, пока сервер не примет сообщение, не обработает его и не ответит на него.

Основные преимущества передачи сообщений:

- Содержание сообщения не зависит от местоположения адресата (локально или удаленно в сети).
- Сообщения обеспечивают четкую границу развязки клиентов и серверов.
- Неявные автоматические механизмы синхронизации и соблюдения очередности сообщений упрощают проектирование ваших приложений.

## **Глава 3**

# **Часы, таймеры и периодические уведомления**

## Часы и таймеры

Пришло время рассмотреть все, что относится ко времени в QNX/Neutrino. Мы увидим, как и почему мы должны использовать таймеры, а также рассмотрим теоретические положения, которые этому сопутствуют. Далее мы обсудим способы опроса и настройки часов реального времени.

Давайте рассмотрим типовую техническую систему — скажем, автомобиль. В этом автомобиле у нас есть ряд программ, большинство из которых выполняются с различными приоритетами. Некоторые из этих программ необходимы для обеспечения реакции на внешние события (например, тормоза или радиоприемник), другие же должны срабатывать периодически (например, система диагностики).

## Периодические процессы

Так как же обеспечивается «периодическая» работа системы диагностики? Можно вообразить себе некоторый процесс, выполняемый процессором нашего автомобиля и делающий нечто подобное следующему:

```
// Процесс диагностики
```

```
int main(void) // Игнорируем аргументы
{
    for (;;) {
        perform_diagnostics();
        sleep(15);
    }
    // Сюда мы не дойдем
    return (EXIT_SUCCESS);
}
```

Видно, что процесс диагностики выполняется бесконечно. Он запускает цикл работ по диагностике, затем «засыпает» на 15 секунд, потом «просыпается», и все повторяется заново.

Если оглянуться назад в мрачные и смутные однозадачные времена, когда один процессор обслуживал одного пользователя программы такого сорта реализовывались путем выполнения функцией *sleep()* активного ожидания. Для этого вам было необходимо узнать быстродействие вашего процессора и написать свою собственную функцию *sleep()*, например:

```

void sleep(int nseconds) {
    long i;
    while (nseconds-->0) {
        for (i = 0; i < CALIBRATED_VALUE; i++);
    }
}

```

В те дни, поскольку в машине не выполнялось никаких других задач, такие программы не составляли большой проблемы, поскольку никакой другой процесс не беспокоило, что вы используете своей функцией *sleep()* все 100% ресурсов процессора.

☞ Даже в наши дни мы иногда отдаем все 100% ресурсов процессора, чтобы отмерить время. В частности, функция *nanospin()* применяется для отсчета времени с очень большой точностью, но делает это за счет монопольного захвата процессора на своем приоритете. Пользуйтесь с осторожностью!

Если вы должны были реализовать некоторое подобие «многозадачного режима», то это обычно делалось путем применения процедуры прерывания, которая либо срабатывала от аппаратного таймера, либо выполнялась в пределах периода «активного ожидания», оказывая при этом некоторое воздействие на калибровку отсчета времени. Это обычно не вызывало беспокойства.

К счастью, в решении этих проблем мы уже ушли далеко вперед. Вспомните параграф «Диспетчеризация и реальный мир» (глава «Процессы и потоки»), там описываются причины, по которым ядро выполняет перепланирование потоков. Причины могут быть следующие:

- аппаратное прерывание;
- системный вызов;
- сбой (исключение).

В данной главе мы подробно проанализируем две первые причины из вышеуказанного списка — аппаратные прерывания и системные вызовы.

Когда поток вызывает функцию *sleep()*, код, содержащийся в Си-библиотеке, в конечном счете делает системный вызов. Этот вызов приказывает ядру отложить выполнение данного потока на заданный интервал времени. Ядро удаляет поток из рабочей очереди и включает таймер.

Все это время ядро принимает регулярно поступающие аппаратные прерывания таймера. Положим для определенности, что эти аппаратные прерывания происходят ровно каждые 10 миллисекунд.

Давайте немного переформулируем это утверждение: каждый раз, когда такое прерывание обслуживается соответствующей подпрограммой обработки прерывания (ISR) ядра, это значит, что истек очередной 10-миллисекундный интервал. Ядро отслеживает время суток путем увеличения специальной внутренней переменной на значение, соответствующее 10 миллисекундам, с каждым вызовом обработчика прерывания.

Так что, реализуя 15-секундный таймер, ядро в действительности выполняет следующее:

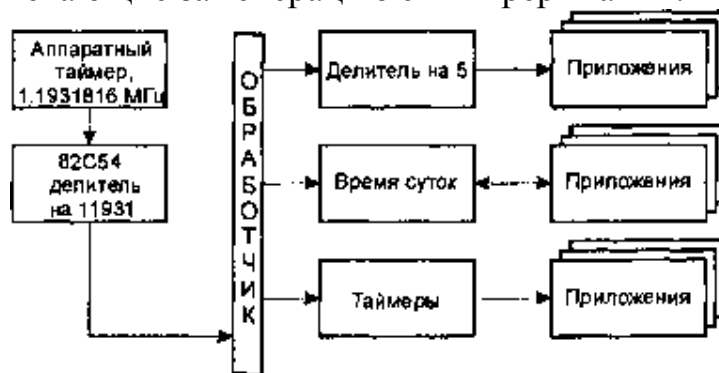
- Устанавливает переменную в текущее время плюс 15 секунд.
- В обработчике прерываний сравнивает эту переменную с текущим временем.
- Когда текущее время станет равным (или больше) данной переменной, поток снова ставится в очередь готовности.

При использовании множества параллельно работающих таймеров — например, когда необходимо активизировать несколько потоков в различные моменты времени — ядро просто ставит запросы в очередь, отсортировав таймеры по возрастанию их времени истечения (в голове очереди при этом окажется таймер с минимальным временем истечения). Обработчик прерывания будет анализировать только переменную, расположенную в голове очереди.

## Источники прерываний таймера

На этом мы, пожалуй, закончим наш краткий экскурс по стране таймеров и перейдем к вещам, которые уже не так очевидны.

Откуда возникают прерывания таймера? На рисунке ниже приведены аппаратные компоненты (и некоторые характерные для PC значения параметров), отвечающие за генерацию этих прерываний.



Источники прерываний таймера в PC.

Из рисунка видно, что в PC используется высокочастотный аппаратный генератор синхроимпульсов (МГц-диапазона). Высокочастотный меандр делится при помощи аппаратного счетчика (на рисунке — микросхема Intel 82C54), который понижает частоту импульсов до сотен килогерц или сотен герц (диапазон, в котором их уже может обработать ISR). ISR таймера входит в состав ядра и взаимодействует непосредственно с его кодом и внутренними структурами данных. В процессорах архитектуры не-x86 (MIPS, PowerPC) тоже происходит подобная последовательность событий; в некоторых микросхемах аппаратный таймер может быть непосредственно встроен в процессор.

Отметим, что импульсы высокой частоты делятся на целочисленный делитель. Это означает, что результирующий период импульсов не будет точно равен 10 миллисекундам, потому что исходный период 10 миллисекундам не кратен. Поэтому ISR ядра из вышеприведенного примера будет реально вызываться по истечении каждых 9.9999296004 миллисекунд.

Большое дело, скажете вы, ну и что? Ну ладно, для нашего 15-секундного счетчика это годится. 15 секунд — это 1500 отсчетов таймера; расчеты показывают, что погрешность будет в районе 106 микросекунд:

$$\begin{aligned} 15 \text{ с} - 1500 * 9.9999296004 \text{ мс} &= \\ &= 15000 \text{ мс} - 14999.8944006 \text{ мс} = \\ &= 0.1055994 \text{ мс} = \\ &= 105.5994 \text{ мкс} \end{aligned}$$

К сожалению, продолжая наши математические выкладки, приходим к выводу, что при таких раскладах погрешность составляет 608 миллисекунд в день, что равняется приблизительно 18.5 секунд в месяц, или почти 3.7 минут в год!

Можно предположить, что при использовании делителей другого типа ошибка может быть либо меньше, либо больше, в зависимости от погрешности округления. К счастью, ядро это знает и вводит соответствующие поправки.

Ключевой момент всей этой истории состоит в том, что независимо от красивого округленного значения, реальное значение выбирается в сторону ускорения отсчета.

## **Разрешающая способность отсчета времени**

Пусть отсчеты времени таймера генерируются чуть чаще, чем раз в 10 миллисекунд. Смогу ли я надежно обеспечить ожидание длительностью в 3



миллисекунды?

Не-а.

Подумайте, что происходит в ядре. Мы вызываем стандартную библиотечную функцию *delay()* для задержки на 3 миллисекунды. Ядро должно присвоить внутренней переменной ISR какое-то значение. Если оно присвоит ей значение текущего времени, то это будет означать, что таймер уже истек, и надо активизироваться немедленно. Если оно присвоит ей значение на один отсчет больше текущего времени, это будет означать, что надо активизироваться на следующем отсчете (т.е. с задержкой *вплоть до* 10 миллисекунд).

### ***Достижение более высокой точности***

Мораль: не следует рассчитывать на то, что разрешающая способность ваших таймеров будет лучше, чем у системного отсчета.

В QNX/Neutrino у приложений есть возможность программной подстройки аппаратного делителя и ядра вместе с ним (чтобы ядро знало, с какой частотой вызывается ISR таймера). Мы поговорим об этом далее в разделе «Опрос и установка часов реального времени».

### **Флуктуации отсчета времени**

Существует еще одно явление, которое вы должны принимать во внимание. Предположим, что разрешающая способность у вас равна 10 миллисекундам, а вы желаете сформировать задержку длительностью в 20 миллисекунд.

Всегда ли вы можете быть уверены, что от момента вызова функции *delay()* до возврата из нее пройдет ровно 20 миллисекунд?

Никогда.

На это есть две серьезные причины. Первая причина довольно проста: при блокировании поток изымается из очереди готовности. Это означает, что процессор может перейти к другому потоку вашего приоритета. Когда ваши 20 миллисекунд истекнут, ваш поток будет помещен в конец очереди готовности по этому приоритету и будет таким образом оставлен на милость потока, выполняющегося в данный момент. Это относится также к обработчикам прерываний и к потокам более высокого приоритета — то, что ваш поток перешел в состояние READY, еще не означает, что ему сразу предоставят процессор.

Вторая причина несколько более хитрая. Чтобы понять ее смысл, посмотрите на нижеприведенный рисунок.



### Флуктуации отсчета времени.

Проблема здесь состоит в том, что ваш запрос является асинхронным по отношению к источнику отсчетов. У вас нет никакой возможности синхронизировать аппаратный таймер с вашим запросом. Поэтому в итоге вы получите интервал задержки где-то в диапазоне от 20 до 30 мс — в зависимости от того, в какой момент между отсчетами аппаратных часов возник ваш запрос.

☞ Это очень важный момент. Флуктуации отсчета времени — одна из печальных жизненных реалий. Единственный способ избавиться от этой проблемы заключается в увеличении разрешающей способности так, чтобы получающиеся погрешности укладывались в пределы установленных допусков. (Как это делается, мы рассмотрим ниже, в разделе «Опрос и установка часов реального времени»). Имейте в виду, что флуктуации проявляются только на первом отсчете таймера — задержка на 100 секунд, реализуемая с помощью таймера с разрешением в 10 мс, попадет в интервал между 100 и 100.01 секундами.

## Типы таймеров

Таймер, работу которого мы только что обсудили, называют *относительным таймером*. Для такого таймера период ожидания задается относительно текущего времени. Если бы вы пожелали задержать выполнение вашего потока до 12 часов 4 минут 33 секунд EDT (Eastern Daylight Time — восточное поясное время — *прим. ред.*) 20 января 2005 года, вам пришлось бы сначала рассчитать точное число секунд от «сейчас» до выбранного вами момента и включить относительный таймер с задержкой на это число секунд. Поскольку это довольно часто

встречающаяся операция, в QNX/Neutrino реализованы абсолютные таймеры, которые обеспечивают задержку до заданного времени (а не на заданное время, как в случае относительного таймера).

А что если вы захотите сделать что-нибудь полезное, пока поток ожидает наступления установленной даты? Или делать что-либо и получать «синхроимпульс» каждые 27 секунд? Здесь нельзя просто так позволить себе спать!

Как мы уже обсуждали в главе «Процессы и потоки», вы можете просто запустить другой поток, и пусть он выполняет работу, пока ваш поток спит. Однако, поскольку мы говорим сейчас о таймерах, посмотрим, как это можно сделать другим способом.

В зависимости от выбранной цели, вы можете сделать это с помощью либо периодического, либо однократного таймера. *Периодический таймер* — это таймер, который срабатывает периодически, уведомляя поток (снова и снова), что истек некоторый временной интервал. *Однократный таймер* — это таймер, который срабатывает только один раз.

Реализация этих таймеров в ядре основана на том же самом принципе, что и в случае с таймером задержки из нашего первого примера. Ядро запоминает абсолютное значение времени (если вы укажете сделать именно так) и хранит его. Обработчик прерываний таймера сравнивает сохраненное значение времени с текущим.

Однако, вместо удаления из очереди на выполнение после системного вызова, на этот раз ваш поток продолжит работу. И в момент, когда суточное время достигнет заданного вами и хранимого в памяти момента времени, ядро уведомит ваш поток о том, что назначенное время пришло.

## Схема уведомления

Как получить уведомление о тайм-ауте? При использовании таймера задержки вы получаете уведомление просто посредством возвращения в состояние READY.

При использовании периодически и однократных таймеров у вас появляется выбор:

- послать импульс;
- послать сигнал;
- создать поток.

Импульсы мы уже обсуждали в главе «Обмен сообщениями»; сигналы — стандартный для UNIX механизм. Здесь же мы кратко рассмотрим уведомления при помощи создания потока.

## Как заполнять структуру *struct sigevent*

Независимо от выбранной вами схемы уведомления, вам обязательно придется заполнять структуру `struct sigevent`. Давайте вкратце посмотрим, как это делается.

```
struct sigevent {
    int sigev_notify;
    union {
        int  sigev_signo;
        int  sigev_coid;
        int  sigev_id;
        void (*sigev_notify_function)(union sigval);
    };
    union sigval sigev_value;
    union {
        struct {
            short sigev_code;
            short sigev_priority;
        };
        pthread_attr_t *sigev_notify_attributes;
    };
};
```

☞ Обратите внимание, что в приведенной декларации используются неименованные объединения и структуры. Внимательное изучение файла заголовка покажет вам, как этот трюк проходит с компиляторами, не поддерживающими такую особенность. По существу там есть директива `#define`, которая заставляет именованные объединения и структуры выглядеть неименованными. Подробнее см. `<sys/siginfo.h>`.

Первое поле, которое вы должны заполнить, — это элемент *sigev\_notify*, который определяет выбранный вами тип уведомления:

`SIGEV_PULSE`

Будет передан импульс.

`SIGEV_SIGNAL`, `SIGEV_` `SIGNAL` `_CODE` или

`SIGEV_SIGNAL_THREAD`

Будет передан сигнал.

`SIGEV_UNBLOCK`

В данном случае не используется; предназначен для тайм-аутов ядра (см. ниже в разделе «Тайм-ауты ядра»).

`SIGEV_INTR`

В данном случае не используется; предназначен для прерываний (см. главу «Прерывания»),

`SIGEV_THREAD`

Будет создан поток.

Поскольку мы намерены использовать структуру `struct sigevent` для таймеров, нас будут интересовать только такие значения `sigev_notify` как `SIGEV_PULSE`, `SIGEV_SIGNAL*` и `SIGNAL_THREAD`; остальные мы рассмотрим в соответствующих их применению разделах.

### ***Уведомление при помощи импульса***

Чтобы передать импульс при срабатывании таймера, присвойте полю `sigev_notify` значение `SIGEV_PULSE` и обеспечьте немного дополнительной информации:

<b>Поле</b>	<b>Значение и смысл</b>
<code>sigev_coid</code>	Идентификатор соединения (connection ID), по каналу которого которому будет передан импульс.
<code>sigev_value</code>	32-разрядное значение (данные импульса — см. параграф «Что внутри импульса?», глава «Обмен сообщениями» — <i>прим. ред.</i> ), которое будет передано по заданному полю <code>sigev_coid</code> соединению.
<code>sigev_code</code>	8-разрядное значение (код импульса — см. параграф «Что внутри импульса?», глава «Обмен сообщениями» — <i>прим. ред.</i> ), которое будет передано по заданному полю <code>sigev_coid</code> соединению.
<code>sigev_priority</code>	Приоритет доставки импульса. Нулевое значение не допускается — слишком уж много людей пострадало от переключения на нулевой приоритет после получения импульса, а поскольку на этом приоритете приходится конкурировать за процессор со спецпроцессом IDLE, много процессорного времени там точно не светит :-).

Отметим, что `sigev_coid` может описывать соединение на любом канале (обычно, хотя и не обязательно, этот канал связан с процессом, который инициирует событие).

### Уведомление при помощи сигнала

Чтобы передать сигнал, укажите в поле `sigev_notify` одно из нижеперечисленных значений:

`SIGEV_SIGNAL`

Процессу будет передан обычный сигнал.

`SIGEV_SIGNAL_CODE`

Процессу будет передан сигнал, содержащий 8-битный код.

`SIGEV_SIGNAL_THREAD`

Сигнал, содержащий 8-битный код, будет передан определенному потоку.

При выборе уведомления типа `SIGEV_SIGNAL*` нужно будет заполнить ряд дополнительных полей:

Поле	Значение и смысл
------	------------------

<code>sigev_signo</code>	Номер сигнала для передачи (берется из <code>&lt;signal.h&gt;</code> , например, <code>SIGALRM</code> ).
--------------------------	--

<code>sigev_code</code>	8-разрядный код (для уведомления типа <code>SIGEV_SIGNAL_CODE</code> или <code>SIGEV_SIGNAL_THREAD</code> ).
-------------------------	--

### Уведомление созданием потока

Для создания потока по срабатыванию таймера установите поле `sigev_notify` в значение `SIGEV_THREAD` и заполните следующие поля:

Поле	Значение и смысл
------	------------------

<code>sigev_notify_function</code>	Адрес функции, возвращающей <code>void*</code> и принимающей <code>void*</code> , которая будет вызвана при возникновении события.
------------------------------------	--

<code>sigev_value</code>	Значение, которое будет передано функции <code>sigev_notify_function()</code> в качестве параметра.
--------------------------	---

<code>sigev_notify_attributes</code>	Атрибутная запись потока (см. главу «Процессы и потоки», параграф «Атрибутная запись потока»).
--------------------------------------	--

Этот тип уведомления воистину страшен. Если ваш таймер будет срабатывать слишком часто, и при этом будут готовы к выполнению потоки с более высоким приоритетом, чем вновь создаваемые, то у вас быстро вырастет огромная куча заблокированных потоков, и они съедят все ресурсы вашей машины. Пользуйтесь этим типом уведомления с осторожностью!

## Общие приемы программирования уведомлений

В файле `<sys/signinfo.h>` есть ряд удобных макросов упрощения заполнения полей в структурах:

*SIGEV\_SIGNAL\_INIT(eventp, signo)*

Установите *eventp* в *SIGEV\_SIGNAL* и впишите соответствующий номер сигнала *signo*.

*SIGEV\_SIGNAL\_CODE\_INIT(eventp, signo, value, code)*

Установите поле *eventp* в *SIGEV\_SIGNAL\_CODE*, укажите номер сигнала в *signo*, а также задайте значения полей *value* и *code*.

*SIGEV\_SIGNAL\_THREAD\_INIT(eventp, signo, value, code)*

Установите *eventp* в *SIGEV\_SIGNAL\_THREAD*, укажите номер сигнала в *signo*, а также задайте значения полей *value* и *code*.

*SIGEV\_PULSE\_INIT(eventp, coid, priority, code, value)*

Установите *eventp* в *SIGEV\_SIGNAL\_PULSE*, укажите идентификатор соединения в *coid*, а также параметры *priority*, *code* и *value*. Отметьте, что для *priority* есть специальное значение *SIGEV\_PULSE\_PRIO\_INHERIT*, которое предотвращает изменение приоритета принимающего потока.

*SIGEV\_UNBLOCK\_INIT(eventp)*

Установите *eventp* в *SIGEV\_UNBLOCK*.

*SIGEV\_INTR\_INIT(eventp)*

Установите *eventp* в *SIGEV\_INTR*.

*SIGEV\_THREAD\_INIT(eventp, func, attributes)*

Задайте значения *eventp*, функции потока *func* и атрибутной записи *attributes*.

## Уведомление при помощи импульса

Предположим, что вы разрабатываете сервер, который будет обречен провести большую часть своей жизни в *RECEIVE*-блокированном состоянии, ожидая сообщение. Идеальным вариантом здесь было бы принять специальное сообщение, указывающее, что момент, которого мы так долго ждали, наконец настал.

Как раз при таком сценарии и надо использовать импульсы в качестве схемы уведомления. В разделе «Применение таймеров», представленном ниже, я приведу пример кода, который можно использовать для периодического получения импульсов.

Предположим, что, с другой стороны, вы выполняете некоторую работу, но не желаете, чтобы она продолжалась вечно. Например, вы ожидаете возврата из некоторой функции, но не можете точно предсказать, сколько времени на это потребуется.

В этом случае оправданным выбором является использование уведомления при помощи сигнала — возможно, даже с обработчиком. (Другой вариант, который мы обсудим позже, заключается в использовании тайм-аутов ядра; см. также параграф «\_NTO\_CHF\_UNBLOCK» в главе «Обмен сообщениями»). В параграфе «Применение таймеров», представленном ниже, мы рассмотрим пример, использующий сигналы.

Если вы вообще не собираетесь принимать сообщения, то использование сигнала и функции *sigwait()* является более экономной альтернативой созданию канала для принятия импульсного сообщения.



## Применение таймеров

Изучив все красоты теории, давайте теперь переключим наше внимание на конкретные образцы кода, чтобы посмотреть, что можно сделать при помощи таймеров.

Чтобы работать с таймером, вам потребуется:

1. Создать объект типа «таймер».
2. Выбрать схему уведомления (сигнал, импульс или создание потока) и создать структуру уведомления (**struct sigevent**).
3. Выбрать нужный тип таймера (относительный или абсолютный, и однократный или периодический).
4. Запустить таймер.

Давайте теперь рассмотрим все это по порядку.

## Создание таймера

Первый этап — это создание таймера с помощью функции *timer\_create()*:

```
#include <time.h>
#include <sys/siginfo.h>

int timer_create(clockid_t clock_id,
    struct sigevent *event, timer_t *timerid);
```

Аргумент *clock\_id* сообщает функции *timer\_create()*, на какой временном базисе вы формируете таймер. Это вещь из области POSIX — стандарт утверждает, что на различных платформах вы можете использовать различные типы временных базисов, но любая платформа должна, по меньшей мере, поддерживать базис `CLOCK_REALTIME`. В QNX/Neutrino есть три базиса:

- `CLOCK_REALTIME`
- `CLOCK_SOFTTIME`
- `CLOCK_MONOTONIC`

## Сигнал, импульс или поток?

Оставим пока на время варианты `CLOCK_SOFTTIME` и `CLOCK_MONOTONIC`, поскольку они еще пока (на момент написания

книги — *прим. ред.*) не реализованы. Вторым параметром является указатель на структуру `struct sigevent`. Эта структура применяется для того, чтобы сообщить ядру о типе события, которое таймер должен сгенерировать при срабатывании. Мы уже обсуждали порядок заполнения `struct sigevent`, когда говорили о выборе схемы уведомления.

Итак, мы вызываем функцию `timer_create()` с временным базисом `CLOCK_REALTIME` и указателем на структуру `struct sigevent`, и ядро создает объект типа «таймер» (он возвращается в последнем аргументе). Этот объект представляет собой небольшое целое число, которое является номером таймера в таблице таймеров ядра. Считайте его просто «дескриптором».

На этот момент никаких событий пока не происходит. Вы просто создали таймер, но ведь вы еще не включали его.

## Какой таймер выбрать?

Создав таймер, теперь вы должны решить, какого типа будет этот таймер. Это осуществляется путем комбинирования аргументов функции `timer_settime()`, которая обычно применяется для собственно запуска таймера:

```
#include <time.h>
```

```
int timer_settime(timer_t timerid, int flags,  
    struct itimerspec *value, struct itimerspec *oldvalue);
```

Аргумент *timerid* — это число, которое вы получите обратно по вызову функции `timer_create()`. Вы можете создать множество таймеров, а затем вызывать `timer_settime()` для них по отдельности, когда вам это будет необходимо.

С помощью аргумента *flags* вы определяете тип таймера — абсолютный или относительный.

Если вы передаете константу `TIMER_ABSTIME`, получается абсолютный таймер, как вы и могли бы предположить. Затем вы передаете реальные дату и время срабатывания таймера.

Если вы передаете нуль, таймер предполагается относительным.

Давайте посмотрим, как определяется время. Вот ключевые фрагменты двух структур данных из `<time.h>`:

```
struct timespec {  
    long tv_sec, tv_nsec;  
};
```

```
struct itimerspec {  
    struct timespec it_value, it_interval;  
};
```

В структуре **struct itimerspec** есть два поля:

*it\_value* — однократно используемое значение

*it\_interval* — перезагружаемое значение

Параметр *it\_value* задает либо интервал времени от настоящего момента до момента срабатывания таймера (в случае относительного таймера), либо собственно время срабатывания (в случае абсолютного таймера). После срабатывания таймера значение величины *it\_interval* задает относительное время для повторной загрузки таймера, чтобы он мог сработать снова. Заметим, что задание для *it\_interval* нулевого значения преобразует данный таймер в однократный. Вы можете предположить, что чтобы создать «исключительно периодический» таймер, вам следует установить параметр *it\_interval* в значение интервала перезагрузки, а параметр *it\_value* — в нуль. К сожалению, последнее неверно — установка параметра *it\_value* в нуль выключает таймер. Если вы хотите создать «исключительно периодический» таймер, присвойте *it\_value* и *it\_interval* одинаковые значения и создайте таймер как относительный. Такой таймер сработает один раз (с задержкой *it\_value*), а затем будет циклически перезагружаться с задержкой *it\_interval*.

Оба параметра *it\_value* и *it\_interval* фактически являются структурами типа **struct timespec** — еще одного POSIX-объекта. Эта структура позволяет вам обеспечить разрешающую способность на уровне долей секунд. Первый ее элемент, *tv\_sec*, — это число секунд, второй элемент, *tv\_nsec*, — число наносекунд в текущей секунде. (Это означает, что никогда не следует устанавливать параметр *tv\_nsec* в значение, превышающее 1 миллиард — это будет подразумевать смещение на более чем 1 секунду).

Несколько примеров:

```
it_value.tv_sec = 5;  
it_value.tv_nsec = 500000000;  
it_interval.tv_sec = 0;  
it_interval.tv_nsec = 0;
```

Это сформирует однократный таймер, который сработает через 5,5 секунды. (5,5 секунд складывается из 5 секунд и 500,000,000 наносекунд.)

Мы предполагаем здесь, что этот таймер используется как относительный, потому что если бы это было не так, то его время

срабатывания уже давно бы его истекло (5.5 секунд с момента 00:00 по Гринвичу, 1 января 1970).

Другой пример:

```
it_value.tv_sec = 987654321;
it_value.tv_nsec = 0;
it_interval.tv_sec = 0;
it_interval.tv_nsec = 0;
```

Данная комбинация параметров сформирует однократный таймер, который сработает в четверг, 19 апреля 2001 года в 00:25:21 по EDT. (Существует множество функций, которые помогут вам преобразовать воспринимаемый человеком интервал времени в «число секунд, истекшее с 00:00:00 по Гринвичу, 1 января 1970 года». См. функции *time()*, *asctime()*, *ctime()*, *mktime()*, *strftime()*, и т.д.).

В данном примере мы предполагаем, что это абсолютный таймер, поскольку в противном случае ждать пришлось бы достаточно долго (987654321 секунд — приблизительно 31.3 года).

Отметьте, что в двух приведенных выше примерах я говорил: «мы предполагаем». В коде функции *timer\_settime()* нет никаких проверок на правильность аргументов! Вы должны самостоятельно доопределить, является таймер абсолютным или относительным. Что до ядра, то оно будет просто счастливо запланировать какое-нибудь событие на 31.3 года вперед.

И еще один пример:

```
it_value.tv_sec = 1;
it_value.tv_nsec = 0;
it_interval.tv_sec = 0;
it_interval.tv_nsec = 500000000;
```

Если предположить, что это относительный таймер, он сработает через одну секунду и далее каждые полсекунды. Не существует никаких требований какого бы то ни было подобия значений интервала перезагрузки значениям задержки однократного срабатывания.

## Сервер с периодическими импульсами

Первое, что следует рассмотреть, — это сервер, который желает получать периодические сообщения. Типовыми применениями такой схемы являются:

- поддерживаемые сервером тайм-ауты клиентских запросов;
- внутренние периодические события серверов.

Конечно, есть и другие, специализированные, применения для таких вещей — например, периодические подтверждения готовности узлов сети («я жив»), запросы на повторную передачу, и т.п.

### *Поддерживаемые сервером тайм-ауты*

В таком сценарии сервер предоставляет клиенту некоторую услугу, и клиент способен задать тайм-аут. Это может использоваться в самых разнообразных приложениях. Например, вы можете сказать серверу «выдай мне данные за 15 секунд» или «дай мне знать, когда истекнут 10 секунд», или «жди прихода данных, но в течение не более чем 2 секунд».

Все это — примеры поддерживаемых сервером тайм-аутов. Клиент посылает сообщение серверу и блокируется. Сервер принимает периодические сообщения от таймера (раз в секунду, реже или чаще) и подсчитывает, сколько этих сообщений он получил. Когда число сообщений о тайм-аутах превышает время ожидания, указанное клиентом, сервер отвечает клиенту с сообщением о тайм-ауте или, возможно, с данными, которые он успел накопить на данный момент — это зависит от того, как структурированы отношения клиента и сервера.

Ниже приведен полный пример сервера, который принимает одно из двух сообщений от клиентуры и сообщения о тайм-ауте в виде импульса. Первое клиентское сообщение говорит серверу: «Дай мне знать, есть ли для меня данные, но не блокируй меня более чем на 5 секунд». Второе клиентское сообщение говорит: «Вот, возьми данные». Сервер должен позволить нескольким клиентам блокироваться на себе в ожидании данных, и поэтому обязан сопоставить клиентам тайм-ауты. Тут-то и нужен импульс; он информирует сервер: «Истекла одна секунда».

Чтобы программа не выглядела излишне громоздкой, перед каждым из основных разделов я прерываю исходный текст небольшими пояснениями. Скачать эту программу вы можете на FTP-сайте компании PARSE ([ftp://ftp.parseftp.parse.com/pub/book\\_v3.tar.gz](ftp://ftp.parseftp.parse.com/pub/book_v3.tar.gz)), файл называется `time1.c`.

### *Декларации*

В первом разделе программы определяются различные именованные константы и структуры данных. В нем также подключаются все необходимые заголовочные файлы. Оставим это без комментариев. :-)

```

/*
 * timer1.c
 *
 * Пример сервера, получающего периодические сообщения
 * от таймера
 * и обычные сообщения от клиента.
 *
 * Иллюстрирует использование функций таймера с импульсами.
 */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <signal.h>
#include <errno.h>
#include <unistd.h>
#include <sys/signinfo.h>
#include <sys/neutrino.h>

// Получаемые сообщения

// Сообщения
#define MT_WAIT_DATA 2 // Сообщение от клиента
#define MT_SEND_DATA 3 // Сообщение от клиента

// Импульсы
#define CODE_TIMER 1 // Импульс от таймера

// Отправляемые сообщения
#define MT_OK 0 // Сообщение клиенту
#define MT_TIMEDOUT 1 // Сообщение клиенту

// Структура сообщения
typedef struct {
    int messageType; // Содержит сообщение от клиента и
                    // клиенту
    int messageData; // Опциональные данные, зависят от
                    // сообщения
} ClientMessageT;

typedef union {

```

```

ClientMessageT msg; // Сообщение может быть
                    // либо обычным,
struct _pulse pulse; // либо импульсом
} MessageT;

// Таблица клиентов
#define MAX_CLIENT 16 // Максимум клиентов
                    // одновременно

struct {
    int in_use; // Элемент используется?
    int rcvid;  // Идентификатор
                // отправителя клиента
    int timeout; // Оставшийся клиенту
                // тайм-аут
} clients[MAX_CLIENT]; // Таблица клиентов

int chid; // Идентификатор канала
          // (глобальный)

int debug = 1; // Режим отладки, 1 ==
               // вкл, 0 == выкл

char *programe = "time1.c";

// Предопределенные прототипы
static void setupPulseAndTimer(void);
static void gotAPulse(void);
static void gotAMessage(int rcvid, ClientMessageT *msg);

```

### ***main()***

Следующий раздел кода является основным и отвечает за следующее:

- создание канала с помощью функции *ChannelCreate()*;
- вызов подпрограммы *setupPulseAndTimer()* (для настройки периодического таймера, срабатывающего раз в секунду и использующего импульс в качестве способа доставки события;
- и, наконец, бесконечный цикл ожидания импульсов и сообщений и их обработки.

Обратите внимание на проверку значения, возвращаемого *MsgReceive()* — нуль указывает, что был принят импульс (здесь мы не делаем никакой дополнительной проверки, *наш* ли это импульс), ненулевое значение говорит о том, что было принято сообщение.

Обработка импульсов и сообщений выполняется функциями *gotAPulse()* и *gotAMessage()*.

```
int main(void) // Игнорировать аргументы
               // командной строки
{
    int rcvid; // PID отправителя
    MessageT msg; // Само сообщение
    if ((chid = ChannelCreate(0)) == -1) {
        fprintf(stderr, "%s: не удалось создать канал!\n",
            progname);
        perror(NULL);
        exit(EXIT_FAILURE);
    }
    // Настроить импульс и таймер
    setupPulseAndTimer();
    // Прием сообщений
    for(;;) {
        rcvid = MsgReceive(chid, &msg, sizeof(msg), NULL);
        // Определить, от кого сообщение
        if (rcvid == 0) {
            // Здесь неплохо бы еще проверить поле «code»...
            gotAPulse();
        } else {
            gotAMessage(rcvid, &msg.msg);
        }
    }
    // Сюда мы никогда не доберемся
    return (EXIT_SUCCESS);
}
```

### *setupPulseAndTimer()*

В функции *setupPulseAndTimer()* вы видите код, в котором определяется тип таймера и схема уведомления. Когда мы рассуждали о таймерных функциях выше, я говорил, что таймер может выдать сигнал



или импульс, либо создать поток. Решение об этом принимается именно здесь, в функции *setupPulseAndTimer()*. Обратите внимание, что здесь мы использовали макроопределение `SIGEV_PULSE_INIT()`. Используя это макроопределение, мы реально присвоили элементу `sigev_notify` значение `SIGEV_PULSE`. (Если бы мы использовали одно из макроопределений семейства `SIGEV_SIGNAL*_INIT()`, мы получили бы уведомление при помощи соответствующего сигнала). Отметим, что при настройке импульса мы с помощью вызова *ConnectAttach()* устанавливаем соединение с самим собой и даем ему уникальный код (здесь — константа `CODE_TIMER`; мы ее определили сами)

Последний параметр в инициализации структуры события — это приоритет импульса; здесь мы выбрали `SIGEV_PULSE_PRIO_INHERIT` (константа, равная -1). Это предписывает ядру не изменять приоритет принимающего импульс потока.

В конце описания функции мы вызываем *timer\_create()* для создания таймера в ядре, после чего настраиваем его на срабатывание через одну секунду (поле *it\_value*) и на периодическую перезагрузку односекундными интервалами (поле *it\_interval*). Отметим, что таймер включается только по вызову *timer\_settime()*, а не при его создании.

☞ Схема уведомления по типу `SIGEV_PULSE` — расширение, свойственное только QNX/Neutrino. Концепция импульсов в POSIX отсутствует.

```
/*
 * setupPulseAndTimer
 *
 * Эта подпрограмма отвечает за настройку импульса, чтобы
 * тот отправлял сообщение с кодом MT_TIMER.
 * Затем устанавливается
 * периодический таймер с периодом в одну секунду.
 */
void setupPulseAndTimer(void) {
    timer_t timerid; // Идентификатор таймера
    struct sigevent event; // Генерируемое событие
    struct itimerspec timer; // Структура данных
                                // таймера

    int coid; // Будем соединяться с
              // собой
    // Создать канал к себе
```

```

coid = ConnectAttach(0, 0, chid, 0, 0);
if (coid == -1) {
    fprintf(stderr, "%s: ошибка ConnectAttach!\n", progname);
    perror(NULL);
    exit(EXIT_FAILURE);
}
// Установить, какое событие мы хотим сгенерировать
// - импульс
SIGEV_PULSE_INIT(&event, coid, SIGEV_PULSE_PRIO_INHERIT,
    CODE_TIMER, 0);
// Создать таймер и привязать к событию
if (timer_create(CLOCK_REALTIME, &event, &timerid) ==
    -1) {
    fprintf(stderr,
        "%s: не удалось создать таймер, errno %d\n",
        progname, errno);
    perror(NULL);
    exit(EXIT_FAILURE);
}
// Настроить таймер (задержка 1 с, перезагрузка через
// 1 с) ...
timer.it_value.tv_sec = 1;
timer.it_value.tv_nsec = 0;
timer.it_interval.tv_sec = 1;
timer.it_interval.tv_nsec = 0;
// ...и запустить его!
timer_settime(timerid, 0, &timer, NULL);
}

```

### ***gotAPulse()***

В функции *gotAPulse()* вы можете видеть, как мы реализовали способность сервера обеспечивать тайм-ауты для клиентов. Мы последовательно просматриваем список клиентуры и, поскольку мы знаем, что импульс выдается один раз в секунду, просто уменьшаем число секунд, которое остается клиенту до тайм-аута. Если эта величина достигает нулевого значения, мы отвечаем этому клиенту сообщением «Извините, тайм-аут» (тип сообщения MT\_TIMEDOUT). Обратите внимание, что мы подготавливаем это сообщение заранее (вне цикла **for**), а затем посылаем

его по мере необходимости. Этот прием — по существу вопрос стиля: если вы предполагаете отвечать часто, возможно, имело бы смысл выполнить настройку однажды и загодя. Если же множество ответов не ожидается, то имело бы больший смысл делать настройки по мере необходимости.

Если значение оставшегося времени еще не достигло нуля, мы не делаем ничего — клиент по-прежнему заблокирован в ожидании сообщения.

```
/*
 * gotAPulse
 *
 * Эта подпрограмма отвечает за обработку тайм-аутов.
 * Она проверяет список клиентов на предмет тайм-аута
 * и отвечает соответствующим сообщением тем клиентам,
 * у которых тайм-аут произошел.
 */
void gotAPulse(void) {
    ClientMessageT msg;
    int i;
    if (debug) {
        time_t now;
        time(&now);
        printf("Получен импульс, время %s", ctime(&now));
    }
    // Подготовить ответное сообщение
    msg.messageType = MT_TIMEDOUT;
    // Просмотреть список клиентов
    for (i = 0; i < MAX_CLIENT; i++) {
        // Элемент используется?
        if (clients[i].in_use) {
            // Тайм-аут?
            if (--clients[i].timeout == 0) {
                // Ответить
                MsgReply(clients[i].rcvid, EOK, &msg, sizeof(msg));
                // Освободить элемент
                clients[i].in_use = 0;
            }
        }
    }
}
```

## *gotAMessage()*

В функции *gotAMessage()* вы видите другую половину заданной функциональности, где мы добавляем клиента в список клиентуры, ожидающей данные (если получено сообщение типа MT\_WAIT\_DATA), или сопоставляем клиента с сообщением, которое было только что получено (если это сообщение типа MT\_SEND\_DATA). Заметьте, что для простоты мы здесь не реализуем очередь клиентов, находящихся в ожидании передачи данных, получатель для которых еще не доступен — это вопрос управления очередями, оставьте его для себя в качестве упражнения.

```
/*
 * gotAMessage
 *
 * Эта подпрограмма вызывается при каждом приеме
 * сообщения. Проверяем тип
 * сообщения (либо «жду данных», либо «вот данные»),
 * и действуем
 * соответственно. Для простоты предположим, что данные
 * никогда не ждут.
 * Более подробно об этом см. в тексте.
 */
void gotAMessage(int rcvid, ClientMessageT *msg) {
    int i;
    // Определить тип сообщения
    switch (msg->messageType) {
    // Клиент хочет ждать данных
    case MT_WAIT_DATA:
        // Посмотрим, есть ли пустое место в таблице клиентов
        for (i = 0; i < MAX_CLIENT; i++) {
            if (!clients[i].in_use) {
                // Нашли место - пометить как занятое,
                // сохранить rcvid
                // и установить тайм-аут
                clients[i].in_use = 1;
                clients[i].rcvid = rcvid;
                clients[i].timeout = 5;
                return;
            }
        }
    }
```

```

    }
    fprintf(stderr,
        "Таблица переполнена, сообщение от rcvid %d"
        " игнорировано, клиент заблокирован\n", rcvid);
    break;
    // Клиент с данными
case MT_SEND_DATA:
    // Посмотрим, есть ли другой клиент, которому можно ответить
    // данными от этого клиента
    for (i = 0; i < MAX_CLIENT; i++) {
        if (clients[i].in_use) {
            // Нашли - использовать полученное сообщение
            // в качестве ответного
            msg->messageType = MT_OK;
            // Ответить ОБОИМ КЛИЕНТАМ!
            MsgReply(clients[i].rcvid, EOK, msg, sizeof(*msg));
            MsgReply(rcvid, EOK, msg, sizeof(*msg));
            clients[i].in_use = 0;
            return;
        }
    }
    fprintf(stderr,
        "Таблица пуста, сообщение от rcvid %d игнорировано,"
        " клиент заблокирован\n", rcvid);
    break;
}
}

```

### ***Примечания***

Несколько общих замечаний по тексту программы:

- Если сообщение с данными прибывает, когда либо никто не ждет, либо список ожидающих клиентов переполнен, в стандартный поток ошибок выводится сообщение, но клиенту при этом мы не отвечаем ничего. Это означает, что ряд клиентов может оказаться в REPLY-блокированном состоянии навсегда — идентификаторы отправителей мы потеряли, а значит, и ответ им дать не можем.

Это сделано намеренно. Вы можете изменить это, добавив соответственно сообщения MT\_NO\_WAITERS и MT\_NO\_SPACE,

которыми можно было бы отвечать всякий раз при обнаружении ошибок данного типа.

- Когда клиент-обработчик ждет, а клиент-поставщик пересылает ему данные, мы отвечаем обоим клиентам. Это критично, поскольку мы должны разблокировать обоих клиентов.

- Мы повторно использовали буфер клиента-поставщика для обоих ответов. Этот прием программирования — опять же, вопрос стиля: в большом приложении у вас, вероятно, было бы много типов возвращаемых значений, и вы могли бы и не захотеть повторно использовать одни и те же буферы.

- В приведенном примере используется «щербатый» массив фиксированной длины с флагом «элемент задействован» (`clients[i].in_use`). Поскольку моей целью здесь является отнюдь не демонстрация хитростей программирования односвязных списков, я использовал простейший для понимания вариант. В конечном же программном продукте, разумеется, имело бы смысл использовать динамический список.

- Когда функция *MsgReceive()* получает импульс, наше решение относительно того, действительно ли это «наш» импульс, фактически является весьма слабо аргументированным — мы просто предполагаем (согласно комментариям), что все входящие импульсы имеют тип `CODE_TIMER`. Опять же, в конечном продукте следовало бы проверять значение кода импульса и сообщать о наличии каких-либо аномалий.

Отметим, что в приведенном примере демонстрируется только один способ реализации тайм-аутов клиентуры. Позже, в этой же главе (в разделе «Тайм-ауты ядра») мы поговорим о тайм-аутах ядра. Это еще один способ делать почти то же самое, только управление на этот раз осуществляется клиентом, а не таймером.

### ***Внутренние периодические события серверов***

Здесь мы имеем несколько другое применение для периодических сообщений о тайм-аутах, когда эти сообщения предназначены сервером исключительно для внутреннего использования и не имеют никакого отношения к клиенту вообще.

Например, некоторые аппаратные средства могут требовать, чтобы сервер опрашивал их периодически — например, такое может быть в случае сетевого соединения: сервер должен периодически проверять,

является ли данное подключение доступным, и это не зависит от команд клиентуры.

Другой вариант — если, например, в аппаратных средствах предусмотрен таймер «выключения по неактивности». Например, если длительное пребывание какого-то аппаратного модуля во включенном состоянии может приводить к неоправданным затратам электроэнергии, то если его никто не использует в течение, скажем, 10 секунд, его можно было бы выключить (или переключить в режим низкого энергопотребления — *прим. ред.*). Опять же, к клиенту это не имеет никакого отношения (за исключением того, что запрос от клиента отменит режим ожидания) — это просто функция, которую сервер должен уметь предоставлять «своим» аппаратным средствам.

Код в этом случае сильно бы напоминал приведенный выше пример, за исключением того, что вместо списка ожидающих клиентов у вас была бы только одна переменная тайм-аута. С каждым событием от таймера ее значение уменьшалось бы, но пока оно больше нуля, ничего бы не происходило. Когда оно стало бы равным нулю, это вызвало бы отключение аппаратных средств (или какое-либо другое соответствующее действие).

Единственный трюк здесь заключается в том, что всякий раз, когда поступает сообщение от клиента, использующего данные аппаратные средства, вы должны восстановить первоначальное значение этой переменной, поскольку обращение к ресурсу должно сбрасывать «обратный отсчет». И наоборот, аппаратным средствам может потребоваться определенный промежуток времени «на разогрев» после включения. В этом случае после выключения аппаратных средств вам придется при поступлении запроса от клиента организовать еще один таймер, чтобы «придерживать» запрос до того момента, пока аппаратные средства не станут готовы.

## Таймеры, посылающие сигналы

На настоящий момент мы уже рассмотрели практически все, что относится к таймерам, за исключением одного небольшого момента. Мы обеспечивали отправку импульса, но у нас также есть возможность посылать POSIX-сигналы. Давайте посмотрим, как это делается:

```
timer_create(CLOCK_REALTIME, NULL, &timerid);
```

Это простейший способ создать таймер, который будет посылать вам сигнал. Он обеспечивает выдачу сигнала SIGALRM при срабатывании

таймера. Если бы мы предоставили `struct sigevent`, мы могли бы определить, какой именно сигнал мы хотим получить:

```
struct sigevent event;  
SIGEV_SIGNAL_INIT(&event, SIGUSR1);  
timer_create(CLOCK_REALTIME, &event, &timerid);
```

Это обеспечит нам выдачу сигнала SIGUSR1 вместо SIGALRM.

Сигналы таймера перехватываются обычными обработчиками сигналов, здесь нет ничего необычного.

## Таймеры, создающие потоки

Если вы хотите по каждому срабатыванию таймера создавать новый поток, то вы можете это сделать с помощью `struct sigevent` и всех остальных таймерных штук, которые мы только что обсудили:

```
struct sigevent event;  
SIGEV_THREAD_INIT(&event, maintenance_func, NULL);
```

Однако, пользоваться этим надо очень осторожно, потому что если вы определите слишком короткий интервал, вы можете просто утонуть в создаваемых потоках. Они просто поглотят все ресурсы вашего процессора и оперативной памяти.

## Опрос и установка часов реального времени, и кое-что еще

Независимо от применения таймеров, вы можете также опрашивать и устанавливать часы реального времени, а также и плавно подстраивать их. Для этих целей можно использовать следующие функции:

Функция	Тип	Описание
<i>ClockAdjust()</i>	QNX/Neutrino	Плавная регулировка времени
<i>ClockCycles()</i>	QNX/Neutrino	Опрос с высоким разрешением
<i>clock_getres()</i>	POSIX	Выборка базового разрешения
<i>clock_gettime()</i>	POSIX	Получение текущего времени суток
<i>ClockPeriod()</i>	QNX/Neutrino	Получение/установка базового разрешения
<i>clock_settime()</i>	POSIX	Установка текущего времени суток
<i>ClockTime()</i>	QNX/Neutrino	Получение/установка текущего времени суток

## Опрос и установка



Функции *clock\_gettime()* и *clock\_settime()* являются POSIX-функциями, основанными на системном вызове *ClockTime()*. Эти функции могут применяться для получения и установки текущего времени суток. К сожалению, установка здесь является «жесткой», то есть независимо от того, какое время вы указываете в буфере, оно немедленно делается текущим. Это может иметь пугающие последствия, особенно когда получается, что время «повернуло вспять», потому что устанавливаемое время оказалось меньше «реального». Вообще настройка часов таким способом должна выполняться только при включении питания или когда время сильно не соответствует «реальному».

Если нужна плавная корректировка текущего времени, ее можно реализовать с помощью функции *ClockAdjust()*:

```
int ClockAdjust(clockid_t id,  
    const struct _clockadjst *new,  
    const struct _clockadjst *old);
```

Параметрами здесь являются источник синхроимпульсов (всегда используйте *CLOCK\_REALTIME*) и параметры *new* и *old*. Оба эти параметра являются необязательными и могут быть заданы как *NULL*. Параметр *old* просто возвращает текущую корректировку. Работа по корректировке часов управляется параметром *new*, который является указателем на структуру, содержащую два элемента, *tick\_nsec\_inc* и *tick\_count*. Действует функция *ClockAdjust()* очень просто — каждые *tick\_count* отсчетов системных часов к существующему значению системного времени добавляется корректировка *tick\_nsec\_inc*. Это означает, что чтобы передвинуть время вперед («догоняя» реальное), вы задаете для *tick\_nsec\_inc* положительное значение. Заметьте, что не надо переводить время назад — вместо этого, если ваши часы спешат, задайте для *tick\_nsec\_inc* небольшое отрицательное значение, и ваши часы соответственно замедлят ход. Таким образом, вы немного замедляете часы, пока их показания не будут соответствовать действительности. Существует эмпирическое правило, гласящее, что не следует корректировать системные часы значением, превышающим 10% от базового разрешения вашей системы (см. функцию *ClockPeriod()* и ее друзей, о них мы поговорим в следующем параграфе).

### ***Регулировка разрешающей способности***

Как мы и говорили на протяжении всей этой главы, нельзя сделать ничего с большей точностью, чем принятая в системе базовая разрешающая

способность по времени. Напрашивается вопрос: а как настроить эту базовую разрешающую способность? Для этого вы можете использовать следующую функцию:

```
int ClockPeriod(clockid_t id,  
    const struct _clockperiod *new,  
    struct _clockperiod *old, int reserved);
```

Как и в случае с описанной выше функцией *ClockAdjust()*, с помощью параметров *new* и *old* вы получаете и/или устанавливаете значения базовой разрешающей способности по времени. Параметры *new* и *old* являются указателями на структуры типа **struct \_clockperiod**, которые, в свою очередь, содержат два элемента — *nsec* и *fract*. На настоящий момент элемент *fract* должен быть равен нулю (это число фемтосекунд (миллиардная доля микросекунды — *прим. ред.*); нам, вероятно, это еще не скоро потребуется). Параметр *nsec* указывает, сколько наносекунд содержится в интервале между двумя базовыми отсчетами времени. Значение этого интервала времени по умолчанию — 10 миллисекунд, поэтому значение *nsec* (если вы используете функцию для получения базового разрешения) будет приблизительно равно 10 миллионам наносекунд. (Как мы уже упоминали ранее в разделе «Источники прерываний таймера», это не будет в точности равняться 10 миллисекундам.)

При этом вы можете, конечно, не стесняться и попробовать назначить базовой разрешающей способности какое-нибудь смехотворно малое значение, но тут вмешается ядро и эту вашу попытку пресечет. В общем случае, в большинстве систем допускаются значения от 1 миллисекунды до сотен микросекунд.

### ***Точные временные метки***

Существует одна система отсчета времени, которая не подчиняется описанным выше правилам «базовой разрешающей способности по времени». Некоторые процессоры оборудованы встроенным высокочастотным (высокоточным) счетчиком, к которому QNX/Neutrino обеспечивает доступ при помощи функции *ClockCycles()*. Например, в процессоре Pentium, работающем с частотой 200 МГц, этот счетчик увеличивается тоже с частотой в 200 МГц, и поэтому он может обеспечить вам значение времени с точностью до 5 наносекунд. Это особенно полезно, когда вы хотите точно выяснить, сколько времени затрачивается на выполнение конкретного фрагмента кода (в предположении, конечно, что

он не будет вытеснен). В этом случае вы должны вызвать функцию *ClockCycles()* перед началом вашего фрагмента и после его окончания, а потом просто подсчитать разность полученных отсчетов. Более подробно это описано в руководстве по Си-библиотеке.

## Тайм-ауты ядра

QNX/Neutrino позволяет вам получать тайм-ауты по всем заблокированным состояниям. Мы обсуждали эти состояния в главе «Процессы и потоки» в разделе «Состояния потоков». Наиболее часто у вас может возникнуть потребность в этом при обмене сообщениями: клиент, посылая сообщение серверу, не желает ждать ответа «вечно». В этом случае было бы удобно использовать тайм-аут ядра. Тайм-ауты ядра также полезны в сочетании с функцией *pthread\_join()*: завершения потока тоже не всегда хочется долго ждать.

Ниже приводится декларация для функции *TimerTimeout()*, которая является системным вызовом, ответственным за формирование тайм-аутов ядра.

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

int TimerTimeout(clockid_t id, int flags,
    const struct sigevent *notify,
    const uint64_t *ntime, uint64_t *otime);
```

Видно, что функция *TimerTimeout()* возвращает целое число (индикатор удачи/неудачи; 0 означает, что все в порядке, -1 — что произошла ошибка, и ее код записан в *errno*). Источник синхроимпульсов (CLOCK\_REALTIME, и т.п.) указывается в *id*, параметр *flags* задает соответствующее состояние (или состояния). Параметр *notify* всегда должен быть событием уведомления типа SIGEV\_UNBLOCK; параметр *ntime* указывает относительное время, спустя которое ядро должно сгенерировать тайм-аут. Параметр *otime* показывает предыдущее значение тайм-аута и в большинстве случаев не используется (вы можете передать вместо него NULL).

☞ Важно отметить, что тайм-ауты «взводятся» функцией *TimerTimeout()*, а запускаются по входу в одно из состояний, указанных в параметре *flags*. Сбрасывается тайм-аут при возврате из любого системного вызова. Это означает, что вы должны заново «взводить» тайм-аут перед каждым системным вызовом, к которому вы хотите его применить. Сбрасывать тайм-аут после системного вызова не надо — это выполняется автоматически.

## Тайм-ауты ядра и функция *pthread\_join()*

Самый простой пример для рассмотрения — это использование тайм-аута с функцией *pthread\_join()*. Вот как это можно было бы сделать:

```
/*
 * tt1.c
 */
#include <stdio.h>
#include <pthread.h>
#include <inttypes.h>
#include <errno.h>
#include <sys/neutrino.h>

#define SEC_NSEC 1000000000LL // В одной секунде
                               // 1 биллион наносекунд

void* long_thread(void *notused) {
    printf("Этот поток выполняется более 10 секунд\n");
    sleep(20);
}

int main(void) // Игнорировать аргументы
{
    uint64_t timeout;
    struct sigevent event;
    int rval;
    pthread_t thread_id;
    // Настроить событие — это достаточно сделать однажды
    // Либо так, либо event.sigev_notify = SIGEV_UNBLOCK:
    SIGEV_UNBLOCK_INIT(&event);
    // Создать поток
    pthread_create(&thread_id, NULL, long_thread, NULL);
    // Установить тайм-аут 10 секунд
    timeout = 10LL * SEC_NSEC;
    TimerTimeout(CLOCK_REALTIME, _NTO_TIMEOUT_JOIN, &event,
        &timeout, NULL);
    rval = pthread_join(thread_id, NULL);
    if (rval == ETIMEDOUT) {
        printf("Истекли 10 секунд, поток %d все еще"
            " выполняется!\n",
            thread_id);
    }
}
```

```

sleep(5);
TimerTimeout(CLOCK_REALTIME, _NTO_TIMEOUT_JOIN, &event,
    &timeout, NULL);
rval = pthread_join(thread_id, NULL);
if (rval == ETIMEDOUT) {
    printf("Истекли 25 секунд, поток %d все еще выполняется"
        " (нехорошо)!\n",
        thread_id);
} else {
    printf("Поток %d завершен (как и ожидалось!)\n",
        thread_id);
}
}

```

Мы применили макроопределение *SIGEV\_UNBLOCK\_INIT()* для инициализации структуры события, но можно было установить *sigev\_notify* в *SIGEV\_UNBLOCK* и «вручную». Можно было даже сделать еще более изящно, передав *NULL* вместо *struct sigevent* — функция *TimerTimeout()* понимает это как знак, что нужно использовать *SIGEV\_UNBLOCK*.

Если поток (заданный в *thread\_id*) остается работающим более 10 секунд, то системный вызов завершится по тайм-ауту — функция *pthread\_join()* возвратится с ошибкой, установив *errno* в *ETIMEDOUT*.

Вы можете использовать и другую «стенографию», указав *NULL* в качестве значения тайм-аута (параметр *ntime* в декларации выше), что предпишет ядру не блокироваться в данном состоянии. Этот прием можно использовать для организации программного опроса. (Хоть программный опрос и считается дурным тоном, его можно весьма эффективно использовать в случае с *pthread\_join()*, периодически проверяя, завершился ли нужный поток. Если нет, можно пока сделать что-нибудь другое.)

Ниже представлен пример программы, в которой демонстрируется неблокирующий вызов *pthread\_join()*:

```

int pthread_join_nb(int tid, void **rval) {
    TimerTimeout(CLOCK_REALTIME, _NTO_TIMEOUT_JOIN,
        NULL, NULL, NULL);
    return (pthread_join(tid, rval));
}

```

Все становится несколько сложнее, когда вы используете тайм-ауты ядра при обмене сообщениями. Вспомните главу «Обмен сообщениями», раздел «Обмен сообщениями и модель «клиент/сервер»» — на момент отправки клиентом сообщения сервер может как ожидать его, так и нет. Это означает, что клиент может заблокироваться как по передаче (если сервер еще не принял сообщение), так и по ответу (если сервер принял сообщение, но еще не ответил). Основной смысл здесь в том, что вы должны предусмотреть оба блокирующих состояния в параметре *flags* функции *TimerTimeout()*, потому что клиент может оказаться в любом из них.

Чтобы задать несколько состояний, сложите их операцией ИЛИ (OR):

```
TimerTimeout(... _NTO_TIMEOUT_SEND | _NTO_TIMEOUT_REPLY,  
...);
```

Это вызовет тайм-аут всякий раз, когда ядро переведет клиента в состояние блокировки по передаче (SEND) или по ответу (REPLY). В тайм-ауте SEND-блокировки нет ничего особенного — сервер еще не принял сообщение, значит, ничего для этого клиента он не делает. Это значит, что если ядро генерирует тайм-аут для SEND-блокированного клиента, сервер об этом информировать не обязательно. Функция *MsgSend()* клиента возвратит признак ETIMEDOUT и обработка тайм-аута завершится.

Однако, как было упомянуто в главе «Обмен сообщениями» (параграф «\_NTO\_CHF\_UNBLOCK»), если сервер уже принял сообщение клиента, и клиент желает разблокироваться, для сервера существует два варианта реакции. Если сервер не указал флаг \_NTO\_CHF\_UNBLOCK на канале, по которому было принято сообщение, клиент будет разблокирован немедленно, и сервер не получит об этом никакого оповещения. У большинства серверов, которые мне доводилось встречать, флаг \_NTO\_CHF\_UNBLOCK был всегда установлен. В этом случае ядро посылает серверу импульс, а клиент остается заблокированным до тех пор, пока сервер ему не ответит! Как было показано в вышеупомянутом разделе главы «Обмен сообщениями», это сделано для того, чтобы сервер мог узнать о запросе клиента на разблокирование и выполнить по этому поводу какие-то действия.

## Резюме

Мы рассмотрели функции QNX/Neutrino, ответственные за манипулирование временем, включая таймеры и их применение, а также тайм-ауты ядра. Относительные таймеры обеспечивают генерацию событий «через определенное число секунд», в то время как абсолютные таймеры генерируют события «в определенное время». Таймеры (и, вообще говоря, структура `struct sigevent`) могут обеспечить как выдачу импульса или сигнала, так и создание потока.

Ядро создает таймеры, сохраняя абсолютное время, представляющее последующее «событие», в отсортированной очереди и сравнивая текущее время (при помощи обработчика прерываний таймера) с значением, расположенным в голове этой очереди. Когда текущее время становится больше или равно времени, хранящемуся в головном элементе очереди, очередь просматривается на предмет дополнительных совпадений, после чего ядро диспетчеризует события или потоки (в зависимости типа элемента очереди) и, возможно, производит перепланирование.

Для обеспечения поддержки функций энергосбережения вы обязаны отключать периодические таймеры, когда в них нет необходимости, иначе энергосбережения как такового не произойдет — система будет все время думать, что у нее есть работа для периодического выполнения.



## **Глава 4**

### **Прерывания**

## QNX/Neutrino и прерывания

В данной главе мы рассмотрим прерывания, как с ними работать в QNX/Neutrino, их воздействие на диспетчеризацию и режим реального времени, а также некоторые стратегии их использования.

Первый вопрос, который приходит на ум: «А что такое прерывание?»

Прерывание — это в точности то, что определяется этим словом — прерывание того, что происходит в данный момент, и переход к выполнению другой задачи.

Например, предположим, что вы сидите за своим рабочим столом и выполняете задание «А». Вдруг звонит телефон — Чрезвычайно Уважаемый Клиент (ЧУК) нуждается в вашем незамедлительном ответе на некий важный вопрос. После того как вы ответите на этот вопрос, вы сможете возвратиться к заданию «А»; впрочем, возможно, что этот ЧУК изменит ваши приоритеты, и вам придется отложить задание «А» и немедленно приступить к заданию «Б».

Давайте теперь рассмотрим это в проекции на QNX/Neutrino.

В любой момент времени процессор занят обработкой готового к выполнению потока с наивысшим приоритетом (этот поток будет находиться в состоянии RUNNING («выполняется»). Чтобы вызвать прерывание, подключенная к шине компьютера аппаратура выставляет сигнал на линии прерывания (в нашей аналогии это был телефонный звонок).

Как только сигнал прерывания выставлен, ядро переключается на участок кода, который настраивает окружение для выполнения *подпрограммы обработки прерывания* (Interrupt Service Routine — ISR) — кода, который определяет, что должно происходить при обнаружении прерывания.

Интервал времени от момента установки аппаратурой сигнала прерывания до выполнения первой инструкции обработчика прерываний называют *временем реакции на прерывание*. Время реакции на прерывание измеряется в микросекундах. Различные процессоры характеризуются различными временами реакции прерывание; это зависит от быстродействия процессора, архитектуры кэша, быстродействия памяти, и, конечно, от эффективности операционной системы.

В нашей аналогии, если вы, например, слушаете музыку в наушниках и не слышите телефонного звонка, вам потребуется больше времени, чтобы обратить внимание на это «прерывание». В QNX/Neutrino может

происходить то же самое, поскольку существует инструкция процессора, которая блокирует прерывания (для процессоров x86 это инструкция `cli`). Процессор не будет обращать внимание на какие бы то ни было прерывания до тех пор, пока они не будут разблокированы (инструкция `sti` для семейства x86).

☞ Чтобы избежать процессорно-зависимых вызовов на ассемблере, QNX/Neutrino обеспечивает четыре функции: *InterruptEnable()* и *InterruptDisable()*, и *InterruptLock()* и *InterruptUnlock()*. Эти функции принимают на себя все заботы о низкоуровневых деталях всех поддерживаемых платформ.

Обработчик прерывания (ISR) обычно выполняет минимально возможный объем работы и завершается (в нашей аналогии это был бы краткий разговор по телефону с ЧУКом — не заставляя же заказчика ждать на линии несколько часов, пока мы сделаем работу! Достаточно сказать: «Не беспокойтесь, все будет сделано!»). Когда обработчик прерывания (ISR) завершается, он может либо сообщить ядру, что ничего больше делать не надо (это означает, что обработчик прерываний полностью завершил обработку события), либо что ядро должно выполнить некоторое действие, вследствие которого некий поток может переключиться в состояние READY («готов»).

В нашей аналогии сообщение ядру о том, что прерывание полностью обработано, подобно сообщению клиенту ответа на поставленный вопрос — после этого можно спокойно вернуться к тому, что мы делали раньше, зная, что вопрос клиента отработан.

Сообщение ядру о том, что требуется выполнить некоторое действие, подобно убеждению заказчика, что вы работаете над его проблемой и дополнительно сообщите, когда она будет решена. Трубка теперь повешена, но телефон может позвонить опять.

## **Подпрограмма обработки прерывания**

Обработчик прерывания (ISR) представляет собой фрагмент кода, ответственный за очистку источника прерывания.

Это ключевой момент, особенно с учетом того, что прерывание имеет приоритет *выше, чем приоритет любой программы*. Это означает, что время, затрачиваемое на выполнение обработчика прерывания, может оказать серьезное воздействие на диспетчеризацию потоков. Время

выполнения ISR должно быть минимальным. Давайте исследуем этот вопрос несколько подробнее.

### *Очистка источника прерываний*

Аппаратное устройство, которое сгенерировало прерывание, будет удерживать сигнал прерывания до тех пор, пока не удостоверится в том, что прерывание успешно обработано. Поскольку аппаратура не умеет читать мысли, программа должна сообщить ей, что отреагировала на вызвавшую прерывание причину. Обычно это выполняется путем чтения регистра состояния из определенного порта ввода/вывода или блока данных из определенного адресного пространства памяти.

При любом событии обычно есть некоторая форма подтверждения между аппаратными средствами и программным обеспечением, чтобы сбросить сигнал прерывания. (Впрочем, иногда подтверждение не предусматривается — например, когда аппаратные средства генерируют прерывание с полной уверенностью, что программное обеспечение обязательно его обработает.)

Поскольку прерывание выполняется с более высоким приоритетом, чем любой программный поток, мы должны потратить как можно меньше времени на непосредственное выполнение обработчика прерывания, чтобы свести воздействие на диспетчеризацию к минимуму. Если очистка источника прерывания выполняется простым считыванием регистра и возможно, записью полученного значения в глобальную переменную, тогда наша задача проста.

Обработка подобного рода выполняется обработчиком прерываний (ISR) последовательного порта. Аппаратура последовательного порта генерирует прерывание по приему символа. Обработчик считывает регистр, содержащий символ, записывает этот символ в кольцевой буфер. Сделано. Общее время на обработку: единицы микросекунд. Ну, собственно, так и должно быть. Представьте, что произошло бы, если бы вы принимали символы со скоростью 115 Кбод (примерно по символу каждые 100 микросекунд); если бы вы затрачивали на обработку прерывания что-то около 100 микросекунд, у вас бы больше ни на что не осталось времени!

☞ Не поймите меня неправильно. ISR последовательного порта может выполняться несколько дольше, потому что в нем еще предусмотрен опрос устройства на предмет наличия дополнительных символов в очереди.

Понятно, что минимизацию времени, затрачиваемого на обработку прерывания, можно трактовать как «повышение качества обслуживания клиента». В нашей аналогии это минимизация времени занятости телефонной линии, чтобы другие клиенты не услышали сигнал «занято».

А что если обработка слишком сложна? Есть два варианта развития событий:

- Затраты времени на очистку источника прерывания невелики, но надо много чего сделать с оборудованием (клиент задал нам короткий вопрос, но на подготовку ответа требуется значительное время).
- Затраты времени на очистку источника прерывания достаточно велики (клиент долго и запутанно объясняет свою проблему).

В первом случае мы бы захотели очистить источник прерывания как можно быстрее, а затем приказать ядру переложить работу с медленной аппаратурой на некий поток. Преимущество такой схемы состоит в том, что ISR проводит на сверхвысоком приоритете минимальное количество времени, а остальная часть работы выполняется потоками на обычных приоритетах. Это подобно ситуации, когда вы подходите к телефону (сверхвысокий приоритет), а затем передаете фактическую работу одному из своих помощников. Далее в данной главе мы рассмотрим, как ISR предписывает ядру запланировать кого-то еще.

Второй случай достаточно уродливый. Если ISR не очистит источник прерывания на момент своего завершения, ядро немедленно будет повторно прервано программируемым контроллером прерываний (Programmable Interrupt Controller — PIC; в процессорах серии x86 серии это микросхема Intel 8259 или ей эквивалентная).

☞ Специально для любителей контроллеров прерываний — мы вскоре кратко рассмотрим прерывания, активные как по уровню, так и по фронту.

Таким образом, ISR так и будет работать все время и не даст активизироваться никакому потоку, который мог бы выполнить обработку.

И какой же ущербный кусок железа может требовать продолжительного времени на очистку источника прерывания? Базовый контроллер дисководов PC удерживает сигнал прерывания на шине до тех пор, пока вы не прочтаете ряд его регистров состояния. К сожалению, данные в этих регистрах не всегда бывают доступны немедленно, и приходится опрашивать регистры на предмет поступления данных. Это может занять порядка миллисекунды — для компьютера это очень много!

Чтобы решить эту проблему, надо временно *маскировать прерывания* — явно приказать контроллеру PIC игнорировать прерывания от определенного источника прерываний, пока вы не прикажете ему сделать обратное. В этом случае, даже при активном сигнале прерывания, контроллер PIC будет игнорировать его и ничего не скажет процессору. Это позволит вашему ISR запланировать поток, чтобы вынести работу с аппаратурой за пределы обработчика прерываний. Когда ваш поток закончит передачу данных от аппаратных средств, он может приказать контроллеру PIC демаскировать это прерывание. Это позволяет снова распознавать прерывания от данного аппаратного модуля. В нашей аналогии это подобно переводу звонка ЧУКа на вашего помощника.

### ***Передача работы потоку***

Как сделать так, чтобы ISR приказал ядру запланировать поток для выполнения некоторой работы? (Или, наоборот, как сказать ядру, что ему *не следует* так поступать?)

Ниже приведен псевдокод типового ISR:

```
FUNCTION ISR
BEGIN
    определить источник прерывания
    очистить источник прерывания
    IF надо передать работу потоку THEN
        RETURN (событие);
    ELSE
        RETURN (NULL);
    END IF
END
```

Трюк здесь заключается в том, что вместо пустого указателя (NULL) можно вернуть некое событие (типа **struct sigevent**, мы говорили об этой структуре в главе «Часы, таймеры и периодические уведомления»).

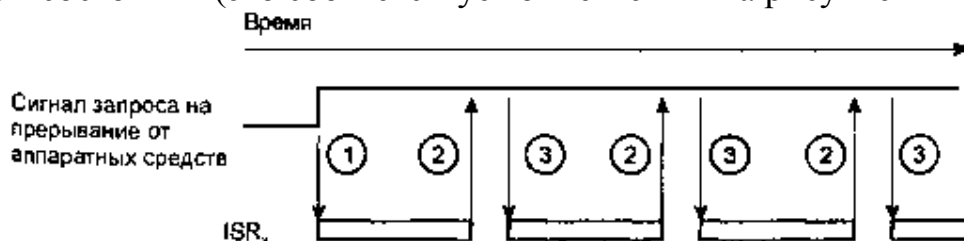
Отметим, что событие, которое вы возвращаете, должно продолжать существовать даже после того, как будет освобожден стек ISR (потому что локальные переменные хранятся в стеке — *прим. ред.*). Это означает, что событие должно либо быть описано вне ISR, либо передаваться из области устойчивых данных при помощи параметра *ISR area*, либо быть описано в пределах ISR как статическое. Это ваш выбор. Если вы возвращаете событие, ядро доставляет его потоку при возврате из вашего ISR. Поскольку событие «предупреждает» поток (путем передачи ему

импульса, как мы говорили в главе «Обмен сообщениями», или сигнала), это может заставить ядро выполнить перепланирование потоков, желающих получить процессор. Если ваш ISR возвращает NULL, это оповещает ядро, что в дополнительных действиях на уровне потоков нет необходимости, и перепланирования не произойдет — будет продолжать выполняться поток, вытесненный вашим ISR.

## Активность прерываний по уровню и по фронту

Недостает еще одного фрагмента мозаики. Большинство контроллеров прерываний могут быть запрограммированы на чувствительность либо к уровню сигнала прерывания, либо к его фронту.

В режиме чувствительности по уровню считается, что сигнал прерывания выставлен, когда соответствующая линия шины находится в активном состоянии (это соответствует отметке «1» на рисунке ниже).

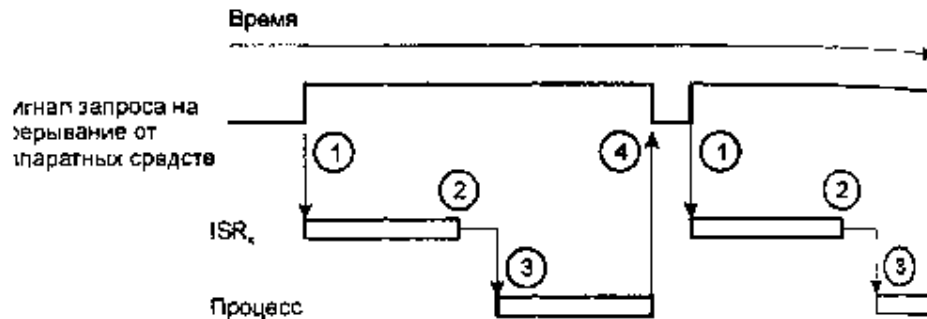


Выставление сигнала прерывания в режиме чувствительности по уровню.

Из рисунка видно, что работа с прерыванием контроллером дисководов в таком режиме привела бы к вышеупомянутой проблеме. Каждый раз при завершении ISR ядро сообщает контроллеру прерываний: «Порядок, это прерывание обработано. Сообщи мне, когда оно возникнет снова.» (отметка «2» на рисунке). Говоря техническим языком, ядро посылает контроллеру сигнал EOI (End Of Interrupt — «конец прерывания»). Контроллер PIC анализирует линию прерывания и если она все еще активна, он немедленно прерывает ядро заново (отметка «3»).

Мы могли бы обойти эту проблему, перепрограммировав контроллер прерываний в режим чувствительности по фронту.

В этом режиме прерывания распознаются контроллером только по переднему фронту сигнала.



Выставление сигнала прерывания в режиме чувствительности по фронту.

Здесь, даже если обработчик прерываний не очищает источник прерывания, после передачи ядром контроллеру сигнала EOI контроллер не может заново прервать ядро, потому что другого переднего фронта на линии прерывания после передачи EOI не будет. Для распознавания следующего прерывания на данной линии ее сначала будет необходимо деактивировать (отметка «4»), а затем активировать вновь (отметка «1»).

Похоже, что все наши проблемы решены! Будем использовать режим чувствительности по фронту и жить счастливо. Но, к сожалению, у режима чувствительности по фронту тоже есть свои проблемы.

Предположим, что ваш ISR не очистил источник прерывания. Когда ядро выдаст контроллеру сигнал EOI, аппаратные средства будут по-прежнему удерживать сигнал прерывания в активном состоянии. Однако, поскольку контроллер работает в режиме чувствительности по фронту, все последующие прерывания от этого устройства он не увидит.

Что же это за добрый парень, который так пишет ISR, чтобы тот забыл очистить источник прерывания? К сожалению, готовых рецептов здесь нет. Представьте себе ситуацию, когда два устройства (например адаптер SCSI и адаптер Ethernet), разделяющих одну и ту же линию прерывания на позволяющей это шине. (Сейчас вы скажете: «Да ну, какой придурок будет так делать?!» Ну, это иногда случается, особенно когда не хватает свободных прерываний...)

В этом случае одному и тому же вектору прерывания соответствовали бы два ISR (это, кстати, допустимо), и ядро при получении прерывания по этой линии вызывало бы их каждый раз поочередно.

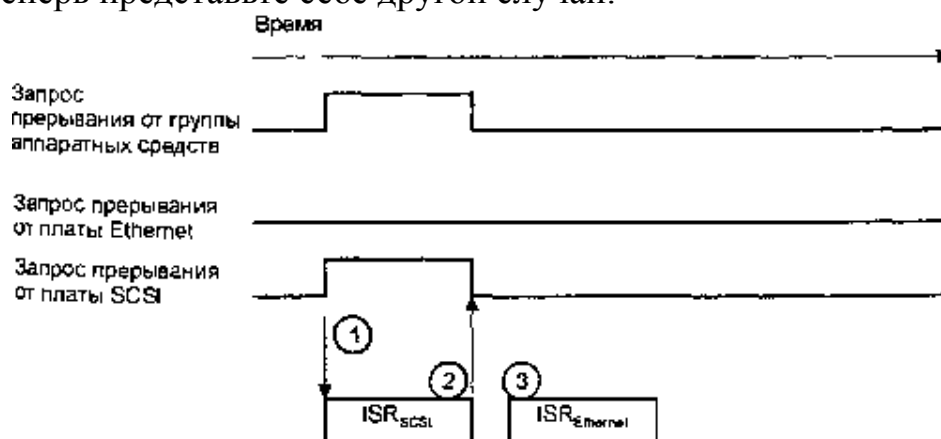
Разделяемые прерывания без перекрытия.

В этом случае, поскольку только одно из аппаратных устройств (плата SCSI) было активно, когда отработал связанный с ним обработчик корректно очистил источник прерывания (этап 2.) Отметьте, что ядро вызывает ISR для платы Ethernet (этап 3) независимо ни от чего — оно



просто не знает, какое конкретное устройство требовало обслуживания, поэтому всегда обрабатывает всю цепочку.

А теперь представьте себе другой случай:



Разделяемые прерывания с перекрытием.

Это как раз та самая проблемная ситуация.

Устройство Ethernet запрашивает прерывание первым. Это приводит к тому, что выставляется сигнал прерывания (передний фронт импульса распознается контроллером прерываний), и ядро вызывает первый в очереди обработчик прерывания (драйвер SCSI; этап 1). Обработчик драйвера SCSI смотрит на свои аппаратные средства и говорит: «Не, это не мое. Ладно, забудь» (этап 2). Затем ядро вызывает следующий обработчик прерывания в очереди, соответствующий плате Ethernet (этап 3). Обработчик драйвера Ethernet смотрит на свои аппаратные средства и восклицает: «О! Мои аппаратные средства запросили прерывание! Надо очистить источник». И тут, как назло, как раз в процессе очистки устройство SCSI генерирует прерывание (этап 4).

Когда ISR платы Ethernet завершит очистку источника прерываний (этап 5), сигнал прерывания по-прежнему останется выставлен вследствие возникшего прерывания от устройства SCSI. Однако, контроллер прерываний, запрограммированный на режим чувствительности по фронту, реагирует на переход группового сигнала прерывания из неактивного состояния в активное. А этого не будет, потому что ядро уже вызвало оба обработчика прерываний и теперь ждет другое прерывание от контроллера.

В этом случае подходящим решением был бы режима чувствительности по уровню, потому что когда ISR Ethernet завершится, и ядро выдаст контроллеру сигнал EOI, контроллер сможет распознать, что сигнал прерывания все еще активен, и прервет ядро заново. Тогда ядро снова прошло бы по всей цепочке ISR, и на этот раз обработкой занялся бы ISR драйвера SCSI.

Выбор режима чувствительности зависит от типа аппаратных средств и стартового кода. Некоторые аппаратные средства поддерживают только либо один, либо другой режим. Аппаратные средства, которые поддерживают оба режима, могут быть запрограммированы на тот или иной режим стартовым кодом. За окончательным ответом обращайтесь к документации по BSP (Board Support Package — пакет поддержки платы), поставляемого с вашей системой.

## Написание обработчиков прерываний

Давайте посмотрим, как настроить обработчики прерываний — вызовы, характеристики и кое-какие стратегии реализации.

### Подключение обработчиков прерываний

Для подключения к источнику прерывания воспользуйтесь функцией *InterruptAttach()* или *InterruptAttachEvent()*.

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

```
int InterruptAttachEvent(int intr,  
    const struct sigevent *event, unsigned flags);
```

```
int InterruptAttach(int intr,  
    const struct sigevent* (*handler)(void *area, int id),  
    const void *area, int size, unsigned flags);
```

Параметр *intr* определяет, к какому прерыванию вы хотите подключить обработчик.

Передаваемые значения определяются стартовым кодом, который перед запуском QNX/Neutrino, среди прочего, инициализирует контроллер обработки прерываний. (В документации по QNX/ Neutrino приводится подробная информация о стартовом коде; см. Справочник по утилитам, главы **startup-\***, например, **startup-p5064**.)

Уже здесь функции *InterruptAttach()* и *InterruptAttachEvent()* различаются. Рассмотрим сначала функцию *InterruptAttachEvent()*, как более простую. Затем вернемся к рассмотрению функции *InterruptAttach()*.

#### *Подключение с помощью функции InterruptAttachEvent()*

Функция *InterruptAttachEvent()* принимает два дополнительных аргумента: *event*, который является указателем на структуру **struct sigevent**, описывающую генерируемое событие, а также *flags*. Функция *InterruptAttachEvent()* сообщает ядру, что при обнаружении прерывания должно быть сгенерировано событие *event*, после чего данный уровень прерываний должен быть демаскирован. Отметим, что за то, как

интерпретировать событие и какой поток перевести в состояние READY, отвечает ядро, при обнаружении прерывания.

### *Подключение с помощью функции `InterruptAttach()`*

При использовании функции `InterruptAttach()` мы определяем другой набор параметров. Параметр *handler* — это адрес функции, которую надо вызвать. Как видно из прототипа, функция *handler()* возвращает структуру `struct sigevent` (указывающую на тип события, которое следует сгенерировать) и принимает два параметра. Первый передаваемый параметр — *area*, тот самый, который передается функции `InterruptAttach()`. Второй параметр, *id*, — идентификатор прерывания, его также возвращает `InterruptAttach()`. Он применяется для идентификации прерывания, а также для маскирования, демаскирования, блокировки и деблокировки прерывания. Четвертый параметр `InterruptAttach()`, *size*, указывает размер (в байтах) области данных, которая передается в параметре *area*. И, наконец, пятый параметр — *flags* — тот же самый параметр *flags*, что и у `InterruptAttachEvent()`; мы скоро к нему вернемся.

### *Теперь, когда вы подключились к прерыванию*

Допустим, что вы уже вызвали функцию `InterruptAttachEvent()` или `InterruptAttach()`.

☞ Поскольку подключение к прерываниям — не та вещь, которую вы хотели бы позволять кому попало, QNX/Neutrino позволяет делать это только потокам, у которых есть право «привилегированного ввода/вывода» («I/O privity») (см. функцию `ThreadCtl()` в справочном руководстве по Си-библиотеке QNX/Neutrino). Получить это право могут только потоки, которые выполняются под идентификатором пользователя `root` или которые установили свой идентификатор пользователя в `root` при помощи `setuid()`. Следовательно, реально эти права есть только у `root`. (Что, кстати, вполне логично — зачем несуперпользовательскому процессу привилегированный ввод/вывод? — *прим. ред.*)

Ниже приведен фрагмент программы, в котором реализовано подключение ISR к вектору аппаратного прерывания, идентифицируемого константой `HW_SERIAL_IRQ`:

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

int InterruptID;

const struct sigevent* intHandler(void *arg, int id) {
    ...
}

int main (int argc, char **argv) {
    interruptID =
        InterruptAttach(HW_SERIAL_IRQ, intHandler, sevent,
            sizeof(event), 0);
    if (interruptID == -1 {
        fprintf(stderr, "%s: ошибка подключения к IRQ %d\n",
            progname, W_SERIAL_IRQ);
        perror(NULL);
        exit(EXIT_FAILURE);
    }
    ...
    return (EXIT_SUCCESS);
}
```

Теперь, если по данному вектору произойдет прерывание, наш обработчик будет подвергнут диспетчеризации. При вызове функции *InterruptAttach()* ядро демаскирует указанный источник прерываний на уровне контроллера (если это прерывание еще не демаскировано, что имело бы место в случае многочисленных обработчиков одного и то же прерывания).

## Отключение обработчика прерывания

Когда вы закончили с обработчиком прерывания, вы можете пожелать уничтожить связь между ним и вектором:

```
int InterruptDetach(int id);
```

Я сказал «можете», потому что обрабатывающие прерывания потоки, как правило, используются в серверах, а серверы обычно не завершаются. Это часто ведет к предрассудку, что хорошо организованному серверу

никогда не понадобится самостоятельно вызывать *InterruptDetach()*. К тому же, при смерти потока или процесса ОС автоматически отключит все связанные с ним обработчики прерываний. Таким образом, если программа просто дойдет до конца *main()*, вызовет *exit()* или завершится по SIGSEGV, все ее ISR будут автоматически отключены от соответствующих векторов прерываний. (Впрочем, вы, вероятно, пожелаете сделать это несколько изящнее, запретив соответствующему устройству генерацию прерываний. Если же прерывание разделяемое, и его используют другие устройства, то здесь двух вариантов быть не может вообще — вы просто обязаны «убрать за собой», иначе у вас либо больше не будет прерываний (в режиме чувствительности по фронту), либо пойдет постоянный поток запросов на прерывание (в режиме чувствительности по уровню).

Продолжая вышеприведенный пример, если бы мы захотели отключиться от прерывания, то мы использовали бы следующий код:

```
void terminateInterrupts(void) {  
    InterruptDetach(interruptID);  
}
```

Если это последний ISR, связанный с данным вектором прерывания, то ядро автоматически произведет маскирование источника прерывания на уровне контроллера, чтобы таких прерываний больше не возникало.

## Параметр *flags*

Последний параметр, *flags*, управляет различными дополнительными опциями:

**\_NTO\_INTR\_FLAGS\_END**

Указывает, что данный обработчик должен сработать после всех других обработчиков данного прерывания (если они есть).

**\_NTO\_INTR\_FLAGS\_PROCESS**

Указывает на то, что данный обработчик связан с процессом, а не с потоком. Что из этого вытекает, так это условие автоматического отключения обработчика. Если вы определяете этот флаг, обработчик будет автоматически отключен от источника прерывания при завершении процесса. Если этот флаг не определен, обработчик прерывания будет отключен от источника, когда завершится поток, подключивший его.

**\_NTO\_INTR\_FLAGS\_TRK\_MSK**

Указывает, что ядро должно отследить, сколько раз данное прерывание было маскировано. Это приводит к несколько большей загрузке ядра, но это

необходимо для корректного демаскирования источника прерываний при завершении потока или процесса.

## Обработчик прерывания

Давайте рассмотрим собственно обработчик прерывания. В первом примере применим *InterruptAttach()*, а затем рассмотрим аналогичный случай, только с применением функции *InterruptAttachEvent()*.

### Применение функции *InterruptAttach()*

В продолжение примера приведем функцию *intHandler()* — наш обработчик прерывания. Она отвечает за микросхему последовательного порта 8250 (допустим, что она генерирует прерывание HW\_SERIAL\_IRQ).

```
/*
 * int1.c
 */
#include <stdio.h>
#include <sys/neutrino.h>
#define REG_RX    0
#define REG_II    2
#define REG_LS    5
#define REG_MS    6
#define IIR_MASK 0x07
#define IIR_MSR   0x00
#define IIR_THE   0x02
#define IIR_RX    0x04
#define IIR_LSR   0x06
#define IIR_MASK 0x07

volatile int serial_msr; // Сохраненное значение
                        // регистра состояния модема
volatile int serial_rx; // Сохраненное значение
                        // регистра приема
volatile int serial_lsr; // Сохраненное значение
                        // регистра состояния линии

static int base_reg = 0x2f8;
```

```

const struct sigevent* intHandler(void *arg, int id) {
    int iir;
    struct sigevent *event = (struct sigevent*)arg;

    /*
     * Определить (и очистить) источник прерывания
     * чтением регистра идентификации прерывания
     */
    iir = in8(base_reg + REG_II) & IIR_MASK;
    /* Нет прерывания? */
    if (iir & 1) {
        /* Значит, нет и события */
        return (NULL);
    }

    /*
     * Выяснить, что вызвало прерывание, и определить, надо ли
     * потоку что-нибудь с этим делать.
     * (Константы основаны на строении регистра
     * идентификации прерывания 8250.)
     */
    switch (iir) {
    case IIR_MSR:
        serial_msr = in8(base_reg + REG_MS);
        /* Разбудить поток */
        return (event);
        break;
    case IIR_THE:
        /* Ничего не делать */
        break;
    case IIR_RX:
        /* Считать символ */
        serial_rx = in8(base_reg + REG_RX);
        break;
    case IIR_LSR:
        /* Сохранить регистр состояния линии */
        serial_lsr = in8(base_reg + REG_LS);
        break;
    default:
        break;
    }
}

```



```

}
/* Никого не беспокоить */
return (NULL);
}

```

Первое, что бросается в глаза, — что все переменные, к которым обращается ISR, должны быть объявлены как **volatile**. В с единственным процессором это делается не для блага обработчиков прерываний, а для облегчения жизни потокам, которые могут быть прерваны обработчиком прерывания в любой момент. Конечно, в многопроцессорной ЭВМ обработчики прерываний вполне могли бы выполняться одновременно с кодом потоков, и в таких случаях надо быть предельно осторожными с вещами подобного рода.

С помощью ключевого слова **volatile** мы указываем компилятору не кэшировать значения этих переменных, поскольку они могут быть изменены в любой точке выполнения программы.

Следующее, на что мы обращаем внимание — это прототип самого обработчика прерывания. Он обозначен как **const struct sigevent\***. Это говорит о том, что подпрограмма *intHandler()* возвращает указатель на **struct sigevent**. Это стандарт для всех подпрограмм обработки прерываний.

Наконец, обратите внимание на то, что решение, передавать или не передавать событие потоку, принимает сам обработчик. Здесь мы генерируем событие только в случае прерывания по изменению регистра состояния модема (MSR) (событие определяется переменной *event*, которая передается обработчику прерывания в момент его подключения). Во всех других случаях мы игнорируем прерывание (и обновляем кое-какие глобальные переменные); однако, источник прерывания мы очищаем во всех случаях. Это выполняется считыванием порта ввода/вывода с помощью вызова *in8()*.

### Применение функции *InterruptAttachEvent()*

Если бы мы должны были переписать вышеприведенную программу с применением функции *InterruptAttachEvent()*, это бы выглядело так:

```

/*
 * Фрагмент int2.c
 */
#include <stdio.h>
#include <sys/neutrino.h>

```

```

#define HW_SERIAL_IRQ 3
#define REG_RX 0
#define REG_II 2
#define REG_LS 5
#define REG_MS 6
#define IIR_MASK 0x07
#define IIR_MSR 0x00
#define IIR_THE 0x02
#define IIR_RX 0x04
#define IIR_LSR 0x06
#define IIR_MASK 0x07

static int base_reg = 0x2f8;

int main(int argc, char **argv) {
    int intId; // Идентификатор прерывания
    int iir; // Регистр идентификации
                // прерывания
    int serial_msr; // Сохраненное значение
                    // регистра состояния модема
    int serial_rx; // Сохраненное значение регистра
                    // приема
    int serial_lsr; // Сохраненное значение
                    // регистра состояния линии
    struct sigevent event;
    // Обычная настройка main()...
    // Настроить событие
    intId = InterruptAttachEvent(HW_SERIAL_IRQ, &event, 0);
    for (;;) {
        // Ждать события от прерывания
        // (можно было использовать MsgReceive)
        InterruptWait(0, NULL);
        /*
         * Определить (и очистить) источник прерывания
         * чтением регистра идентификации прерывания
         */
        iir = in8(base_reg + REG_II) & IIR_MASK;
        // Демаскировать прерывание, чтобы оно
        // могло сработать снова
        InterruptUnmask(HW_SERIAL_IRQ, intId);
    }
}

```

```

/* Нет прерывания? */
if (iir & 1) {
    /* Ждать нового */
    continue;
}
/*
 * Выяснить, что вызвало прерывание,
 * и надо ли что-то с этим делать
 */
switch (iir) {
case IIR_MSR:
    serial_msr = in8(base_reg + REG_MS);
    /*
     * Выполнить какую-нибудь обработку...
     */
    break;
case IIR_THE:
    /* Не делать ничего */
    break;
case IIR_RX:
    /* Считать символ */
    serial_rx = in8(base_reg + REG_RX);
    break;
case IIR_LSR:
    /* Запомнить регистр состояния линии */
    serial_lsr = in8(base_reg + REG_LS);
    break;
}
}
/* Сюда мы не доберемся */
return (0);
}

```

Обратите внимание, что функция *InterruptAttachEvent()* возвращает идентификатор прерывания (небольшое целое число). Мы сохранили это значение в переменной *intId*, чтобы впоследствии смогли с его помощью демаскировать прерывание. После того как мы подключились к прерыванию, мы должны ждать его возникновения. Поскольку в данном случае мы применили функцию *InterruptAttachEvent()*, при каждом возникновении прерывания мы будем получать соответствующее предварительно созданное событие. Сравните это с предыдущим случаем,

где применялась *InterruptAttach()* — там решение о том, посылать событие или нет, принимал сам обработчик. При использовании функции *InterruptAttachEvent()* ядро понятия не имеет, было ли аппаратное прерывание «существенным» для нас или нет — оно просто каждый раз генерирует событие, затем маскирует соответствующее прерывание и предоставляет нам самим возможность решать, насколько это важно и что с этим делать.

В примере с *InterruptAttach()* мы принимали решение путем возврата либо **struct sigevent**, чтобы указать, что что-то должно произойти, либо константы NULL. Обратите внимание на изменения в программе в варианте с *InterruptAttachEvent()*:

- «Работа ISR» теперь выполняется потоком — в функции *main()*.
- Теперь мы должны всегда демаскировать источник прерывания после получения нашего события (потому что ядро маскирует его для нас).
- Если прерывание для нас является несущественным, мы не будем ничего предпринимать и просто пойдем дальше по программному циклу в ожидании другого прерывания.
- Если прерывание существенно для нас, мы непосредственно обрабатываем его (см. блок операторов **case IIR\_MSR**).

Где в программе очищать источник прерывания — это зависит от ваших аппаратных средств и от выбранной схемы уведомления. При использовании SIGEV\_INTR в сочетании с *InterruptWait()* ядро не ставит в очередь более одного уведомления; при использовании же SIGEV\_PULSE в сочетании с *MsgReceive()* поставлены в очередь будут все. Если вы используете сигналы (например, в сочетании с SIGEV\_SIGNAL), вы сами определяете, ставить их в очередь или нет. Одни аппаратные средства требуют очистки источника прерываний до начала чтения данных, другие — нет, и можно читать данные из них при выставленном сигнале прерывания.

Такой сценарий как ISR, возвращающий SIGEV\_THREAD, повергает меня в дикий ужас. Настоятельно рекомендую по возможности избегать этого «приема».

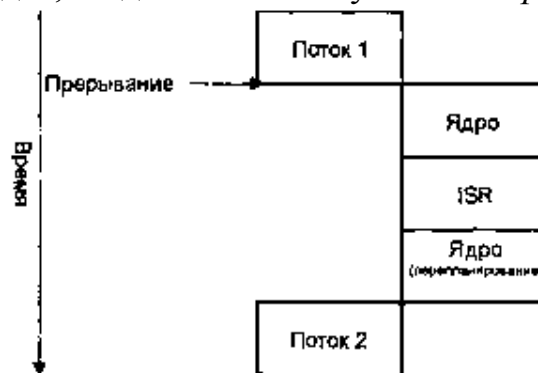
В вышеприведенном примере с программированием последовательного порта мы приняли решение использовать функцию *InterruptWait()*, которая ставит в очередь только одно событие. Аппаратура последовательного порта может выставить еще одно прерывание сразу после чтения нами регистра идентификации прерываний (IIR), но это нормально, потому что в очередь будет поставлен максимум один SIGEV\_INTR. Мы подберем это уведомление на следующей итерации цикла **for**.

## Различия между *InterruptAttach()* и *InterruptAttachEvent()*

Напрашивается естественный вопрос: «Когда и какую функцию выбирать?»

Наиболее очевидное преимущество *InterruptAttachEvent()* состоит в том, что применять ее значительно проще, чем *InterruptAttach()* — поскольку нет никакого ISR, его и отлаживать не надо. Другое преимущество состоит в том, что поскольку в пространстве ядра ничего не выполняется (а ISR делал бы именно так), нет никакой опасности разрушить систему — если вы столкнетесь с ошибкой программирования, то пострадает конкретный процесс, а не система в целом. Однако, применение этой функции будет более или менее эффективным по сравнению с *InterruptAttach()* в зависимости от того, чего вы пытаетесь достичь. Этот вопрос достаточно сложен, и сводить его к нескольким словам (типа «быстрее» или «лучше») было бы неправильно. Давайте рассмотрим несколько возможных сценариев.

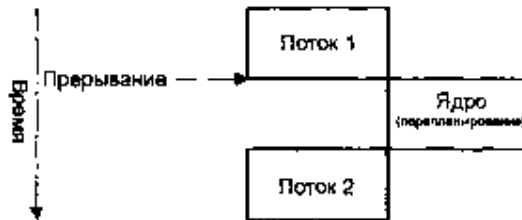
Вот что происходит, когда мы используем *InterruptAttach()*:



Поток управления при использовании *InterruptAttach()*.

Выполняющийся поток («Поток 1») прерывается, и мы переключаемся в ядро. Ядро сохраняет контекст «Потока 1». Затем ядро смотрит, кто ответственен за обработку данного прерывания и решает, что это «ISR». Ядро настраивает контекст для «ISR» и передает ему управление. «ISR» опрашивает аппаратуру и решает вернуть `struct sigevent`. Ядро отмечает возвращаемое событие, выясняет, кто должен его обработать, и переводит их в состояние READY. Это может привести к планированию ядром другого потока, «Потока 2».

Теперь давайте сопоставим это с тем, что будет происходить при использовании *InterruptAttachEvent()*:



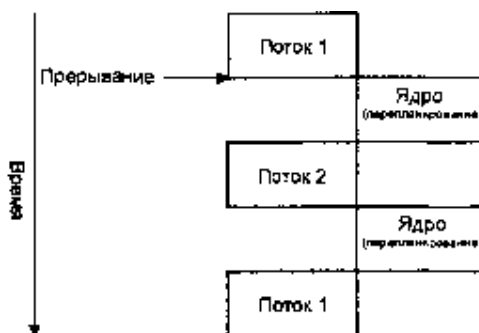
Поток управления при использовании *InterruptAttachEvent()*.

В этом случае путь обслуживания прерываний намного короче. Мы выполнили одно переключение контекста от выполнявшегося потока («Поток 1») в ядро. Вместо второго переключения контекста в ISR ядро просто «притворилось», что получило от ISR `struct sigevent` и среагировало на него, запланировав «Поток 2».

Теперь вы думаете: «Великолепно! Забудем про *InterruptAttach()* и будем использовать простую функцию *InterruptAttachEvent()*.»

Это не такая хорошая идея, как кажется на первый взгляд, потому что вы можете и не захотеть просыпаться от каждого прерывания, генерируемого аппаратурой! Вернитесь к примеру, который приведен выше — событие там возвращалось только тогда, когда изменялся регистр состояния модема, а не по приему символа, изменению регистра состояния линии или опустошению буфера передачи.

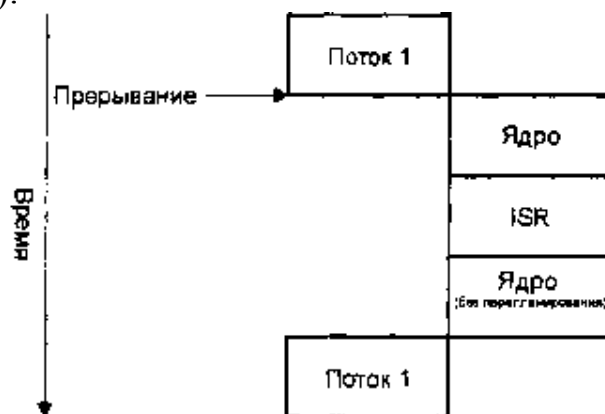
В этом случае, особенно если бы последовательный порт принимал символы (которые вы хотели бы проигнорировать), вы бы потратили много времени впустую на перепланирование своего потока — и только ради того, чтобы он проанализировал состояние последовательного порта и принял решение, что ничего делать не надо. В данном случае все бы выглядело примерно так:



Поток управления при использовании *InterruptAttachEvent()* с излишним перепланированием.

Происходящее по сути заключается в том, что вы вызываете переключение контекста для перехода к «Потоку 2», он опрашивает аппаратуру и понимает, что делать ничего не требуется, и это влечет за собой еще одно лишнее переключение контекста обратно в «Поток 1».

Вот что произошло бы, если бы вы применили функцию *InterruptAttach()*, но не пожелали планировать другой поток (т.е. просто вернулись обратно):



Поток управления при использовании *InterruptAttach()* без перепланирования потоков.

Ядро знает, что выполнялся «Поток 1», и что ISR не сказал ему что-либо сделать, поэтому после прерывания оно может смело вернуть управление «Потоку 1».

Для справки: вот что делает функция *InterruptAttachEvent()* (это не реальный исходный текст, поскольку функция *InterruptAttachEvent()* в действительности связывает с ядром структуру данных — она не реализована как отдельная вызываемая функция!):

```
// «Внутренний» обработчик
static const struct sigevent*
internalHandler(void *arg, int id) {
    struct sigevent *event = arg;
    InterruptMask(intr, id);
    return (arg);
}

int InterruptAttachEvent(int intr,
const struct sigevent *event, unsigned flags) {
    static struct sigevent static_event;
    memcpy(&static_event, event, sizeof(static_event));
    return
        (InterruptAttach(intr, internalHandler, &static_event,
            sizeof(*event), flags));
}
```

## Что выбрать?

Так какую функцию применять? От редко возникающих прерываний почти всегда можно отмахнуться применением *InterruptAttachEvent()*. Поскольку прерывания будут происходить редко, даже лишние перепланирования потоков значительного воздействия на общую производительность системы не окажут. Единственный момент, когда это проявится — это если на данном прерывании будут «сидеть» еще и другие устройства; в этом случае, поскольку функция *InterruptAttachEvent()* маскирует источник прерывания, то это прерывание останется заблокированным до тех пор, пока источник не будет демаскирован обратно. Если при этом первое устройство требует много времени на обслуживание, остальным придется все это время ждать демаскирования. По большому счету, это проблема аппаратной организации системы — не следует размещать медленные устройства на одной линии прерывания с быстрыми.

При выборе функции для более часто возникающих прерываний требуется учесть множество факторов:

- Ненужные прерывания — если их число будет существенным, лучше применять *InterruptAttach()* и отфильтровывать их прямо в ISR. Возьмем, например, тот же случай с последовательным устройством. Поток может выдать команду: «Дай мне 64 байта». Если ISR запрограммирован с учетом того, что пока не будут приняты все 64 байта, ничего полезного не произойдет, все возникающие в процессе приема прерывания будут отфильтрованы. ISR возвратит событие только после окончания приема всех 64 байт.

- Время реакции — если ваши аппаратные средства чувствительны к интервалу времени от момента выставления запроса на прерывание до отработки ISR, вам следует использовать *InterruptAttach()*, чтобы свести это время к минимуму. Это сработает, потому что диспетчеризация ISR в ядре выполняется очень быстро.

- Буферизация — если ваша аппаратура имеет встроенные средства буферизации, вы можете обойтись функцией *InterruptAttachEvent()* и очередью из единственного события, как в случае с комбинацией SIGEV\_INTR и *InterruptWait()*. Этот метод позволяет прерываниям возникать с такой частотой, как им захочется» при этом позволяя вашему потоку выбирать значения из буфера с такой скоростью, с какой он сможет. Поскольку данные буферизуются на аппаратном уровне, никаких проблем со временем реакции на прерывание не будет.



## Функции, которые может вызывать ISR

Следующий вопрос, за который следует взяться, — это список функций, которые может вызывать ISR.

Небольшое отступление. Исторически, причина основных затруднений при написании обработчиков прерываний заключалась (и в большинстве других операционных систем до сих пор заключается) в том, что ISR работают в особом окружении.

Одна из конкретных причин, усложняющих написание ISR, состоит в том, что с точки зрения ядра ISR на самом деле не является «полноправным» потоком. С позиции ядра это, если хотите, такой таинственный «аппаратный» поток. Это означает, что ISR не имеет права делать никаких манипуляций «на уровне потока» — таких как, например, обмен сообщениями, синхронизация, системные вызовы, дисковый ввод/вывод, и т.д.

Не усложняет ли это написание ISR? Конечно. И поэтому решение заключается в том, чтобы в самом теле обработчика выполнять минимум работы, а все остальное делать уже на уровне потока, где есть доступ ко всем сервисам.

Ваши цели при написании ISR должны заключаться в следующем:

- считать переменчивую (в оригинале было «transient» — *прим. ред.*) информацию;
- очистить источник прерывания;
- возможно, запланировать поток, который сделает реальную работу.

Такая «архитектура» держится на том, что QNX/Neutrino обеспечивает очень быстрые времена переключения контекста. Вы знаете, что сможете быстро переключиться в ваш обработчик для выполнения работы, критичной по времени. Вы также знаете, что когда обработчик возвратит событие для запуска потока, то поток тоже активизируется очень быстро. И именно эта философия «ничего не делайте в теле ISR» делает обработчики прерываний в QNX/Neutrino столь простыми!

Итак, какие же вызовы можно использовать в теле ISR? Вот официальный список:

- функции семейства *atomic\_\*()* (например, *atomic\_set()*);
- функции семейства *mem\*()* (типа *memcpy()*);
- большинство функций семейства *str\*()* (типа *strcmp()*).

Остерегайтесь, однако, потому что не все эти функции являются безопасными — например, *strdup()* вызывает *malloc()*, в которой используется мутекс, а это запрещено.

Вообще, что касательно строковых функций, перед их использованием надо индивидуально смотреть их описание в руководстве по Си-библиотеке;

- *InterruptMask()*;
- *InterruptUnmask()*;
- *InterruptLock()*;
- *InterruptUnlock()*;
- *InterruptDisable()*;
- *InterruptEnable()*;
- *in\*()* и *out\*()*.

Основное эмпирическое правило формулируется примерно так: «Не используйте ничего, что требует большого объема стека или больших затрат времени, и не используйте ничего, что делает системные вызовы». Требование по стековому пространству проистекает из того факта, что ISR имеют очень ограниченный объем стека.

Список функций, безопасных для применения в ISR, имеет реальный смысл — например, если вам потребуется скопировать область памяти, хорошим выбором будет применение функций типа *mem\*()* и *str\*()*. Скорее всего, вам потребуется читать регистры аппаратных средств (например, чтобы сохранить какие-либо значения или очистить источник прерывания), тогда вам пригодятся функции ввода/вывода из семейств *in\*()* и *out\*()*.

А как насчет ошарашивающего выбора функций семейства *Interrupt\*()*? Давайте рассмотрим их попарно.

#### *InterruptMask()* и *InterruptUnmask()*

Эти функции ответственны за маскирование источника прерывания на уровне контроллера; это предохраняет прерывания от передачи процессору. Обычно эти функции применяются, когда вы хотите доделать работу в потоке, но не можете очистить источник прерывания непосредственно в теле ISR. В этом случае ISR должен вызвать *InterruptMask()*, а поток, после завершения работы, — *InterruptUnmask()*.

Имейте в виду, что число вызовов *InterruptUnmask()* должно соответствовать числу вызовов *InterruptMask()* — чтобы прерывание продолжало работать, вы обязаны демаскировать его ровно столько раз, сколько раз оно было маскировано.

Заметьте, между прочим, что функция *InterruptAttachEvent()* выполняет *InterruptMask()* автоматически (в ядре), поэтому ваш обрабатывающий прерывание поток должен вызывать *InterruptUnmask()*.

#### *InterruptLock()* и *InterruptUnlock()*

Эти функции используются для блокировки (*InterruptLock()*) и деблокировки (*InterruptUnlock()*) прерываний в одно- или

многопроцессорной системе. Вам может понадобиться заблокировать прерывания, например, чтобы защитить поток от ISR (или, дополнительно, в SMP-системе — защитить ISR от потока). Когда вы сделаете нужные манипуляции с критическими данными, вы сможете деблокировать прерывания обратно. Отметьте, что данные функции рекомендованы к применению вместо известных вам функций *InterruptDisable()* и *InterruptEnable()*, потому что корректно работают в SMP-системах. По сравнению со «старыми» функциями, проверка на многопроцессорность вносит дополнительные издержки, но в однопроцессорной системе ими можно пренебречь, поэтому я рекомендую вам всегда использовать *InterruptLock()* и *InterruptUnlock()*.

#### *InterruptDisable() и InterruptEnable()*

Не используйте эти функции в новых проектах. Исторически, эти функции применялись для вызова инструкций `cli` и `sti` в процессорах серии x86, когда QNX/Neutrino еще не была многоплатформенной ОС.

С тех пор функции были модернизированы для работы со всеми типами процессоров, но чтобы не огорчать SMP-системы, используйте лучше функции *InterruptLock()* и *InterruptUnlock()*.

Еще одна вещь, которую не вредно будет повторить, заключается в том, что в SMP-системе возможно одновременное выполнение ISR и другого потока.

## Резюме

При работе с прерываниями принимайте во внимание следующие положения:

- Не оставайтесь в обработчике прерывания слишком долго — выполняйте в нем минимальный объем работы. Это поможет сократить время реакции на прерывание и упростить отладку.
- Применяйте функцию *InterruptAttach()* только тогда, когда нужно обращаться к аппаратным средствам непосредственно после прерывания, в противном случае избегайте ее.
- Применяйте функцию *InterruptAttachEvent()* во всех других случаях. Ядро планирует поток (на основе события, которое вы передадите) для обработки возникшего прерывания.
- Защищайте переменные, используемые как в обработчиках прерываний (при использовании *InterruptAttach()*), так и в потоках, путем вызова *InterruptLock()* и *InterruptUnlock()*.
- Объявляйте переменные, используемые в качестве посредников между потоками и обработчиками прерываний, как `volatile`, чтобы компилятор не кэшировал их «просроченные» значения, уже измененные обработчиком прерывания.

## **Глава 5**

# **Администраторы ресурсов**

## Что такое администратор ресурсов?

В данной главе мы рассмотрим все, что вы должны знать для самостоятельного написания *администратора ресурса*.

Администратор ресурса — это просто программа с рядом четко определенных характеристик. Эта программа по-разному называется в различных операционных системах — «драйвер», «устройство», «драйвер устройства», «администратор ввода/вывода», «файловая система», и т.п. Однако, во всех случаях предназначение этой программы (мы будем называть ее просто «администратором ресурса») заключается в том, чтобы предоставить абстрактную форму некоего сервиса.

Также, поскольку QNX/Neutrino является POSIX-совместимой ОС, основу предоставляемой абстракции составляют спецификации POSIX.

## Примеры администраторов ресурсов

Прежде чем уйти в тонкости проблемы, давайте проанализируем пару примеров и увидим, как в них «абстрагируются» сервисы. Рассмотрим реальный аппаратный блок (последовательный порт) и кое-что более абстрактное (файловую систему).

### *Последовательный порт*

В типовой системе обычно существует какой-нибудь способ программирования обмена информацией по последовательному интерфейсу типа RS-232. Этот интерфейс составляют ряд аппаратных устройств, включая микросхему UART (Universal Asynchronous Receiver Transmitter — универсальный асинхронный приемопередатчик), которая умеет преобразовывать параллельные данные от центрального процессора в последовательный поток и обратно.

В этом случае сервисом, предоставляемым соответствующим администратором ресурса, будет возможность передачи и приема символьных данных через последовательный порт.

Мы говорим, что имеет место «абстрагирование» сервиса, потому что клиентская программа (та, которая непосредственно использует сервис) не знает (да и незачем ей) о микросхеме UART и ее реализации. Все, что знает клиентская программа, — что для передачи символа она должна вызвать

функцию *fprintf()* а для приема символов — функцию *fgets()*. Обратите внимание, что взаимодействия с последовательным портом мы использовали стандартные функции POSIX.

### **Файловая система**

В качестве другого примера администратора ресурса рассмотрим файловую систему. Она состоит из ряда взаимодействующих модулей: собственно файловой системы, драйвера блочного ввода/вывода и дискового драйвера.

Предлагаемый здесь сервис состоит в способности считывать и записывать символы на некоторый носитель. Абстракция здесь та же самая, что и в предыдущем примере с последовательным портом — клиентская программа по-прежнему может использовать *те же самые* вызовы функций (например, *fprintf()* и *fgets()*) для доступа к носителю. Фактически, клиент действительно не знает или даже не должен знать, с каким конкретно администратором ресурсов он имеет дело.

### **Характеристики администраторов ресурсов**

Как мы увидели в приведенных выше примерах, ключом к универсальности администраторов ресурсов является возможность использования стандартных функций POSIX — мы ведь не использовали никакие «специальные» функции, когда общались с последовательным портом. А если вам понадобится сделать нечто «особенное», характерное только для применяемого вами устройства? Например, операция настройки скорости обмена по последовательному порту в бодах специфична для администратора последовательного порта, но абсолютно бессмысленна для администратора файловой системы. Аналогично, операция по позиционированию в файле с помощью функции *lseek()* имеет смысл для файловой системы, но является несодержательной для последовательного порта. В POSIX эта проблема решается просто. Некоторые функции — например, *lseek()* — при попытке применить их к устройству, которое их не поддерживает, просто возвращают код ошибки. Для реализации функций, специфичных для каждого устройства, в POSIX предусмотрена функция управления устройствами, *devctl()*. Если устройство не понимает команду, передаваемую ему посредством *devctl()*, оно просто возвращает код ошибки, аналогично устройствам, которые не понимают функцию *lseek()*.

Поскольку мы уже упомянули функции *lseek()* и *devctl()* как общеупотребительные, следует заметить, что администраторы ресурсов обычно поддерживают весь спектр функций, работающих с дескрипторами файлов (или **FILE\*** *stream*).

Это естественно приводит нас к выводу о том, что администраторы ресурсов будут работать почти исключительно с вызовами дескриптор-ориентированных функций. Поскольку QNX/Neutrino — операционная система, организованная на основе обмена сообщениями, из этого следует, что вызовы POSIX-функций транслируются в сообщения, которые затем пересылаются администраторам ресурсов.

Именно эта трансляция вызовов POSIX в сообщения позволяет нам отвязать клиентуру от администраторов ресурсов. Все, что должен уметь делать администратор ресурса, — это обрабатывать ряд строго определенных сообщений. Все, что должен уметь делать клиент, — это генерировать эти самые строго определенные сообщения, которые администратор ресурса ожидает принимать и обрабатывать.

Поскольку взаимодействие между клиентурой и администраторами ресурсов основано на обмене сообщениями, имеет смысл делать этот «передаточный уровень» как можно «тоньше». Например, когда клиент выполняет функцию *open()* и получает в ответ дескриптор файла, этот дескриптор фактически является идентификатором соединения! Данный идентификатор соединения (он же дескриптор файла) используется затем функциями клиентской Си-библиотеки (например, функцией *read()*) при создании и отправке сообщения для администратора ресурсов.



## Взгляд со стороны клиента

Мы уже намекнули, что ожидает клиент. Он ожидает интерфейс на основе файловых дескрипторов с применением стандартных функций POSIX.

В действительности «под колпаком» происходит еще кое-что.

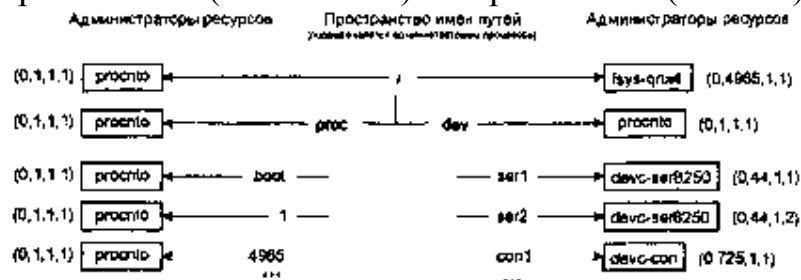
Например, как в действительности клиент соединяется с соответствующим администратором ресурса? Что происходит в случае объединённых файловых систем (когда несколько файловых систем ответственны за то же самое пространство имен)? Как обрабатываются каталоги?

## Поиск сервера

Первое, что делает клиент, — это вызывает *open()*, чтобы получить дескриптор файла. (Заметьте, что если клиент вместо этого вызывает функцию более высокого уровня — например, *fopen()* — утверждение остается справедливым, поскольку *fopen()* в конечном счете вызывает *open()*).

Реализация функции *open()* в Си-библиотеке создает сообщение, которое затем пересылается администратору процессов (*procnto*).

Администратор процессов отвечает за поддержание информации о пространстве имен путей. Данная информация представляет собой древовидную структуру имен путей, с которой связаны дескрипторы узлов (*node descriptors*), идентификаторы процессов (*process IDs*), идентификаторы каналов (*channel IDs*) и обработчики (*handles*):



Пространство имен путей в QNX/Neutrino.

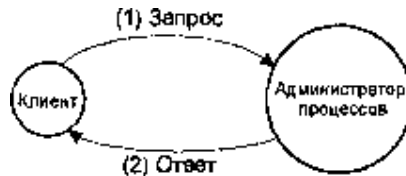
☞ Отметьте, что на представленном выше рисунке и в последующих описаниях для обозначения администратора ресурса, который реализует файловую систему QNX4, я

использовал имя **fsys-qnx4**. В действительности все немного сложнее, потому что драйверы файловой системы представляют собой группы связанных между собой динамических библиотек (DLL), так что никакой программы с именем **fsys-qnx4** на самом деле не существует; мы просто используем это имя в качестве «заполнителя» для компонента файловой системы.

Давайте предположим, что клиент вызывает функцию *open()*:

```
fd = open("/dev/ser1", O_WRONLY);
```

Реализация функции *open()* в клиентской Си-библиотеке создает сообщение и пересылает его администратору процессов. Это сообщение гласит: «Хочу открыть **/dev/ser1**. К кому мне обратиться по этому вопросу?»



Первая стадия разрешения имени.

Администратор процессов принимает запрос и просматривает дерево имен на предмет соответствия (давайте предположим здесь, что нам необходимо точное соответствие). Имя пути **«/dev/ser1»** вполне подойдет, и администратор процессов может ответить клиенту: «Нашел **/dev/ser1**. За обработку отвечает канал 1 процесса 44 на узле 0, спроси его!»

Не забывайте: мы все еще в клиентском коде *open()*!

Функция *open()* создает другое сообщение и соединяется с указанным процессом (PID 44) на указанном узле (NID 0 означает локальный узел) по заданному каналу (CHID 1), помещая обработчик (*handle*) непосредственно в сообщение. Это воистину «сообщение установки соединения» — то самое сообщение которое клиентская функция *open()* использует для установления связи с администратором ресурса (3 стадия на рисунке ниже) Когда администратор ресурса получает сообщение установки соединения, он анализирует его и проверяет на корректность. Например, вы могли бы попытаться применить операцию записи к администратору ресурса, который реализует файловую систему с доступом только для чтения — в этом случае вы бы получили обратно признак ошибки (в данном случае — EROFS). В нашем примере, однако, администратор последовательного порта смотрит на запрос (мы указали там **O\_WRONLY**, что для последовательного порта абсолютно кошерно) и отвечает признаком EOK (4 стадия на рисунке ниже).



Сообщение `_IO_CONNECT`.

Затем, наконец, клиентская функция `open()` возвращает клиенту корректный дескриптор файла.

На самом деле этот дескриптор файла представляет собой идентификатор соединения, который мы только что использовали для отправки сообщения администратору ресурса! Если бы администратор ресурса *не* ответил признаком ЕОК, мы бы сообщили клиенту, что произошла ошибка (`open()` возвратила бы `-1` и установила код ошибки в `errno`).

## Поиск администратора процессов

Теперь, когда мы знаем основные этапы поиска конкретного администратора ресурса, осталось раскрыть тайну поиска администратора процесса, с которого все начинается. На самом деле все очень просто. По определению, администратору процессов соответствует дескриптор узла 0 (то есть текущий узел), идентификатором процесса 1 и идентификатор канала 1. Так что администратор процессов всегда идентифицируется триплетом ND/PID/CHID, равным 0/1/1.

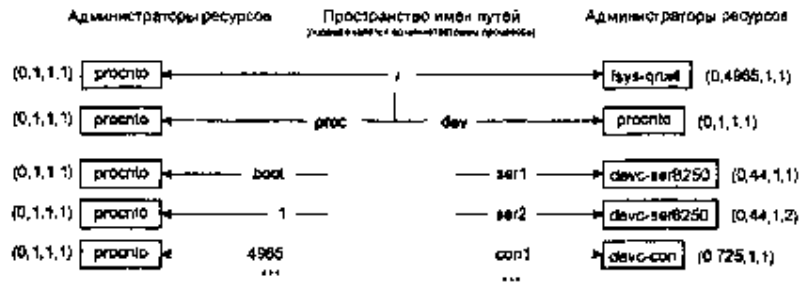
## Обработка каталогов

Пример, рассмотренный выше, относился к администратору последовательного порта. Мы также высказывали предположение, что хотим точного соответствия имен путей при поиске по дереву. Это предположение справедливо только наполовину — все соответствия имен путей, о которых мы будем говорить в этой главе, основаны на *полном* соответствии *компонента* имени пути, но вовсе не обязательно имени пути *целиком*. Давайте вкратце это поясним.

Предположим, у меня есть код, который делает следующее:

```
fp = fopen("/etc/passwd", "r");
```

Напомним, что функция `fopen()` в конечном счете вызывает функцию `open()`, так что реально мы имеем функцию `open()`, запрашивающую имя пути `/etc/passwd`. Но такого имени на рисунке нет:



Пространство имен путей в QNX/Neutrino.

Однако, из рисунка видно, что модуль **fs-qnx4** зарегистрировал свою тройку ND/PID/CHID для имени пути «/». Хотя это и не показано на рисунке, файловая система **fs-qnx4** зарегистрировалась как «администратор каталога», сказав администратору процессов, что будет отвечать за «/» и все то, что расположено «ниже». «Администраторы устройств» (например, администратор последовательного порта) так не делают. Установив флаг каталога, **fs-qnx4** получает возможность обработать запрос для имени пути «/etc/passwd», потому что это имя начинается с «/», а значит, есть совпадение!

А что произошло бы, если бы мы попытались сделать так?

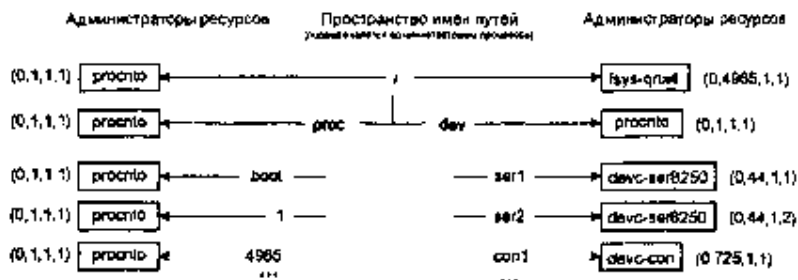
```
fd = open("/dev/ser1/9600.8.1.n", O_WRONLY);
```

Ну, поскольку у администратора последовательного порта не установлен флаг каталога, администратор процессов увидит это и скажет: «Опаньки, извините, /dev/ser1 — не каталог. В обработке отказано». Запрос прямо здесь и заканчивается — администратор процессов даже не возвращает функции *open()* четверку ND/PID/CHID/handle.

Из параметров функции *open()* в примере выше видно, что может показаться заманчивой идеей позволить некоторым «традиционным» устройствам открываться с дополнительными параметрами, указываемыми после «обычного» имени. Однако, эмпирическое правило здесь такое: если это пройдет на совещании по организации проекта, тогда вперед. Некоторые из моих студентов, услышав это от меня, заявляют: «Так я и есть сам себе комитет по проектным решениям!» На что я обычно отвечаю. «Пистолет у вас есть. Прострелите себе ногу. :-))»

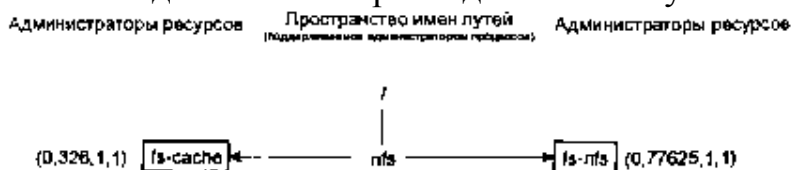
## Объединенные файловые системы

Взгляните повнимательнее на уже знакомый нам рисунок.



## Пространство имен путей в QNX/Neutrino.

Обратите внимание, что ответственными за префикс «/» объявили себя как файловая система **fs-qnx4**, так и администратор процессов. Это нормально, и беспокоиться тут не о чем. Мало того, иногда это оказывается очень даже неплохой идеей. Рассмотрим один такой случай.



## Файловые системы с перекрытием

Предположим, что у вас очень медленное сетевое соединение, и вы смонтировали поверх него сетевую файловую систему. Вы замечаете, что некоторые файлы используются достаточно часто, и хотели бы, чтобы эти файлы неким волшебным способом «кешировались» на вашей машине, но увы и ах, проектировщики сетевой файловой системы это почему-то не предусмотрели. И тогда вы решаете самостоятельно написать кеширующую файловую систему (назовем ее, например, **fs-cache**) и поместить ее поверх сетевой файловой системы. Вот как это будет смотреться с позиции клиента:

Обе файловые системы, **fs-nfs** (сетевая файловая система) и ваша кэшированная файловая система (**fs-cache**) регистрируются под одним и тем же префиксом, «/nfs» уже упомянули выше, в QNX/Neutrino это нормально и абсолютно законно.

Предположим, что ваша система только что стартовала, и в вашей кэшированной файловой системе еще ничего нет. Клиентская программа пробует открыть какой-нибудь файл — скажем **/nfs/home/rk/abc.txt**. Ваша кэшированная файловая система находится «перед» сетевой файловой системой (я потом покажу вам, как это сделать, когда мы будем обсуждать реализацию администратора ресурса).

Клиентский вызов *open()* выполняет свои обычные действия:

1. Спрашивает администратор процессов: «К кому обратиться по поводу файла **/nfs/home/rk/abc.txt**?»

2. Получает ответ от администратора процессов: «Поговори сначала с **fs-cache**, а потом с **fs-nfs**».

Обратите внимание, что здесь администратор процессов возвращает *две* четверки ND/PID/CHID/handle — одну для файловой системы **fs-cache** и одну для файловой системы **fs-nfs**. Это критично.

Далее функция *open()* делает следующее:

1. Направляет сообщение файловой системе **fs-cache**. «Я бы хотел открыть файл `/nfs/home/rk/abc.txt` на чтение, пожалуйста.»

2. Получает ответ от файловой системы **fs-cache**: «Сожалею, но я никогда о таком не слышала.»

Здесь становится ясно, что с администратором файловой системы **fs-cache** клиентской функции *open()* не повезло. Файл не существует! Однако, вызов *open()* знает, что он получил список из двух четверок ND/PID/CHID/handle, и поэтому пробует второй вариант:

1. Направляет сообщение файловой системе **fs-nfs**: «Я бы хотел открыть файл `/nfs/home/rk/abc.txt` на чтение, пожалуйста.»

2. От файловой системы приходит ответ: «Запросто, никаких проблем!»

Теперь, после того как у функции *open()* есть ЕОК («никаких проблем»), она возвращает дескриптор файла. Все дальнейшие операции клиент выполняет непосредственно с администратором сетевой файловой системы **fs-nfs**.

☞ Имя пути разрешается только один раз — во время вызова функции *open()*. Это означает, что как только мы успешно открыли нужный администратор ресурса, все дальнейшие вызовы, работающие с дескрипторами файлов, будут идти через него.

Так когда же вступает в игру наша кеширующая файловая система **fs-cache**? Ну, допустим, пользователь закончил считывание файла (файл теперь загружен в текстовый редактор). Когда файл понадобится сохранить, произойдет та же самая последовательность действий, но возникнет один любопытный поворот:

1. Сообщение администратору процессов: «С кем я должен переговорить насчет файла `/nfs/home/rk/abc.txt`?»

2. Ответ администратора процессов: «Поговори сначала с **fs-cache**, а затем с **fs-nfs**».

3. Сообщение **fs-cache**: «Мне хотелось бы открыть файл `/nfs/home/rk/abc.txt` на запись, пожалуйста».

4. Ответ от **fs-cache**: «Запросто, нет проблем».

Обратите внимание на то, что на 3 этапе мы открыли файл на запись, а не на чтение, как в первый раз. Поэтому не удивительно, что **fs-cache** на этот раз разрешает эту операцию (этап 4).

Еще более интересные события происходят, когда мы повторно пытаемся прочитать этот файл:

1. Сообщение администратору процессов: «С кем я должен переговорить насчет файла `/nfs/home/rk/abc.txt`?»

2. Ответ администратора процессов: «Поговори сначала с **fs-cache**, а затем с **fs-nfs**».

3. Сообщение **fs-cache**: «Мне хотелось бы открыть файл `/nfs/home/rk/abc.txt` на *чтение*, пожалуйста».

4. Ответ от **fs-cache**: «Запросто, нет проблем».

Да-да, на этот раз **fs-cache** обработала запрос на чтение!

Мы опустили несколько деталей, но для восприятия базовых идей они не так важны. Очевидно, кеширующая файловая система должна предусматривать некоторый способ отправки данных по сети на «реальный» носитель. Она также должна уметь перед отправкой данных клиенту проверять, не изменился ли файл (чтобы клиент не получил устаревшие данные). К тому же, кеширующая файловая система вполне могла бы сама обработать первый запрос на чтение, загрузив данные из сетевой файловой системы в свой кэш. И так далее.

***Объединенные файловые системы (UFS — Unioned File Systems) и объединенные точки монтирования (UMP — Unioned Mount Points)***

Дабы не путать понятия, сделаем небольшой экскурс в терминологию. Основное различие между объединенной файловой системой (UFS) и объединенной точкой монтирования (UMP) заключается в том, что UFS ориентирована на файлы, а UMP — на точки монтирования. В вышеупомянутой кеширующей файловой системе у нас была UFS, потому что оба администратора могли получить доступ к файлу вне зависимости от глубины его размещения файла в дереве каталогов. Давайте для примера рассмотрим другой администратор ресурса (назовем его «**foobar**»), отвечающий за путь «`/nfs/other`». В UFS-системе процесс **fs-cache** был бы способен кэшировать файлы и оттуда тоже, присоединившись к «`/nfs`». В случае с UMP, что принято в QNX/Neutrino по умолчанию, поскольку там все основано на соответствии самого длинного префикса, запросы смог бы обрабатывать только администратор **foobar**.

## Резюме о клиенте

На этом с клиентом все. Перечислим ключевые моменты, которые следует запомнить:

- Клиент обычно налаживает связь с администратором ресурса с помощью вызова *open()* (или *fopen()*).
- После того как запрос клиента разрешился в конкретный администратор ресурса, мы его больше не меняем.
- Все дальнейшие клиентские сообщения в этом сеансе основываются на дескрипторах файлов (или **FILE\*** **stream**) — например, *read()*, *lseek()*, *fgets()*, и т.п.
- Сеанс прекращается, когда клиент закрывает дескриптор файла или поток (или завершается по какой-либо причине).

Все вызовы, основанные на дескрипторах файлов, транслируются в сообщения.



## Взгляд со стороны администратора ресурсов

Давайте теперь посмотрим на вещи с позиции администратора ресурса. Перво-наперво администратор ресурса должен сообщить администратору процессов, что он берет на себя ответственность за некоторую часть пространства имен путей (то есть *зарегистрироваться*). Затем он должен принимать сообщения от клиентуры и их обрабатывать. Очевидно, не так тут все просто.

Давайте кратко рассмотрим функции, реализуемые администраторами ресурсов, а затем уже углубимся в детали.

### Регистрация префикса

Администратор ресурса должен сообщить администратору процессов, что одно или более имя пути теперь является его *доменом ответственности* — иными словами, что он готов обрабатывать клиентские запросы, относящиеся к этим именам путей.

Администратор последовательного порта способен обрабатывать (допустим, так) четыре последовательных порта. В этом случае он должен зарегистрировать у администратора процесса четыре различных имени пути: `/dev/ser1`, `/dev/ser2`, `/dev/ser3` и `/dev/ser4`. В результате этого в дереве имен путей у администратора процессов появятся еще четыре элемента, по одному на каждый из последовательных портов. Четыре элемента — это неплохо. А что если бы администратор последовательного порта обрабатывал, например, новомодную мультипортовую плату на 256 портов? Регистрация 256 *отдельных* префиксов (то есть от `/dev/ser1` до `/dev/ser256`) привела бы к появлению в дереве имен путей администратора процессов 256 различных элементов! Администратор процессов не оптимизирован для поиска по этому дереву — он предполагает, что элементы в нем, конечно, есть, но не сотни же там этих элементов.

Как правило, не следует регистрировать больше чем несколько дюжин отдельных префиксов, потому что поиск по дереву является линейным. Число 256 портов определенно больше. В таких случаях что мультипортовый администратор ресурсов должен сделать, так это зарегистрировать каталогоподобный префикс, например, `/dev/multiport`. Это займет только один элемент в дереве имен путей. Клиент открывает последовательный порт, скажем, порт 57:

```
fp = fopen("/dev/multiport/57", "w");
```

Администратор процессов разрешает это в четверку ND/PID/CHID/handle для мультипортового администратора; решать, насколько корректен при этом остаток имени («57») — это уже дело самого администратора ресурса. В этом примере, предположив, что часть имени пути после точки монтирования хранится в переменной *path*, администратор ресурса мог бы выполнить проверку *очень* простым способом:

```
devnum = atoi(path);  
if ((devnum <= 0) || (devnum >= 256)) {  
    // Неправильный номер устройства  
} else {  
    // Правильный номер устройства  
}
```

Этот будет однозначно быстрее, чем поиск, выполняемый администратором процессов, — хотя бы потому что администратор процессов по сути своей намного более универсален, чем наш администратор ресурса.

## Обработка сообщений

Как только мы зарегистрировали один или более префиксов, мы должны быть готовы принимать сообщения от клиентов. Это делается «обычным» способом с помощью функции *MsgReceive()*. Существуют менее 30 четко определенных типов сообщений, которые администратор ресурса должен быть способен обработать. Однако, для упрощения обсуждения и реализации их условно делят на две группы:

Сообщения установления соединения (connect messages)

Всегда содержат имя пути; либо являются однократными, либо устанавливают контекст для последующих сообщений ввода/вывода.

Сообщения ввода/вывода (I/O messages)

Всегда базируются на сообщениях установления соединения; выполняют всю последующую работу.

### Сообщения установления соединения

Сообщения установления соединения всегда содержат имя пути. Прекрасным примером функции, генерирующей сообщение установления соединения, является уже не раз упомянутая нами функция *open()*. В этом

случае обработчик (`handle`) сообщения установления соединения устанавливает контекст для последующих сообщений ввода/вывода. (В конце-то концов, после `open()` мы все-таки собираемся делать что-то наподобие `read()`.).

Примером «однократного» сообщения установления соединения может быть сообщение, сгенерированное в результате вызова `rename()`. Здесь никакого контекста не предполагается — обработчик в администраторе ресурса изменяет имя указанного файла на новое, и все.

### ***Сообщения ввода/вывода***

Сообщения ввода/вывода возникают только после соответствующего сообщения установления соединения и ссылаются на установленный им контекст. Как уже упоминалось ранее при обсуждении сообщений установления соединения, идеальный пример — вызов функции `open()`, за которым следует `read()`.

### ***На самом деле групп сообщений три***

Кроме сообщений об установлении соединения и сообщений ввода/вывода, есть еще и «другие» сообщения, которые администратор ресурсов может принимать и обрабатывать. Но поскольку они не являются в полной мере «административными», покамест отложим их обсуждение и вернемся к ним позже.

## Библиотека администратора ресурсов

Прежде чем лезть в глубины организации администраторов ресурсов, познакомимся сначала с библиотекой администратора ресурсов, разработанной QSSL. Отметим, что в действительности эта «библиотека» состоит из нескольких четко различимых частей:

- функции пула потоков (мы обсуждали их в главе «Процессы и потоки», в параграфе «Пулы потоков»);
- интерфейс диспетчеризации;
- функции администратора ресурсов;
- вспомогательные функции POSIX-библиотеки.

При том, что можно было бы, конечно, писать администраторы ресурсов «с нуля» (как это делалось в QNX4), эта овчинка часто не стоит такой выделки.

Просто для демонстрации практичности библиотечного подхода — вот код однопоточной версии администратора «`/dev/null`»:

```
/*
 * resmgr1.c
 *
 * /dev/null на основе библиотеки администратора ресурсов
 */
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int main(int argc, char **argv) {
    dispatch_t          *dpp;
    resmgr_attr_t        resmgr_attr;
    resmgr_context_t     *ctp;
    resmgr_connect_funcs_t connect_func;
    resmgr_io_funcs_t     io_func;
    iofunc_attr_t        attr;
    // Создать структуру диспетчеризации
    if ((dpp = dispatch_create()) == NULL) {
        perror("Ошибка dispatch_create\n");
        exit(EXIT_FAILURE);
    }
```

```

}
// Инициализировать структуры данных
memset(&resmgr_attr, 0, sizeof(resmgr_attr));
resmgr_attr.nparts_max = 1;
resmgr_attr.msg_max_size = 2048;
// Назначить вызовам обработчики по умолчанию
iofunc_func_init(_RESMGR_CONNECT_NFUNCS, &connect_func,
    _RESMGR_IO_NFUNCS, &io_func);
iofunc_attr_init(&attr, S_IFNAM | 0666, 0, 0);
// Зарегистрировать префикс в пространстве имен путей
if (resmgr_attach(dpp, &resmgr_attr,
    "/dev/mynull", _FTYPE_ANY,
    0, &connect_func, &io_func, &attr) == -1) {
    perror("Ошибка resmgr_attach\n");
    exit(EXIT_FAILURE);
}
ctp = resmgr_context_alloc(dpp);
// Ждать сообщений в вечном цикле
while (1) {
    if ((ctp = resmgr_block(ctp)) == NULL) {
        perror("Ошибка resmgr_block\n");
        exit(EXIT_FAILURE);
    }
    resmgr_handler(ctp);
}
}

```

И все! Полнофункциональный администратор ресурса `/dev/null` реализуется всего несколькими вызовами функций!

Если бы пришлось писать аналогичный по функциональности администратор (то есть с поддержкой функций *stat()*, *chown()*, *chmod()*, и т.д.) «с нуля», то вам пришлось бы перелопатить сотни, если не тысячи строк Си-кода.

## Реально все это за вас делает библиотека

Как вариант начального знакомства с библиотекой, давайте посмотрим, что делают вызовы, использованные в администраторе ресурсов `/dev/null`:

*dispatch\_create()*

Создает структуру диспетчеризации; она будет использоваться для блокирования по приему сообщения.

*iofunc\_attr\_init()*

Инициализирует используемую устройством атрибутную запись. Мы обсудим атрибутные записи в подробностях несколько позже, а вкратце так: атрибутная запись содержит информацию об устройстве, и на каждое имя устройства имеется по одной атрибутной записи.

*iofunc\_func\_init()*

Инициализирует две структуры данных, *cfuncs* и *ifuncs*, которые содержат соответственно указатели на функции установления соединения и функции ввода/вывода. Это, пожалуй, самый «магический» вызов, поскольку именно он назначает подпрограммы обработки сообщений, привязывая их к структурам данных. Заметьте, что никакого *кода* обработки сообщений установления соединения или сообщений ввода/вывода, генерируемых функциями *read()*, *stat()* или им подобными, в администраторе нет. Дело в том, что библиотека содержит для всех сообщений готовые POSIX-обработчики по умолчанию, и как раз функция *iofunc\_func\_init()*-то и привязывает их к двум передаваемым ей таблицам.

*resmgr\_attach()*

Создает канал, который администратор ресурса будет использовать для приема сообщений, и говорит администратору процессов, что мы намерены отвечать за «*/dev/null*». Параметров тут много, но к этой головной боли мы вернемся несколько позже. Сейчас же важно отметить, что именно здесь связываются воедино дескриптор диспетчера (*dpp*), имя пути (строка «*/dev/null*») и обработчики функций установления соединения (*cfuncs*) и ввода/вывода (*ifuncs*).

*resmgr\_context\_alloc()*

Выделяет внутренний контекстный блок администратора ресурса. Мы рассмотрим этот блок в подробностях несколько позже, а вкратце — он содержит информацию, относящуюся к обрабатываемому сообщению.

*resmgr\_block()*

Это блокирующий вызов администратора ресурса — функция, с помощью которой мы ожидаем сообщение от клиента.

*resmgr\_handler()*

После того как сообщение от клиента получено, для его обработки вызывается эта функция.

## За кулисами библиотеки

Вы уже видели, что наша программа ответственна за предоставление основного рабочего цикла приема сообщений:

```
while (1) {
    // Здесь ждем сообщения
    if ((ctp = resmgr_block(ctp)) == NULL) {
        perror("Unable to resmgr_block\n");
        exit(EXIT_FAILURE);
    }
    // Обработать сообщение
    resmgr_handler(ctp);
}
```

Это очень удобно, поскольку позволяет вам помещать точки останова на принимающей функции и перехватывать сообщения (например, с помощью отладчика) в процессе работы.

Библиотека осуществляет все магические манипуляции внутри функции *resmgr\_handler()*, потому что это как раз то самое место, где сообщение анализируется и обрабатывается в соответствии с таблицами функций установления соединения и ввода/вывода, о которых мы уже говорили ранее.

В действительности, библиотека состоит из двух взаимодействующих уровней — базового, который обеспечивает «сырую» функциональность администратора ресурсов, и уровня POSIX, который содержит вспомогательные функции POSIX и обработчики по умолчанию. Сейчас мы кратко обрисуем эти два уровня, а затем в разделе «Структура администратора ресурсов» рассмотрим все в подробностях.

### **Базовый уровень**

Самый нижний, базовый, уровень состоит из функций, имена которых начинаются с «*resmgr\_*». Этот класс функций относится к низкоуровневым механизмам функционирования администратора ресурсов.

Здесь я только кратко приведу описание этих функций, которые являются доступными и которые мы будем использовать. Затем я отправлю вас к изучению QSSL документации для получения более подробных сведений об этих функциях.

К функциям базового уровня относятся:

*resmgr\_msgreadv()* и *resmgr\_msgread()*

Считывают данных из адресного пространства клиента при помощи обмена сообщениями.

*resmgr\_msgwritev()* и *resmgr\_msgwrite()*

Записывают данные в адресное пространство клиента при помощи обмена сообщениями.

*resmgr\_open\_bind()*

Связывает контекст с запросом на установление соединения, поступившим от соответствующей клиентской функции. Этот контекст далее будет использоваться функциями ввода/вывода.

*resmgr\_attach()*

Создает канал и связывает воедино имя пути, дескриптор диспетчера, функции установления соединения, функции ввода/вывода и другие параметры. Посылает сообщение администратору процессов для регистрации имени пути (префикса).

*resmgr\_detach()*

Противоположна функции *resmgr\_attach()*. Уничтожает связь между именем пути и администратором ресурса.

*pulse\_attach()*

Связывает код импульса с функцией. Поскольку приема сообщений реализуется библиотекой, это удобный способ «перехватывать управление» для обработки импульсов.

*pulse\_detach()*

Отвязывает код импульса от функции.

В дополнение к функциям, перечисленным выше, есть также множество функций, посвященных интерфейсу диспетчеризации.

Одна функция из вышеупомянутого списка заслуживает особого упоминания — функция *resmgr\_open\_bind()*. Данная функция, когда приходит сообщение установления соединения (обычно это происходит в результате клиентского вызова *open()* или *fopen()*), создает некую контекстную информацию, чтобы она была готова к моменту прихода сообщения ввода/вывода. Почему ее не быт администраторе */dev/null*? Потому что POSIX-функции обработки сообщений, принятые по умолчанию, сами вызывают для нас эту функцию. Если бы мы обрабатывали все сообщения самостоятельно, нам, конечно, пришлось бы вызывать данную функцию.

Функция *resmgr\_open\_bind()* не только формирует контекстный блок для последующих сообщений ввода/вывода, но также инициализирует и другие структуры данных, используемые непосредственно библиотекой администратора ресурсов.

Остальные функции из вышеупомянутого списка достаточно интуитивны, так что отложим их обсуждение до тех пор, пока не дойдем до их явного применения.



## *Уровень POSIX*

Второй уровень библиотеки администратора ресурсов является POSIX-уровнем. Как и в случае с базовым уровнем, вы могли бы написать администратор ресурсов и без использования этих функций, но это отняло бы уйму трудов! Прежде чем обсуждать функции уровня POSIX в подробностях, мы должны рассмотреть ряд структур данных базового уровня, приходящие от клиентуры сообщения, а также общую структуру и сферу ответственности администратора ресурсов.

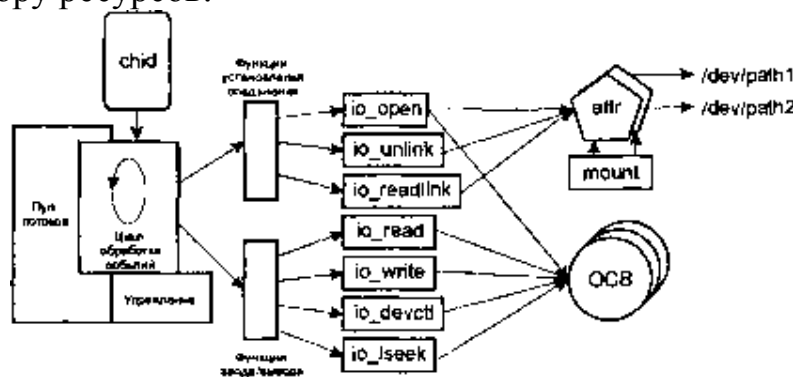
## Написание администратора ресурсов

Теперь, когда мы знаем основы, — как выглядит мир глазами клиента, в каком цвете видит все администратор ресурсов, и что из себя представляют оба уровня библиотеки — пришло время сконцентрироваться на деталях.

В этом разделе мы рассмотрим следующие темы:

- структуры данных;
- структуру администратора ресурсов;
- структуры данных POSIX-уровня;
- подпрограммы обработки сообщений;
- и, конечно, множество примеров.

Постарайтесь запомнить приведенную ниже «большую картинку» — на ней изображено практически все, что имеет отношение к администратору ресурсов:



Архитектура администратора ресурсов — общая схема.

## Структуры данных

Первое, в чем следует разобраться, — это структуры данных, которые управляют работой библиотеки:

- управляющая структура `resmgr_attr_t`
- таблица функций установления соединения `resmgr_connect_funcs_t`
- таблица функций ввода-вывода `resmgr_io_funcs_t` и еще одна внутренняя структура данных библиотеки:
- внутренний блок контекста `resmgr_context_t`

Позже мы рассмотрим такие типы данных как блок открытого контекста (OCB), атрибутную запись (attributes structure) и запись точки

монтирования (mount structure), которые используются POSIX-уровнем библиотеки.

### ***Управляющая структура `resmgr_attr_t`***

Управляющая структура (типа `resmgr_attr_t`) передается функции `resmgr_start()`, которая несколько ее элементов и отвечает за основной цикл приема сообщений.

Управляющая структура (взято из `<sys/dispatch.h>`) содержит следующее:

```
typedef struct _resmgr_attr {
    unsigned flags;
    unsigned nparts_max;
    unsigned msg_max_size;
    int      (*other_func)(resmgr_context_t *ctp, void *msg);
} resmgr_attr_t;
```

### ***Обработчик сообщений `other_func`***

Вообще говоря, использования этого элемента следует избегать. Этот элемент, если не равен NULL, указывает на подпрограмму, которая должна быть вызвана, если принято сообщение, не распознанное библиотекой. Хотя это и можно было бы использовать для реализации «нестандартных» сообщений, но это нехорошая практика (применяйте либо обработчики `_IO_DEVCTL`, либо `_IO_MSG` — см. ниже). Если вы хотите обрабатывать входящие импульсы, рекомендую для этого применять функцию `pulse_attach()`.

Так что оставьте у этого элемента значение NULL.

### ***Параметры, задающие размеры структур данных***

Эти два параметра используются для управления размерами областей памяти, используемых при обмене сообщениями.

Параметр `nparts_max` управляет размером динамически выделяемого вектора ввода/вывода (элемент `iov` в структуре типа `resmgr_context_t` — контекстном блоке библиотеки администратора ресурсов, см. ниже). Обычно этот параметр подстраивают, когда некоторые из функций-

обработчиков возвращают более чем одноэлементный вектор ввода-вывода (IOV). Отметим, что этот параметр применяется только к исходящим сообщениям на поступающие сообщения не влияет.

Параметр *msg\_max\_size* управляет размером буферного пространства, которое библиотека администратора ресурсов должна выделить под входящее сообщение. Библиотека администратора ресурсов установит этот параметр в значение как минимум соответствующее наибольшему заголовку принимаемого сообщения. Это гарантирует, что когда будет вызвана функция-обработчик, ей будет передан полный заголовок сообщения. Отметим, однако, что присутствие в буфере следующих за заголовком данных (если таковые имеются) не гарантируется, даже если параметр *msg\_max\_size* задан достаточно большим. (Размеры буферов обсуждаются в параграфе «Внутренний контекстный блок *resmgr\_context\_t*», см. ниже).

### *Параметр **flags***

Этот параметр дает библиотеке администратора ресурсов дополнительную информацию. В нашем случае мы передадим просто нуль (0). Другие значения этого параметра можно найти в справочном руководстве по Си-библиотеке, в разделе, посвященном функции *resmgr\_attach()*.

### *Таблица функций установления соединения **resmgr\_connect\_funcs\_t***

Когда библиотека администратора ресурсов принимает сообщение, она проверяет тип сообщения и смотрит, что можно сделать. Базовый уровень библиотеки содержит две таблицы, которые определяют это поведение. Это таблица типа *resmgr\_connect\_funcs\_t*, которая содержит список обработчиков сообщений установления соединения, а также таблица типа *resmgr\_io\_funcs\_t*, которая содержит аналогичный список обработчиков сообщений ввода/вывода — ее мы рассмотрим несколько позже.

Когда придет время заполнить таблицы функций установления соединения и ввода/вывода, рекомендуется сначала воспользоваться функцией *iofunc\_func\_init()*, чтобы инициализировать таблицы функциями по умолчанию, определенными на уровне POSIX. Тогда, если вам потребуется заменить обработчик какого-либо сообщения, вы просто подставляете вместо POSIX-обработчика по умолчанию свою собственную

функцию. Мы увидим это в разделе «Подстановка своих собственных функций». А сейчас давайте рассмотрим собственно таблицу функций установления соединения (взято из `<sys/resmgr.h>`):

```
typedef struct _resmgr_connect_funcs {
    unsigned nfuncs;
    int (*open)(ctp, io_open_t *msg, handle, void *extra);
    int (*unlink)(ctp, io_unlink_t *msg, handle,
        void *reserved);
    int (*rename)(ctp, io_rename_t *msg, handle,
        io_rename_extra_t *extra);
    int (*mknod)(ctp, io_mknod_t *msg, handle,
        void *reserved);
    int (*readlink)(ctp, io_readlink_t *msg, handle,
        void *reserved);
    int (*link)(ctp, io_link_t *msg, handle,
        io_link_extra_t *extra);
    int (*unblock)(ctp, io_pulse_t *msg, handle,
        void *reserved);
    int (*mount)(ctp, io_mount_t *msg, handle,
        io_mount_extra_t *extra);
} resmgr_connect_funcs_t;
```

Заметьте, что я сократил прототипы, опустив тип первого параметра, *ctp* (`resmgr_context_t*`), и третьего, *handle* (`RESMGR_HANDLE_T`).

Полный прототип для, например, функции *open()* в действительности имеет вид:

```
int (*open)(resmgr_context_t *ctp, io_open_t *msg,
    RESMGR_HANDLE_T *handle, void *extra);
```

Первый элемент структуры (*nfuncs*) указывает, насколько она велика (то есть сколько в ней содержится элементов). Для приведенной выше структуры он должен быть равен 8, поскольку элементов в ней восемь (от *open()* до *mount()*). Этот элемент нужен главным образом для того, чтобы позволить QSSL обновлять данную библиотеку без каких бы то ни было вредных последствий для вашего кода. Предположим, к примеру, что вы компилировали свой код со значением 8, а затем QSSL обновила библиотеку, и параметр стал равен 9. Библиотека могла бы себе сказать: «Ага! Пользователь библиотеки был скомпилирован с расчетом на 8 функций, а их у нас теперь 9. Надо бы назначить 9-й функции обработчик по умолчанию.» Текущее значение параметра *nfuncs* хранится в заголовочном файле `<sys/resmgr.h>` в виде именованной константы `_RESMGR_CONNECT_NFUNCS`. Используйте эту константу при

заполнении таблицы функций установления соединения вручную (хотя *лучше всего* применять для этого функцию *iofunc\_func\_init()*).

Отметим, что формат у всех прототипов один и тот же. Первый параметр, *ctp*, указывает на структуру **resmgr\_context\_t**. Это внутренний контекстный блок, используемый библиотекой администратора ресурсов и который изменять не следует (за исключением одного поля, к обсуждению которого мы еще вернемся).

Второй параметр всегда указывает на сообщение. Поскольку функции в таблице предназначены для обработки различных типов сообщений, тип второго параметра в прототипе соответствует типу сообщения, которое данная функция должна обрабатывать.

Третий параметр — структура типа **RESMGR\_HANDLE\_T**, называемая дескриптором (*handle*). Она используется для идентификации устройства, которому предназначалось сообщение. Мы тоже рассмотрим ее позже, когда будем говорить об атрибутной записи.

И, наконец, последний параметр является «резервным», или «дополнительным», и используется для функций, которым необходимы какие-либо дополнительные данные. Мы продемонстрируем применение параметра *extra* по назначению в обсуждении функций-обработчиков сообщений.

<b>Таблица функций ввода/вывода <i>resmgr_io_funcs_t</i></b>
--

Таблица функций ввода/вывода подобна таблице функций установления соединения. Вот она (взято из **<sys/resmgr.h>**):

```
typedef struct _resmgr_io_funcs {
    unsigned nfuncs;
    int (*read)(ctp, io_read_t *msg, ocb);
    int (*write)(ctp, io_write_t *msg, ocb);
    int (*close_ocr)(ctp, void *reserved, ocb);
    int (*stat)(ctp, io_stat_t *msg, ocb);
    int (*notify)(ctp, io_notify_t *msg, ocb);
    int (*devctl)(ctp, io_devctl_t *msg, ocb);
    int (*unblock)(ctp, io_pulse_t *msg, ocb);
    int (*pathconf)(ctp, io_pathconf_t *msg, ocb);
    int (*lseek)(ctp, io_lseek_t *msg, ocb);
    int (*chmod)(ctp, io_chmod_t *msg, ocb);
    int (*chown)(ctp, io_chown_t *msg, ocb);
    int (*utime)(ctp, io_utime_t *msg, ocb);
};
```

```

int (*openfd)(ctp, io_openfd_t *msg, ocb);
int (*fdinfo)(ctp, io_fdinfo_t *msg, ocb);
int (*lock)(ctp, io_lock_t *msg, ocb);
int (*space)(ctp, io_space_t *msg, ocb);
int (*shutdown)(ctp, io_shutdown_t *msg, ocb);
int (*mmap)(ctp, io_mmap_t *msg, ocb);
int (*msg)(ctp, io_msg_t *msg, ocb);
int (*umount)(ctp, void *msg, ocb);
int (*dup)(ctp, io_dup_t *msg, ocb);
int (*close_dup)(ctp, io_close_t *msg, ocb);
int (*lock_ocrb)(ctp, void *reserved, ocb);
int (*unlock_ocrb)(ctp, void *reserved, ocb);
int (*sync)(ctp, io_sync_t *msg, ocb);
} resmgr_io_funcs_t;

```

В этой структуре я тоже сократил прототипы, опустив тип элемента *ctp* (**resmgr\_context\_t\***) и тип последнего элемента, *ocrb* (**RESMGR\_OCB\_T\***). Полный прототип, например, для функции *read()* в действительности имеет вид:

```

int (*read)(resmgr_context_t *ctp, io_read_t *msg, RESMGR_OCB_T
*ocrb);

```

Самый первый элемент структуры (*nfuns*) указывает, насколько она велика (то есть сколько элементов она содержит). Текущее значение этого элемента содержится в константе **\_RESMGR\_IO\_NFUNCS**.

Отметим, что списки параметров в таблице функций ввода/вывода также довольно однообразны. Первый параметр — *ctp*, второй параметр — *msg*, как и у обработчиков из таблицы функций установления соединения.

Третий параметр, однако, отличается. Этот параметр называется *ocrb*, что расшифровывается как «Open Context Block» — «блок открытого контекста». Этот блок содержит контекст, созданный обработчиком сообщения установления соединения (например, в результате клиентского запроса *open()*) и доступный функциям ввода/вывода.

Как уже упоминалось ранее, когда придет время заполнять таблицы функций, рекомендуется пользоваться для этого функцией *iofunc\_func\_init()*, чтобы сначала загрузить таблицы POSIX-обработчиками по умолчанию. Если же вам будет нужно переопределить обработчики сообщений определенного типа, вы сможете просто заменить POSIX-обработчики по умолчанию на свои собственные. Мы рассмотрим это в разделе «Подстановка своих собственных функций».

## Внутренний контекстный блок `resmgr_context_t`

И, наконец, еще одна структура данных используется базовым уровнем библиотеки, чтобы отслеживать кое-какую информацию *для себя*. Вам не следует изменять содержимое этой структуры, за исключением одного элемента — вектора ввода/вывода *iov*.

Вот эта структура данных (взято из `<sys/resmgr.h>`):

```
typedef struct _resmgr_context {
    int                rcvid;
    struct _msg_info    info;
    resmgr_iomsgs_t    *msg;
    struct _resmgr_ctrl *ctrl;
    int                id;
    int                status;
    int                offset;
    int                size;
    iov_t              iov[1];
} resmgr_context_t;
```

Как и в случае с другими структурами данных, я позволил себе опустить зарезервированные поля.

Давайте взглянем на ее содержимое.

Идентификатор отправителя, полученный от *MsgReceivev()*.

*rcvid* Указывает, кому вы должны ответить (если вы намерены отвечать самостоятельно).

*info* Содержит информационную структуру, возвращаемую функцией *MsgReceivev()* в основном цикле приема сообщений библиотеки администратора ресурсов. Полезна для получения информации о клиенте, включая дескриптор узла, идентификатор процесса (PID), идентификатор потока и т.д. Подробнее см. документацию по функции *MsgReceivev()*.

*msg* Указатель на объединение (union) всех возможных типов сообщений. Практически бесполезен, потому что каждая из ваших функций-обработчиков получает соответствующий элемент объединения вторым параметром.

*ctrl* Указатель на управляющую структуру, которую вы передали в самом начале. Опять же, для вас этот параметр не очень полезен, но зато полезен для библиотеки администратора ресурсов.

*id* Идентификатор точки монтирования, которой предназначалось



сообщение. Когда вы вызывали *resmgr\_attach()*, она вернула вам небольшой целочисленный идентификатор. Это и есть значение *id*. Отметим, что вы вероятнее всего никогда не будете использовать этот параметр самостоятельно, а будете полагаться вместо этого на атрибутную запись, передаваемую вам обработчиком *io\_open()*.

*status* Сюда ваша функция-обработчик помещает результат выполнения операции. Отметим, что вы должны всегда использовать макрос *\_RESMGR\_STATUS()* для заполнения этого поля. Например, если вы обрабатываете сообщение установления соединения от функции *open()*, причем ваш администратор ресурса предоставляет доступ «только для чтения», а клиент хотел открыть ресурс на запись, вы возвратите клиенту через *errno* код EROFS при помощи (обычно) *\_RESMGR\_STATUS(ctp, EROFS)*.

*offset* Текущее смещение (в байтах) в клиентском буфере сообщений. Имеет смысл для базового уровня библиотеки только при чтении составных сообщений функцией *resmgr\_msgreadv()*.

*size* Этот параметр говорит, сколько байт в буфере сообщения, переданном вашей функции-обработчику, являются достоверными. Это важная цифра, поскольку она указывает на то, требуется ли читать дополнительные данные от клиента (например, если не все данные были считаны базовым уровнем библиотеки), и надо ли выделить память для ответа клиенту (например, для ответа на запрос *read()*). (Отметим, что в версии 2.00 есть ошибка, из-за которой это поле *не заполняется* в случае несоставного сообщения установления соединения. Все остальные сообщения обрабатываются корректно. Обходной путь здесь (и только здесь!) заключается в использовании параметра *msglen* структуры *info*.)

*iov* Таблица векторов ввода/вывода, в которую вы можете записывать возвращаемые значения, если это необходимо. Например, когда клиент вызывает *read()*, и у вас вызывается соответствующий обработчик *read()*, вам может потребоваться вернуть клиенту данные. Можно задать эти данные при помощи массива *iov* и вернуть что-нибудь типа *\_RESMGR\_NPARTS(2)*, указав тем самым (в нашем случае), векторы *iov[0]* и *iov[1]* содержат данные для клиента. Заметьте, что массив *iov* определен как одноэлементный. Однако, заметьте также, что он очень удобно расположен в конце структуры. Фактическое число элементов в массиве *iov* определяете вы сами, когда присваиваете значение полю *nparts\_max*

вышеупомянутой управляющей структуры (см. параграф «Управляющая структура `resmgr_attr_t`»).

## Структура администратора ресурсов

Теперь, когда мы имеем представление о структурах данных, мы можем обсудить взаимодействие между компонентами, которые вам предстоит написать, чтобы ваш администратор ресурсов мог что-нибудь реально *сделать*.

Мы рассмотрим:

- Функцию `resmgr_attach()` и ее параметры;
- Подстановку своих собственных функций;
- Общую схему работы администратора ресурсов;
- Сообщения, которые должны бы быть сообщениями установления соединения, но таковыми не являются;
- Составные сообщения.

### Функция `resmgr_attach()` и ее параметры

Как вы уже видели в приведенном выше примере с администратором `/dev/null`, первое, что вы должны сделать — это зарегистрировать у администратора процессов свою точку монтирования. Это делается с помощью функции `resmgr_attach()`, которая имеет следующий прототип:

```
int resmgr_attach(void *dpp, resmgr_attr_t *resmgr_attr,
    const char *path, enum _file_type file_type,
    unsigned flags,
    const resmgr_connect_funcs_t *connect_funcs,
    const resmgr_io_funcs_t *io_funcs,
    RESMGR_HANDLE_T *handle);
```

Давайте исследуем по порядку ее аргументы и посмотрим, как они применяются.

<i>dpp</i>	Дескриптор диспетчера (dispatch handle). Обеспечивает интерфейсу диспетчеризации возможность управлять приемом сообщений для вашего администратора ресурсов.
<i>resmgr_attr</i>	Управляет характеристиками администратора ресурсов, как обсуждалось ранее.
<i>path</i>	Точка монтирования, которую вы регистрируете. Если вы регистрируете дискретную точку монтирования (как,

например, в случае с `/dev/null` или `/dev/ser1`), клиент должен указывать ее точно, без каких бы то ни было дополнительных компонентов имени пути в ее конце. Если вы регистрируете каталоговую точку монтирования (как было бы, например, в случае с сетевой файловой системой, монтируемой как `/nfs`), то соответствие тоже должно быть точным, но с той оговоркой, что в этом случае продолжение имени пути допускается; то, что идет после точки монтирования, будет передано функции установления соединения (например, имя пути `/nfs/etc/passwd` даст совпадение с точкой монтирования сетевой файловой системой, а «остатком» будет `etc/passwd`). (Эта особенность, кстати, может пригодиться и там, где на первый взгляд логичнее было бы регистрировать дискретную точку монтирования — см. параграф «Регистрация префикса» раздела «Взгляд со стороны администратора ресурсов» — *прим. ред.*)

*file\_type*

Класс администратора ресурсов. См. ниже.

*flags*

Дополнительные флаги, управляющие поведением вашего администратора ресурсов. Эти флаги выбираются из множества `_RESMGR_FLAG_BEFORE`, `_RESMGR_FLAG_AFTER`, `_RESMGR_FLAG_DIR` и константы 0. Флаги «BEFORE» (букв, «перед») и «AFTER» (букв, «после») указывают на то, что ваш администратор ресурсов хочет зарегистрироваться на данной точке монтирования перед или, соответственно, после других. Эти два флага могут быть полезны, если надо реализовать объединенные файловые системы. Мы вскоре вернемся к взаимосвязи этих флагов. Флаг «DIR.» («каталог») указывает на то, что ваш администратор ресурса хочет обслуживать указанную точку монтирования и все, что находится ниже ее — этот стиль характерен для администратора файловой системы, в противоположность администратору ресурсов, регистрирующему дискретную точку монтирования.

*connect\_funcs*  
и *io\_funcs*

Эти параметры являются просто списком функций установления соединения и функций ввода/вывода, которые вы хотите привязать к точке монтирования.

*handle*

Это «расширяемая» структура (также известная как «атрибутная запись»), описывающая монтируемый ресурс.

Например, в случае последовательного порта вы могли бы расширить стандартную атрибутную запись POSIX-уровня информацией о базовом адресе последовательного порта, скорости обмена в бодах, и т.д.

Вы можете вызывать функцию *resmgr\_attach()* столько раз, сколько вам захочется зарегистрировать различных точек монтирования. Вы также можете вызывать функцию *resmgr\_attach()* из тела функций установления соединения или ввода/вывода — эта аккуратная особенность позволяет вам «создавать» устройства «на лету».

Когда вы определились с точкой монтирования и хотите ее зарегистрировать, вы должны сообщить администратору процессов, хочет ли ваш администратор ресурсов обрабатывать запросы от кого попало или только от клиентуры, которая пометает свои сообщения установления соединения специальными метками. Например, рассмотрим драйвер очередей сообщений POSIX (*mqueue*). Ему совершенно ни к чему «обычные» вызовы *open()* от старых добрых клиентов — он просто не будет знать, что с ними делать. Он примет сообщения только от тех клиентов, которые используют POSIX-вызовы *mq\_open()*, *mq\_receive()*, и т.п. Чтобы не позволять администратору процессов даже перенаправлять «обычные» запросы администратору очередей *mqueue*, у этого администратора в параметре параметр *file\_type* задается значение *\_TYPE\_MQUEUE*. Это означает, что когда клиент пытается с помощью администратора процессов выполнить разрешение имени, при этом явно не определив, что хочет поговорить с администратором ресурсов, зарегистрированным как *\_FTYPE\_MQUEUE*, администратор процессов не будет даже рассматривать администратор *mqueue* как возможный вариант.

Если только вы не делаете что-либо уж очень специфичное, вам лучше всего подойдет значение *file\_type*, равное *\_FTYPE\_ANY*, означающее, что ваш администратор ресурсов готов обработать запрос от любого клиента. Полный список именованных констант *\_FTYPE\_\** приведен в файле *<sys/ftype.h>*.

Что касательно флагов «BEFORE» и «AFTER», тут все становится интереснее. Вы можете задать либо один из этих флагов, либо константу 0.

Давайте посмотрим, как это работает. Стартуют несколько администраторов ресурсов, в указанном в таблице порядке. В таблице также приведены флаги, указанные каждым из них в параметре *flags*. Взгляните на получившуюся очередность.

Администратор	Флаг	Очередность
1	<i>_RESMGR_BEFORE</i>	1

2	<code>_RESMGR_AFTER</code>	1, 2
3	0	1, 3, 2
4	<code>_RESMGR_BEFORE</code>	1, 4, 3, 2
5	<code>_RESMGR_AFTER</code>	1, 4, 3, 5, 2
6	0	1, 4, 6, 3, 5, 2

Из таблицы видно, что первый администратор ресурса, явно определивший флаг, далее не сдвигается со своей позиции. (См. таблицу: администратор ресурсов № 1 был первым определившим флаг «BEFORE»; кто бы теперь ни зарегистрировался, он так и останется первым в списке. Аналогично, администратор ресурсов № 2 был первым определившим флаг «AFTER» — и снова, независимо от того, кто еще будет регистрироваться после него, он всегда остается в списке последним.) Если не определен никакой флаг, это действует как флаг «между». Когда стартует администратор ресурсов № 3 (указав нулевой флаг), он помещается в середину очереди. Как и в случае с флагами «BEFORE» и «AFTER», здесь имеет место упорядочивание, в результате чего все вновь регистрирующиеся «средние» администраторы ресурсов располагаются перед уже зарегистрированными «средними».

Однако, в действительности, только в очень редких случаях вам придется регистрировать более одного, и в еще меньшем числе случаев — более двух администраторов ресурсов при той же самой точке монтирования. Полезный совет: обеспечьте возможность установки флагов непосредственно в командной строке администратора ресурса, чтобы конечный пользователь вашего администратора ресурса мог сам задать, например, флаг «BEFORE» опцией `-b`, флаг «AFTER» — опцией `-a`, а нулевой флаг («между») был бы, скажем, установкой по умолчанию.

Имейте в виду, что данное обсуждение применимо только для администраторов ресурсов, регистрируемых при одной и той же точке монтирования. Монтирование «`/nfs`» с флагом «BEFORE» и «`/disk2`» с флагом «AFTER» не будет иметь никакого взаимного влияния. Однако, если вы затем будете монтировать еще одну «`/nfs`» или «`/disk2`», вот тогда эти флаги и проявят себя.

И наконец, функция `resmgr_attach()` в случае успешного завершения возвращает дескриптор (handle) в виде небольшого целого числа (или -1 при неудаче). Этот дескриптор можно затем применить для того, чтобы убрать данное имя пути из внутренней таблицы имен путей администратора процессов.

## Подстановка своих собственных функций

Проектируя свой самый первый администратор ресурсов вы, скорее всего, захотите действовать постепенно. Было бы очень досадно написать несколько тысяч строк кода только для того, чтобы понять, что в самом начале была допущена фундаментальная ошибка, и теперь придется либо наспех затыкать дырки (э-э, я хотел сказать — вносить коррективы), либо выкинуть все это и начать заново.

Чтобы все работало как надо, рекомендуемым подходом здесь является использование функции-инициализатора *iofunc\_func\_init()* из уровня POSIX, чтобы заполнить таблицы функций установления соединения и функций ввода/вывода заданными по умолчанию функциями POSIX-уровня. Это значит, что вы можете фактически написать каркас вашего администратора ресурсов, как мы уже делали выше, с помощью всего нескольких вызовов.

Какую функцию запрограммировать первой — это будет зависеть от того, какой администратор ресурсов вы пишете. Если это администратор файловой системы, отвечающий за точку монтирования и все, что под ней, то вам, скорее всего, лучше всего начать с функции *io\_open()*. С другой стороны, если вы пишете администратор ресурса с дискретной точкой монтирования, который выполняет «традиционные» операции ввода/вывода (то есть вы будете общаться с ним преимущественно вызовами типа *read()* и *write()*), то лучшей стартовой позицией для вас были бы функции *io\_read()* и/или *io\_write()*. Если же вы пишете администратор ресурса с дискретной точкой монтирования, но вместо «традиционных» операций ввода/вывода основу его функциональности составляют вызовы типа *devctl()* или *ioctl()*, то правильнее было бы начать с *io\_devctl()*.

Независимо от того, с чего вы начинаете, вам нужно будет удостовериться в том, что ваши функции вызываются так, как вы предполагаете. В данном ключе функции-обработчики POSIX-уровня по умолчанию обладают очень полезным свойством — их можно помещать непосредственно в таблицы функций установления соединения и таблицы функций ввода/вывода.

Это означает, что если вы захотите что-то дополнительно проконтролировать, просто добавьте дополнительный диагностический вызов *printf()*, чтобы он сказал что-то типа «Я тут!», а затем делайте «то, что надо сделать» — все очень просто.

Вот фрагмент администратора ресурсов, который перехватывает функцию *io\_open()*:

```
// Упреждающая декларация
int io_open(resmgr_context_t*, io_open_t*,
    RESMGR_HANDLE_T*, void*);

int main() {
    // Все как в примере /dev/null,
    // кроме следующего за этой строкой:
    iofunc_func_init(_RESMGR_CONNECT_NFUNCS, &cfuncs,
        _RESMGR_IO_NFUNCS, &ifuncs);
    // Добавьте это для перехвата управления:
    cfuncs.open = io_open;
```

Если вы описали функцию *io\_open()* корректно, как в этом примере кода, то вы можете вызывать функцию, заданную по умолчанию, из вашей собственной!

```
int io_open(resmgr_context_t *ctp, io_open_t *msg,
    RESMGR_HANDLE_T *handle, void *extra) {
    printf("Мы в io_open!\n");
    return (iofunc_open_default(ctp, msg, handle, extra));
}
```

Таким образом, вы по-прежнему применяете POSIX-обработчик по умолчанию *iofunc\_open\_default()*, но заодно перехватываете управление для вызова *printf()*.

Очевидно, что вы могли бы выполнить аналогичные действия для функций *io\_read()*, *io\_write()*, *io\_devctl()* и любых других, для которых есть обработчики POSIX-уровня по умолчанию. Идея, кстати, действительно отличная, потому что такой подход показывает вам, что клиент вызывает ваш администратор ресурса именно так, как вы предполагаете.

### Общая схема работы администратора ресурсов

Как мы уже намекнули выше в разделах, посвященных краткому рассмотрению клиента и администратора ресурсов, последовательность действий начинается на клиентской стороне с вызова *open()*. Он транслируется в сообщение установления соединения, которое принимается и обрабатывается функцией администратора ресурсов *io\_open()*.

Это действительно ключевой момент, потому что функция *io\_open()* выполняет для вашего администратора ресурсов функцию «швейцара». Если «швейцар» посмотрит на сообщение и отклонит запрос, вы *не*

*получите* никаких запросов на ввод/вывод, потому что у клиента не будет корректного дескриптора файла. И наоборот, если «швейцар» пропустит сообщение, тогда клиент получит корректный дескриптор файла, и логично будет ожидать от него сообщений ввода/вывода.

Но на самом деле роль функции *io\_open()* гораздо значительнее. Она отвечает не только за проверку, может клиент открыть ресурс или нет, но также за следующее:

- инициализацию внутренних параметров библиотеки;
- привязку к запросу контекстного блока;
- привязку к контекстному блоку атрибутной записи.

Первые две операции выполняются с помощью функции базового уровня *resmgr\_open\_bind()*, а привязка атрибутной записи сводится к простому присваиванию.

Будучи однажды вызвана, *io\_open()* выпадает из рассмотрения. Клиент может либо прислать сообщение ввода/вывода, либо нет, но в любом случае должен будет однажды завершить «сеанс связи» с помощью сообщения, соответствующего функции *close()*. Заметьте, что если клиента вдруг постигает внезапная смерть (например, он получает SIGSEGV, или выходит из строя узел, на котором он работает), операционная система автоматически синтезирует сообщение *close()*, чтобы администратор ресурсов смог корректно завершить сессию. Поэтому вы *гарантированно* получите сообщение *close()*!

***Сообщения, которые должны быть сообщениями установления соединения, но таковыми не являются***

Тут есть один интересный момент, который вы, может быть, для себя уже отметили. Прототип клиентской функции *chown()* имеет вид:

```
int chown(const char *path, uid_t owner, gid_t group);
```

Вспомните: сообщение об установлении соединения всегда содержит имя пути и является либо однократным, либо устанавливает контекст для дальнейших сообщений ввода/вывода.

Так почему же сообщение, соответствующее клиентской функции *chown()*, не является сообщением установления соединения? К чему здесь сообщение ввода/вывода, когда в прототипе даже дескриптора файла нет?!

Ответ простой — чтобы облегчить вам жизнь.

Представьте себе, что было бы, если бы функции типа *chown()*, *chmod()*, *stat()* и им подобные требовали от администратора ресурсов, чтобы он сначала анализировал имя пути, а затем уже выполнял нужные



действия. (Именно так, кстати, все реализовано в QNX4.) Типичные проблемы этого подхода:

- Каждой функции приходится вызывать процедуру поиска.
- Для функций, у которых есть также версия, ориентированная на файловый дескриптор, драйвер должен обеспечить две отдельные точки входа: одну для версии с именем пути, и еще одну — версии с дескриптором файла.

В QNX/Neutrino же происходит следующее. Клиент создает *составное сообщение* — реально это одно сообщение, но оно включает в себя несколько сообщений администратору ресурсов. Без составных сообщений мы могли бы смоделировать функцию *chown()* чем-то таким:

```
int chown(const char *path, uid_t owner, gid_t group) {
    int fd, sts;
    if ((fd = open(path, O_RDWR)) == -1) {
        return (-1);
    }
    sts = fchown(fd, owner, group);
    close(fd);
    return (sts);
}
```

где функция *fchown()* — это версия функции *chown()*, ориентированная на файловые дескрипторы. Проблема здесь в том, что мы в этом случае используем три вызова функций (а значит, и три отдельных транзакции передачи сообщений) и привносим дополнительные накладные расходы применением функций *open()* и *close()* на стороне клиента.

При использовании составных сообщений в QNX/Neutrino непосредственно клиентским вызовом *chown()* создается одиночное сообщение, выглядящее примерно так:

__IO_CONNECT_COMBINE_CLOSE	__IO_CHOWN
----------------------------	------------

Составное сообщение.

Сообщение состоит из двух частей. Первая часть посвящена установлению соединения (подобно сообщению, которое сгенерировала бы функция *open()*), вторая — вводу/выводу (эквивалент сообщения, генерируемого функцией *fchown()*). Никакого эквивалента функции *close()* здесь нет, поскольку мы выбрали сообщение типа `__IO_CONNECT_COMBINE_CLOSE`, которое гласит: «Открой указанное имя пути, используй полученный дескриптор файла для обработки остальной части сообщения, а когда закончишь дела или столкнешься с ошибкой, закрой дескриптор».

Написанный вами администратор ресурса даже не заметит, вызвал ли клиент функцию *chown()* или сначала сделал *open()*, а потом вызвал *fchown()* и далее *close()*. Все это скрыто базовым уровнем библиотеки.

### Составные сообщения

Как выясняется, концепция составных сообщений полезна не только для экономии ресурсов вследствие уменьшения числа сообщений (как в случае с *chown()*, см. выше). Она также критически важна для обеспечения атомарности операций.

Предположим, что в клиентском процессе есть два или более потоков, работающих с одним дескриптором файла. Один из потоков в клиенте вызывает функцию *lseek()*, за которой следует *read()*. Все так, как мы и предполагаем. А вот если другой клиента попытается выполнить ту же самую последовательность операций с тем же самым дескриптором файла, вот тут у нас начнутся проблемы. Поскольку функции *lseek()* и *read()* друг о друге ничего не знают, то возможно, например, что первый поток выполнит *lseek()*, а затем будет вытеснен вторым потоком. Второй поток выполнит свои *lseek()* и *read()*, после чего освободит процессор. Проблема здесь состоит в том, что поскольку эти два потока разделяют один и тот же дескриптор файла, у первого потока теперь получается неправильное смещение *lseek()*, поскольку оно было изменено функциями *lseek()* и *read()* второго потока! Эта проблема проявляется также с дескрипторами файлов, которые дублируются (*dup()*) между процессами, не говоря уже о сети.

Очевидным решением здесь является заключение *lseek()* и *read()* в пределы действия мутекса — когда первый поток захватит мутекс, мы будем знать, что он имеет эксклюзивный доступ к дескриптору. Второй поток должен будет ждать освобождения мутекса, прежде чем он сможет творить свое безобразие с позиционированием дескриптора.

К сожалению, если кто-то вдруг забудет захватить мутекс *перед каждой и любой операцией с дескриптором файла*, всегда остается возможность того, что такой «незащищенный» доступ может привести поток к чтению или записи данных не туда, куда надо.

Давайте взглянем на библиотечный вызов *readblock()* (из `<unistd.h>`):

```
int readblock(int fd, size_t blksize, unsigned block,
              int numblks, void *buff);
```

(Функция *writeblock()* описывается аналогично.)

Вы можете вообразить для функции *readblock()* довольно «простенькую» реализацию:

```
int readblock(int fd, size_t blksize, unsigned block,
int numblks, void *buff) {
lseek(fd, blksize * block, SEEK_SET); // Идем к блоку
read(fd, buff, blksize * numblks);
}
```

Очевидно, что от такой реализации в многопоточной среде толку мало. Нам нужно будет как минимум добавить использование мутекса:

```
int readblock(int fd, size_t blksize, unsigned block,
int numblks, void *buff) {
pthread_mutex_lock(&block_mutex);
lseek(fd, blksize * block, SEEK_SET); // Идем к блоку
read(fd, buff, blksize * numblks);
pthread_mutex_unlock(&block_mutex);
}
```

(Мы здесь предполагаем, что мутекс уже инициализирован.)

Этот код по-прежнему уязвим для «незащищенного» доступа — если некий поток вызовет *lseek()* на этом файловом дескрипторе без предварительной попытки захвата мутекса, вот у нас уже и ошибка.

Решение проблемы заключается в использовании составного сообщения, аналогично вышеописанному случаю с функцией *chown()*. В данном случае библиотечная реализация функции *readblock()* помещает обе операции — *lseek()* и *read()* — в единое сообщение и посылает это сообщение администратору ресурсов:

<code>_IO_LSEEK</code>	<code>_IO_READ</code>
------------------------	-----------------------

Составное сообщение для функции *readblock()*.

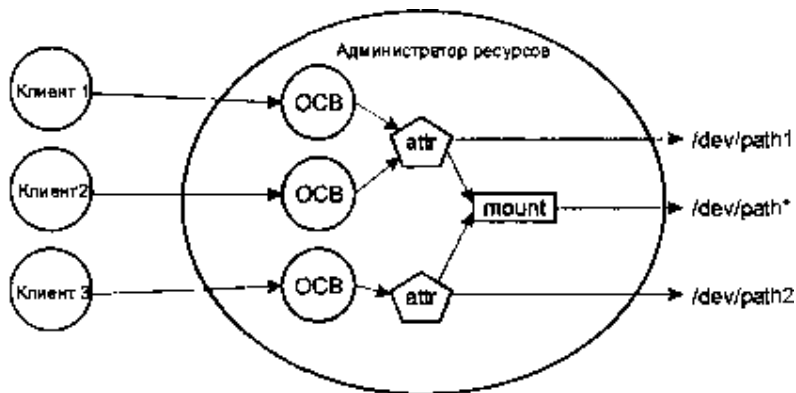
Это работает, потому что передача сообщения является атомарной операцией. С точки зрения клиента, сообщение уходит либо целиком, либо не уходит вообще. Поэтому, вмешательство «незащищенной» функции *lseek()* становится несущественным — когда администратор ресурсов принимает сообщение с запросом *readblock()*, он делает это за один прием. (Очевидно, что в этом случае пострадает сама «незащищенная» *lseek()*, поскольку после отработки *readblock()* смещение на этом дескрипторе файла будет отличаться от того, которое она хотела установить.)

А как насчет самого администратора ресурсов? Как он *обрабатывает* операцию *readblock()* за один прием? Мы вскоре рассмотрим это, когда будем обсуждать операции, выполняемые для каждого компонента составных сообщений.

## Структуры данных уровня POSIX

К подпрограммам POSIX-уровня относятся три структуры данных. Отметьте, что пока речь идет о *базовом уровне*, вы можете использовать любые структуры данных, которые пожелаете; соответствия определенной структуре и содержанию требует именно уровень POSIX. Однако, преимущества, которые предоставляет POSIX-уровень, с лихвой окупают вносимые ограничения. Как мы увидим далее, вы также сможете дополнять эти структуры вашим собственным содержанием.

На рисунке приведены эти три структуры данных для случая, когда несколько клиентов используют администратора ресурсов, объявивший два устройства:



Структуры данных — общая схема.

Структурами данных являются:

`iofunc_ocr_t` — OCB (блок открытого контекста)

Содержит информацию по каждому дескриптору файла.

`iofunc_attr_t` — атрибутная запись

Содержит информацию по каждому устройству.

`iofunc_mount_t` — запись точки монтирования

Содержит информацию по каждой точке монтирования.

Когда мы обсуждали таблицы функций установления соединения и ввода/вывода, мы уже видели блоки открытого контекста и атрибутные записи — в таблицах функций ввода/вывода OCB был последним передаваемым параметром. Атрибутная запись передавалась как параметр *handle* (третий по счету) в функциях установления соединения. Запись точки монтирования обычно представляет собой глобальную структуру и привязывается к атрибутной записи «вручную» (в инициализационном коде, написанном вами для вашего администратора ресурса).

**Структура блока открытого контекста (OCB) `iofunc_ocr_t`**

Структура блока открытого контекста (ОСВ) содержит информацию по каждому дескриптору файла. Это означает, что когда клиент выполняет вызов *open()* и получает в ответ дескриптор файла (в противоположность коду ошибки), администратор ресурсов создает ОСВ и связывает его с данным клиентом. Этот ОСВ будет существовать до тех пор, пока клиент держит данный дескриптор файла открытым. В действительности, ОСВ и дескриптор файла — всегда согласованная пара. По каждому сообщению ввода/вывода от клиента библиотека администратора ресурсов автоматически ищет нужный ОСВ и вместе с сообщением передает его нужной функции из таблицы функций ввода/вывода. Это ответ на вопрос, зачем всем функциям ввода/вывода параметр *ocb*. В конце концов клиент закрывает дескриптор файла (применив *close()*), что заставит администратора ресурса отвязать ОСВ от дескриптора файла и от клиента. Заметьте, что клиентская функция *dup()* просто увеличивает счетчик связей. В этом случае ОСВ отделяется от дескриптора файла и от клиента только тогда, когда значение счетчика связей достигнет нуля (то есть когда число вызовов *close()* будет соответствовать числу *open()* и *dup()*).

Как вы, наверное, догадываетесь, в ОСВ содержатся важные вещи по каждому открытию ресурса и по каждому дескриптору файла. Вот его содержание (взято из **<sys/iofunc.h>**):

```
typedef struct _iofunc_ocb {
    IOFUNC_ATTR_T *attr;
    int32_t        ioflag;
    CM НИЖЕ!!!     offset;
    uint16_t       sflag;
    uint16_t       flags;
} iofunc_ocb_t;
```

Проигнорируем пока комментарий относительно поля *offset*; мы вернемся к этому вопросу сразу же после данного обсуждения.

Поля структуры **iofunc\_ocb\_t**:

Указатель на атрибутную запись, связанную с данным блоком ОСВ.

*attr* В функциях ввода/вывода вы будете встречать устоявшуюся идиому «*ocb->attr*»; она используется для получения доступа к элементам атрибутной записи.

*ioflag* Режим открытия, то есть как был открыт ресурс (например, «только для чтения»). Заметьте, что поле *ioflag* содержит режим открытия (который был передан клиентской функции *open()*) плюс единица. Например, режим открытия **O\_RDONLY** (значение 0) появится в поле *ioflag*, как значение, равное единице (1) (константа **\_READ** из

`<stdio.h>`). Это позволяет трактовать два младших бита поля `ioflag` как флаги разрешения чтения и записи (`ioflag & _READ` указывает на право доступа по чтению; `ioflag & _WRITE` — по записи).

*offset* Текущее смещение `lseek()` в данном ресурсе.

*sflag* Флаг разделяемого использования (см. `<share.h>`), используемый с клиентской функцией вызова `sopen()`. Возможны значения `SH_COMPAT`, `SH_DENYRW`, `SH_DENYWR`, `SH_DENYRD`, и `SH_DENYN`

*flags* Системные флаги. В настоящее время поддерживаются два флага: `IOFUNC_OCB_PRIVILEGED`, указывающий на то, что этот OCB был создан в результате сообщения установления соединения от привилегированного процесса, и `IOFUNC_OCB_MMAP`, указывающий, используется ли этот OCB функцией `mmmap()` на стороне клиента. На настоящий момент никаких других флагов не определено. Вы можете использовать биты, заданные в `IOFUNC_OCB_FLAGS_PRIVATE`, по своему собственному усмотрению.

Если вы хотите наряду со «стандартным» OCB сохранить какие-либо дополнительные данные, то будьте покойны — OCB можно «расширять». Мы обсудим это в разделе «Дополнительно».

### *Это странное поле offset*

Поле *offset*, скажем так, как минимум любопытно. Посмотрите в `<sys/iofunc.h>`, как оно реализовано. В зависимости от того, какие у вас заданы флаги препроцессора, вы можете получить одну из шести (!) возможных раскладок поля *offset*. Но не беспокойтесь особо по поводу реализации — реально есть смысл рассматривать только два случая, в зависимости от того, хотите вы поддерживать 64-разрядные смещения или нет:

- если да, то поле *offset* 64-разрядное;
- если нет (у вас 32-разрядные целые), то поле *offset* — это младшие 32 бита; старшие 32 бита хранятся в поле *offset\_hi*.

Для наших целей, если речь не идет о явном противопоставлении 32- и 64-разрядных значений, мы будем предполагать, что все смещения являются 64-разрядными (типа `off_t`), и платформа знает, что делать с 64-разрядными числами.

## Атрибутная запись `iofunc_attr_t`

В то время как ОСВ был определен как структура данных по каждому дескриптору файла, атрибутная запись является структурой данных по каждому устройству. Вы видели, что стандартный ОСВ типа `iofunc_ocr_t` имеет элемент, называемый *attr*, который представляет собой указатель на атрибутную запись. Это сделано для того, чтобы у ОСВ был доступ к информации об устройстве. Давайте посмотрим на атрибутную запись (взято из `<sys/iofunc.h>`):

```
typedef struct _iofunc_attr {
    IOFUNC_MOUNT_T      *mount;
    uint32_t             flags;
    int32_t              lock_tid;
    uint16_t             lock_count;
    uint16_t             count;
    uint16_t             rcount;
    uint16_t             wcount;
    uint16_t             rlocks;
    uint16_t             wlocks;
    struct _iofunc_mmap_list *mmap_list;
    struct _iofunc_lock_list *lock_list;
    void                 *list;
    uint32_t             list_size;
    CM_НИЖЕ!!!           nbytes;
    CM_НИЖЕ!!!           inode;
    uid_t                uid;
    gid_t                gid;
    time_t               mtime;
    time_t               atime;
    time_t               ctime;
    mode_t               mode;
    nlink_t              nlink;
    dev_t                rdev;
} iofunc_attr_t;
```

У полей *nbytes* и *inode* такой же набор директив условной компиляции, что и у поля *offset* в ОСВ (см. параграф «Это странное поле *offset*»).

Заметьте, что некоторые из полей атрибутной записи полезны только для вспомогательных функций POSIX.

Давайте рассмотрим поля в индивидуальном порядке:

<i>mount</i>	<p>Указатель на необязательную запись точки монтирования (типа <code>iofunc_mount_t</code>). Он применяется аналогично указателю на атрибутную запись, входящему в состав в ОСВ, за исключением того, что здесь это поле может принимать NULL — в этом случае для записи точки монтирования применяются установки по умолчанию (см. ниже параграф «Запись точки монтирования <code>iofunc_mount_t</code>»). Как уже было упомянуто, запись точки монтирования привязывается к атрибутной записи «вручную» в инициализационном коде вашего администратора ресурсов.</p>
<i>flags</i>	<p>Содержит флаги, которые описывают состояние других полей атрибутной записи. Мы вскоре к ним вернемся.</p> <p>Для предупреждения проблем синхронизации доступ множества потоков к одной и той же атрибутной записи должен</p>
<i>lock_tid</i>	<p>быть взаимно исключающим. Поле <i>lock_tid</i> содержит идентификатор потока (thread ID), которым данная атрибутная запись заблокирована в настоящий момент.</p> <p>Указывает, сколько потоков пытаются использовать данную атрибутную запись. Нулевое значение указывает на то, что</p>
<i>lock_count</i>	<p>структура не заблокирована. Значение, большее нуля (единица или более) указывает на то, что данную структуру используют один или более потоков.</p> <p>Указывает на число ОСВ, которые по какой-либо причине открыли эту атрибутную запись. Например, если у одного клиента есть ОСВ, открытый на чтение, у другого — другой</p>
<i>count</i>	<p>ОСВ, открытый на чтение/запись, и оба эти ОСВ указывают на одну и ту же атрибутную запись, то значение <i>count</i> для нее должно быть равно 2. Это будет указывать на то, что данный ресурс открыт двумя клиентами.</p> <p>Число читателей. В примере, приведенном для <i>count</i>, <i>rcount</i></p>
<i>rcount</i>	<p>будет также иметь значение 2, потому что ресурс открыт на чтение двумя клиентами.</p> <p>Число писателей. В примере, приведенном для <i>count</i>, <i>wcount</i></p>
<i>wcount</i>	<p>будет иметь значение 1, потому что ресурс открыт на чтение только одним клиентом.</p> <p>Показывает число ОСВ, наложивших на данный ресурс блокировки по чтению. Если значение этого поля равно нулю, это означает, что никаких блокировок по чтению нет, но могут быть блокировки по записи.</p>
<i>rlocks</i>	



<i>wlocks</i>	Аналогично <i>rlocks</i> , только для блокировок по записи.
<i>mmap_list</i>	Для внутреннего использования POSIX-функцией <i>iofunc_mmap_default()</i> .
<i>lock_list</i>	Для внутреннего использования POSIX-функцией <i>iofunc_lock_default()</i> .
<i>list</i>	Зарезервировано.
<i>list_size</i>	Размер области, зарезервированной под поле <i>list</i> .
<i>nbytes</i>	Размер ресурса в байтах. Например, если ресурс описывает конкретный файл, и этот файл имеет размер 7756 байт, то поле <i>nbytes</i> будет содержать значение 7756.
<i>inode</i>	Содержит порядковый номер файла или ресурса; он должен быть уникален для каждой точки монтирования. Значение поля <i>inode</i> никогда не должно быть нулевым, потому что нуль указывает на неиспользуемый файл.
<i>uid</i>	Идентификатор пользователя владельца данного ресурса.
<i>gid</i>	Идентификатор группы владельца данного ресурса.
<i>mtime</i>	Время последней модификации файла, обновленное или как минимум ставшее недействительным вследствие обработки клиентской функции <i>write()</i> .
<i>atime</i>	Время последнего доступа к файлу, обновленное или как минимум ставшее недействительным вследствие обработки клиентской функции <i>read()</i> , возвратившей ненулевое количество прочитанных байт.
<i>ctime</i>	Время последнего изменения файла, обновленное или как минимум ставшее недействительным вследствие обработки клиентских функций <i>write()</i> , <i>chown()</i> или <i>chmod()</i> .
<i>mode</i>	Режим доступа к файлу. Содержит стандартные значения <i>S_*</i> из <code>&lt;sys/stat.h&gt;</code> (например, <i>S_IFCHR</i> ), или восьмеричные значения (например, 0664), указывающие на режим доступа для владельца объекта ( <i>owner</i> ), группы ( <i>group</i> ) и всех остальных ( <i>other</i> ).
<i>nlink</i>	Число связей (линков) файла, возвращаемое клиентским вызовом <i>stat()</i> .
<i>rdev</i>	Для специальных символьных устройств это поле состоит из старшего ( <i>major</i> ) и младшего ( <i>minor</i> ) кодов устройства (10 младших бит — младший код, старшие 6 бит — старший). Для устройств другого типа это поле содержит номер устройства

(подробности см. ниже в параграфе «О номерах устройств, индексных дескрипторах и нашем друге *rdev*»).

Как и в случае с ОСВ, вы можете расширять «стандартную» атрибутивную запись вашими собственными данными — см. раздел «Дополнительно».

### *Запись точки монтирования `iofunc_mount_t`*

Запись точки монтирования содержит информацию, общую для нескольких атрибутивных записей.

Вот содержимое записи точки монтирования (взято из `<sys/iofunc.h>`):

```
typedef struct _iofunc_mount {
    uint32_t flags;
    uint32_t conf;
    dev_t      dev;
    int32_t     blocksize;
    iofunc_funcs_t *funcs;
} iofunc_mount_t;
```

Поле *flags* содержит только один флаг — `IOFUNC_MOUNT_32BIT`. Этот флаг указывает на то, что параметр *offset* в ОСВ и параметры *nbytes* и *inode* в атрибутивной записи являются 32-разрядными. Заметьте, что вы можете определять ваши собственные флаги в поле *flags*, используя биты, определенные константой `IOFUNC_MOUNT_FLAGS_PRIVATE`.

Поле *conf* содержит следующие флаги:

`IOFUNC_PC_CHOWN_RESTRICTED`

Указывает, что файловая система является «chown-ограниченной», то есть никто, кроме суперпользователя (*root*), не может применять к файлам операцию *chown()*.

`IOFUNC_PC_NO_TRUNC`

Указывает на то, что файловая система не выполняет усечение имен.

`IOFUNC_PC_SYNC_IO`

Указывает на то, что файловая система поддерживает синхронные операции ввода-вывода.

`IOFUNC_PC_LINK_DIR`

Указывает на то, что допускается создание/уничтожение связей (*linking/unlinking*) для каталогов.

Поле *dev* содержит номер устройства и описывается ниже в параграфе «О номерах устройств, индексных дескрипторах и нашем друге *rdev*».

Поле *blocksize* описывает типовой для данного устройства размер блока в байтах. Например, для дисковых устройств типовым значением будет 512.

И наконец, поле *funcs* указывает на следующую структуру (взято из `<sys/iofunc.h>`):

```
typedef struct _iofunc_funcs {
    unsigned nfuncs;
    IOFUNC_OCB_T *(*ocb_alloc) (
        resmgr_context_t *ctp, IOFUNC_ATTR_T *attr);
    void (*ocb_free) (IOFUNC_OCB_T *ocb);
} iofunc_funcs_t;
```

Как и в таблицах функций установления соединения и ввода/вывода, поле *nfuncs* должен содержать текущий размер таблицы. Используйте для этого константу `_IOFUNC_NFUNCS`.

Указатели на функции *ocb\_alloc* и *ocb\_free* могут быть заполнены адресами функций, которые следует вызывать всякий раз при создании и уничтожении ОСВ. Зачем вам могут понадобиться эти функции — мы обсудим это чуть позже, когда будем говорить о расширении ОСВ.

### ***О номерах устройств, индексных дескрипторах и нашем друге *rdev****

Запись точки монтирования содержит поле с именем *dev*. Атрибутная запись содержит два поля: *inode* и *rdev*. Давайте рассмотрим их взаимосвязь на примере традиционной дисковой файловой системы. Файловая система монтируется на блок-ориентированном устройстве (которое представляет собой весь диск целиком). Это блок-ориентированное устройство может называться, скажем, `/dev/hd0` (первый жесткий диск в системе). На этом диске может быть несколько разделов, один из которых вполне может называться `/dev/hd0t77` (первый раздел файловой системы QNX на этом конкретном устройстве). И, наконец, на этом разделе может находиться произвольное число файлов, один из которых может иметь имя `/hd/spud.txt`.

Поле *dev* («device number» — «номер устройства») содержит число, уникальное для узла, на котором зарегистрирован данный администратор ресурсов. Поле *rdev* — значение *dev* для корневого устройства (root device). И, наконец, поле *inode* — порядковый номер файла.

Попробуем соотнести это с нашим примером дисковой системы. В приведенной ниже таблице приведен ряд чисел; взглянем на таблицу, а потом посмотрим, откуда появились эти номера и как они соотносятся.

**Устройство dev inode rdev**

/dev/hd0	6	2	1
/dev/hd0t77	1	12	77
/hd/spud.txt	77	47343	-

Для «сырого» блок-ориентированного устройства */dev/hd0* значения *dev* и *inode* назначил администратор процессов (значения 6 и 2 в таблице, см. выше). Администратор ресурсов при старте получил для устройства уникальное значение *rdev* (число 1).

Для раздела */dev/hd0t77* значение *dev* взялось из значения *rdev* «сырого» блок-ориентированного устройства (та самая 1). Значение *inode* было выбрано администратором ресурсов как уникальное в пределах *rdev*. Так появилась цифра 12. Наконец, значение *rdev* было также выбрано администратором ресурсов — здесь разработчик администратора выбрал значение 77, потому что оно соответствует типу раздела.

И наконец, для файла */hd/spud.txt*, значение *dev* (77) было взято из значения *rdev* для раздела. Значение *inode* было выбрано администратором ресурса (в случае файла номер выбирается так, чтобы соответствовать некоторому внутреннему представлению файла — конкретное число не имеет значения, лишь бы оно было ненулевым и уникальным в пределах *rdev*). Отсюда число 47343. Для файла поле *rdev* не имеет значения.

## Функции-обработчики

Не все вызовы обработчиков соответствуют клиентским сообщениям. Некоторые из них синтезируются ядром, а некоторые — библиотекой.

Я организовал этот раздел следующим образом:

- общие замечания;
- замечания о функциях установления соединения;
- алфавитный список сообщений установления соединения и ввода/вывода.

### Общие замечания

Каждой функции-обработчику передается внутренний контекстный блок (параметр *ctp*), который следует рассматривать как «только для чтения», за исключением поля *iov*. Как это уже упоминалось в параграфе «Внутренний контекстный блок `resmgr_context_t`», этот контекстный блок содержит несколько интересных вещей. Также, каждой функции передается указатель на сообщение (аргумент *msg*). Вы будете активно использовать этот указатель, поскольку он содержит параметры, которыми его для вас заполнил клиентский библиотечный вызов. Функция, которую вы пишете, должна возвращать некоторое значение (все функции описаны как возвращающие `int`).

Значения выбираются из следующего списка:

`_RESMGR_NOREPLY`

Указывает библиотеке администратора ресурсов, что она *не должна* выполнять *MsgReplyv()* — в предположении, что вы либо уже сделали это самостоятельно в вашей функции-обработчике, либо собираетесь сделать это несколько позже.

`_RESMGR_NPARTS(n)`

Указывает библиотеке администратора ресурсов при выполнении *MsgReplyv()* вернуть *n*-элементный вектор ввода/вывода (он располагается в `ctp->iov`). Ваша функция ответственна за заполнение поля *iov* структуры *ctp* и возврат `_RESMGR_NPARTS` с корректным числом элементов.

☞ Память под поле *iov* структуры *ctp* выделяется динамически, и ее должно быть достаточно, чтобы вместить столько число элементов массива, сколько вы записываете в *iov*!

Детали о настройке поля *nparts\_max* см. выше в разделе «Управляющая структура *resmgr\_attr\_t*».

### \_RESMGR\_DEFAULT

Это говорит библиотеке администратора ресурсов выполнить *низкоуровневую функцию по умолчанию* (это другое семейство функций; не путайте их с *iofunc\_\*\_default()*!). Это возвращаемое значение вам вряд ли когда-нибудь пригодится. В общем случае оно заставляет библиотеку администратора ресурсов вернуть клиенту значение *errno*, равное *ENOSYS*, что означает «функция не поддерживается».

### \_RESMGR\_ERRNO(errno)

(Устаревшее.) Данное возвращаемое значение использовалось для «инкапсуляции» значения *errno* в возвращаемое сообщением значение. Например, если бы клиент выдал запрос *open()* (по записи — прим. ред.) устройству, доступному только для чтения, корректно было бы вернуть код ошибки *EROFS*. Поскольку данный способ сделать это считается устаревшим, вы можете вернуть код ошибки непосредственно (например, при помощи `return (EROFS);` вместо громоздкого `_RESMGR_ERRNO(EROFS);`).

### \_RESMGR\_PTR(ctp, addr, len)

Это макрос для удобства. Он берет указатель на контекст *ctp* и заполняет его первый элемент IOV адресом *addr* и длиной *len*, а затем возвращает библиотеке эквивалент `_RESMGR_NPARTS(1)`. Это может быть полезно для функций, возвращающих одноэлементные IOV.

## ***Блокировки, разблокировки и обработка составных сообщений***

Мы видели клиентский взгляд на составные сообщения, когда рассматривали функцию *readblock()* (в параграфе «Составные сообщения»). Клиент мог атомарно создать сообщение, которое содержало бы несколько «подсообщений» администратору ресурсов — в нашем примере это были сообщения, соответствующие функциям *lseek()* и *read()*. С точки зрения клиента две (или более) функций были как минимум атомарно *переданы* (и, вследствие самой сути обмена сообщениями, будут атомарно *приняты* администратором ресурсов). О чем мы еще не говорили, так это о том, как мы сможем гарантированно обеспечить атомарность *обработки* этих сообщений.

Данные рассуждения применимы не только к составным сообщениям, но и ко *всем* сообщениям, принимаемым библиотекой администратора

ресурсов. Первое, что делает библиотека администратора ресурсов, — она блокирует атрибутную запись, соответствующую ресурсу, используемому полученным сообщением. Затем она обрабатывает одно или более «подсообщений», содержащихся в полученном сообщении. Затем она снова разблокирует атрибутную запись.

Это гарантирует, что поступающие сообщения обрабатываются атомарно, поскольку никакой другой поток администратора ресурсов (в случае многопоточного администратора, конечно) не может «влезть» и изменить ресурс, пока наш поток этот ресурс использует. Без блокировок два клиентских потока могли бы оба выдать то, что, по их мнению, являлось бы атомарным составным сообщением (скажем, пару «*lseek()* — *read()*»). Поскольку администратор ресурсов мог выделить на обработку этих сообщений два различных потока, эти потоки могли бы в произвольном порядке вытеснять друг друга, и их *lseek()* могли бы друг другу помешать. Блокировки же позволяют это предотвратить, потому что каждое сообщение, получающее доступ к ресурсу, обрабатывается целиком и атомарно.

Блокировка и разблокировка ресурса выполняются вспомогательными функциями по умолчанию (*iofunc\_lock\_ocb\_default()* и *iofunc\_unlock\_ocb\_default()*), которые размещаются в таблице функций ввода/вывода в полях *lock\_ocb* и *unlock\_ocb* соответственно. Вы можете, конечно, переназначить эти функции, если хотите выполнить в процессе блокировки и разблокировки какие-либо дополнительные действия.

Заметьте, что ресурс разблокируется *перед* вызовом *io\_close()*. Это необходимо, поскольку функция *io\_close()* освободит ОСВ, что автоматически сделает недействительным указатель на атрибутную запись, а блокировка хранится именно там!

## Замечания о функциях установления соединения

Перед тем как углубиться в отдельные сообщения, однако, есть смысл подчеркнуть, что для всех функции установления соединения структура сообщений идентична (взято из `<sys/iomsg.h>`, с небольшими изменениями):

```
struct _io_connect {
    // Для внутреннего использования
    uint16_t type;
    uint16_t subtype;
    uint32_t file_type;
```

```

uint16_t reply_max;
uint16_t entry_max;
uint32_t key;
uint32_t handle;
uint32_t ioflag;
uint32_t mode;
uint16_t sflag;
uint16_t access;
uint16_t zero;
uint8_t  eflag;
// Для конечного пользователя
uint16_t path_len;
uint8_t  extra_type;
uint16_t extra_len;
char path[1];
};

```

Вы заметите, что я разделил структуру `struct _io_connect` на две части, часть «Для внутреннего использования» и часть «Для конечного пользователя».

### ***Поля для внутреннего использования***

Первая часть состоит из полей, которые библиотека администратора ресурсов использует для:

- определения типа сообщения, полученного от клиента;
- проверки сообщения на достоверность (не является ли оно дезинформацией);
- отслеживания режима доступа (используется вспомогательными функциями).

Для простоты я бы рекомендовал вам *всегда* применять вспомогательные функции (из семейства `iofunc_*_default()`) во *всех* функциях установления соединения. Эти функции возвратят вам признак успешного/неудачного завершения, после чего вы сможете использовать в функции установления соединения «поля для конечного пользователя».

### ***Поля для конечного пользователя***



Вторая половина полей непосредственно относится к вашей реализации функции установления соединения:

*path\_len* и *path*

Имя пути (и его длина), которые являются *операндом* (то есть это имя пути, над которым производится действие).

*extra\_type* и *extra\_len*

Дополнительные параметры (имена путей, например), соответствующее данной функции установления соединения.

Чтобы получить представление о том, как поле *path* используется в качестве «имени пути, над которым производится действие», давайте рассмотрим что-нибудь типа функции *rename()*. Эта функция принимает два имени пути: «изначальное» имя пути и «новое» имя пути. Изначальное имя пути передается в *path*, потому что именно над ним производится операция (это имя файла, подлежащего переименованию). Новое имя пути — аргумент данной операции. Вы увидите, что параметр *extra*, который передается функции установления соединения, как раз содержит указатель на аргумент операции, в данном случае — на новое имя пути.

(В принятой реализации новое имя пути располагается в памяти непосредственно следом за изначальным (на которое указывает *path*) с учетом выравнивания, но вам делать на этот счет ничего не надо — параметр *extra* уже содержит правильный указатель.)

## Алфавитный список функций установления соединения и ввода/вывода

В данном разделе в алфавитном порядке приведен список точек входа в функции установления соединения и ввода/вывода, которые вы можете заполнять самостоятельно (эти две таблицы затем передаются функции *pathname\_attach()*). Помните, что если вы просто вызываете функцию *iofunc\_func\_init()*, все эти точки входа будут заполнены соответствующими значениями по умолчанию; вам следует переопределять конкретные точки входа только в том случае, если вы собираетесь обрабатывать данное конкретное сообщение самостоятельно. Ниже, в разделе «Примеры», вы увидите несколько примеров общеупотребительных функций.

☞ Поначалу все это может показаться излишне запутанным, но отметим, что на самом деле существуют *два* разблокирующих вызова — один для функций установления соединения и один для функций ввода/вывода. Все вполне корректно, поскольку

отражает то, когда происходит разблокирование. Вариант для функций установления соединения применяется тогда, когда ядро разблокирует клиента немедленно после отправки им сообщения установления соединения. Вариант для функций ввода/вывода применяется тогда, когда ядро разблокирует клиента немедленно после отправки им сообщения ввода/вывода.

Чтобы не путать точки входа в функции-обработчики сообщений с вызовами клиентской Си-библиотеки (например, *open()*), к именам всех приведенных здесь функций добавлен префикс «*io\_*». Например, обработчик функции установления соединения *open()* будет называться *io\_open()*.

### *io\_chmod()*

```
int io_chmod(resmgr_context_t *ctp, io_chmod_t *msg,  
    RESMGR_OCB_T *ocb)
```

Классификация: Функция ввода/вывода

Обработчик по умолчанию: *iofunc\_chmod\_default()*

Вспомогательные функции: *iofunc\_chmod()*

Клиентская функция: *chmod()*, *fchmod()*

Сообщения: *\_IO\_CHMOD*

Структура данных:

```
struct _io_chmod {  
    uint16_t type;  
    uint16_t combine_len;  
    mode_t    mode;  
};
```

```
typedef union {  
    struct _io_chmod i;  
} io_chmod_t;
```

Описание: Отвечает за изменение режима доступа к ресурсу, указанному в переданном ей параметре *ocb*, в значение, содержащееся в поле сообщения *mode*.

Возвращает: Код завершения, при помощи вспомогательного макроса *\_RESMGR\_STATUS*.

## *io\_chown()*

```
int io_chown(resmgr_context_t *ctp, io_chown_t *msg,  
             RESMGR_OCB_T *ocb)
```

Классификация: Функция ввода/вывода

Обработчик по умолчанию: *iofunc\_chown\_default()*

Вспомогательные функции: *iofunc\_chown()*

Клиентская функция: *chown()*, *fchown()*

Сообщения: `_IO_CHOWN`

Структура данных:

```
struct _io_chown {  
    uint16_t type;  
    uint16_t combine_len;  
    int32_t  gid;  
    int32_t  uid;  
};
```

```
typedef union {  
    struct _io_chown i;  
} io_chown_t;
```

Описание: Ответственна за изменение полей идентификатора пользователя и группы для ресурса, указанному в переданном ей параметре *ocb*, соответственно в значения *uid* и *gid*. Отметим, что чтобы узнать, позволяет ли данная файловая система выполнять *chown()* кому-либо, кроме суперпользователя (*root*), надо проверить запись точки монтирования на предмет флага `IOFUNC_PC_CHOWN_RESTRICTED`, а также поле *flags* в *OCB*.

Возвращает: Код завершения, при помощи вспомогательного макроса `_RESMGR_STATUS`.

## *io\_close\_dup()*

```
int io_close_dup(resmgr_context_t *ctp, io_close_t *msg,  
                 RESMGR_OCB_T *ocb)
```

Классификация: Функция ввода/вывода

Обработчик по умолчанию: *iofunc\_close\_dup\_default()*

Вспомогательные функции: *iofunc\_close\_dup()*

Клиентская функция: *close()*, *fclose()*

Сообщения: `_IO_CLOSE_DUP`

Структура данных:

```
struct _io_close {
    uint16_t type;
    uint16_t combine_len;
};
```

```
typedef union {
    struct _io_close i;
} io_close_t;
```

Описание: Это *реальный* обработчик клиентских вызовов `close()` и `fclose()`. Отметим, что вам почти никогда не придется переназначать эту функцию; оставляйте в таблице функций ввода/вывода значение `iofunc_close_dup_default()`. Причиной этому служит то, что базовый уровень библиотеки отслеживает число сообщений `open()`, `dup()` и `close()`, выданных по каждому ОСВ, и синтезирует вызов `io_close_ocb()` (см. ниже) после получения для данного ОСВ *последнего* сообщения `close()`. Отметим, что идентификаторы отправителей, расположенные в `ctp->rcvid`, могут и не совпадать с переданными функции `io_open()`; однако, совпадение по меньшей мере одного идентификатора гарантируется. «Лишние» идентификаторы отправителей являются результатом (возможно, внутренних) вызовов типа `dup()`.

Возвращает: Код завершения, при помощи вспомогательного макроса `_RESMGR_STATUS`.

### `io_close_ocb()`

```
int io_close_ocb(resmgr_context_t *ctp, void*reserved,
    RESMGR_OCB_T *ocb)
```

Классификация: Функция ввода/вывода (синтезируется библиотекой)

Обработчик по умолчанию: `iofunc_close_default()`

Вспомогательные функции: Нет

Клиентская функция: Нет (синтезируется библиотекой)

Сообщения: Нет (синтезируется библиотекой)

Структура данных:

```
// Синтезируется библиотекой
struct _io_close {
    uint16_t type;
    uint16_t combine_len;
```

```
};
```

```
typedef union {  
    struct _io_close i;  
} io_close_t;
```

Описание: Это функция, которая синтезируется базовым уровнем библиотеки, когда для некоего ОСВ получено последнее сообщение *close()*. Это то самое место, где вам следует «подчистить» все перед уничтожением ОСВ. Отметим, что идентификатор отправителя в *ctp->rcvid* есть нуль (0), потому что данная функция синтезируется библиотекой и не обязательно соответствует какому-либо конкретному сообщению.

Возвращает: Код завершения, при помощи вспомогательного макроса *\_RESMGR\_STATUS*.

### *io\_devctl()*

```
int io_devctl(resmgr_context_t *ctp, io_devctl_t *msg,  
    RESMGR_OCB_T *ocb)
```

Классификация: Функция ввода/вывода

Обработчик по умолчанию: *iofunc\_devctl\_default()*

Вспомогательные функции: *iofunc\_devctl()*

Клиентская функция: *devctl()*, *ioctl()*

Сообщения: *\_IO\_DEVCTL*

Структура данных:

```
struct _io_devctl {  
    uint16_t type;  
    uint16_t combine_len;  
    int32_t dcmd;  
    int32_t nbytes;  
    int32_t zero;  
};
```

```
struct _io_devctl_reply {  
    uint32_t zero;  
    int32_t ret_val;  
    int32_t nbytes;  
    int32_t zero2;  
};
```

```
typedef union {
    struct _io_devctl      i;
    struct _io_devctl_reply o;
} io_devctl_t;
```

Описание: Выполняет над устройством операцию ввода/вывода, переданную от клиентской функции *devctl()* в параметре *dcmd*. Клиент кодирует направление передачи данных двумя старшими разрядами *dcmd*, указывая этим, как функция *devctl()* должна передавать данные (поле «to» соответствует биту `_POSIX_DEVDIR_TO`, поле «from» — биту `_POSIX_DEVDIR_FROM`):

**Поле «to» Поле «from» Значение**

0	0	Передачи данных нет
0	1	Передача от драйвера клиенту
1	0	Передача от клиента драйверу
1	1	Двунаправленная передача

В случае, когда передачи данных нет, предполагается, что драйвер просто выполняет команду, заданную в *dcmd*. В случае передачи данных предполагается, что драйвер передает данные клиенту и/или обратно, используя вспомогательные функции *resmgr\_msgreadv()* и *resmgr\_msgwritev()*. Клиент указывает размер передачи в поле *nbytes*; драйвер должен установить число передаваемых байт в поле *nbytes* исходящей структуры.

Отметим, что структуры данных, предназначенные для ввода и вывода, дополнены нулями, чтобы быть выровненными друг относительно друга. Это означает, что неявная область данных начинается в этих структурах с того же самого адреса.

Если вы используете вспомогательную функцию *iofunc\_devctl()*, то имейте в виду, что если она не сможет сделать что-либо с сообщением *devctl()*, она возвратит вам константу `_RESMGR_DEFAULT`. Эта сделано для отделения корректных значений *errno* от возвращаемого признака «нераспознанная команда». Получив `_RESMGR_DEFAULT`, базовый уровень библиотеки ответит установкой *errno* в значение `ENOSYS`, которое будет транслировано клиентской библиотечной функцией *devctl()* в значение `ENOTTY`, «корректное» с точки зрения POSIX.

Проверка режима открытия и сопоставление его с выполняемой операцией лежит всецело на совести вашей функции — ни в клиентской *devctl()*, ни в библиотеке администратора ресурсов никаких проверок не выполняется. Например, можно открыть администратор ресурса в режиме «только для чтения», а затем выдать ему посредством *devctl()* команду

«отформатировать жесткий диск» (которая, в общем, является весьма нехилой операцией записи). Так вот, с точки зрения администратора было бы весьма предусмотрительно до выполнения такой операции сначала проверить режим открытия ресурса.

Отметим, что диапазон доступных пользователю значений *dcmd* ограничен (значения от 0x0000 до 0x0FFF включительно зарезервированы QSSL). Другие значения можно смело использовать — см. заголовочные файлы с именами `<sys/dcmd_*.h>`.

Возвращает: Код завершения, при помощи вспомогательного макроса `_RESMGR_STATUS`, и буфер приема (с ответными данными, если надо).

Для примера см. ниже параграф «Простой пример функции *io\_devctl()*».

### *io\_dup()*

```
int io_dup(resmgr_context_t *ctp, io_dup_t *msg,  
          RESMGR_OCB_T *ocb)
```

Классификация: Функция ввода/вывода

Обработчик по умолчанию: NULL (обрабатывается базовым уровнем)

Вспомогательные функции: Нет

Клиентская функция: *dup()*, *dup2()*, *fcntl()*, *fork()*, *spawn\*()*, *fork()*

Сообщения: `_IO_DUP`

Структура данных:

```
struct _io_dup {  
    uint16_t          type;  
    uint16_t          combine_len;  
    struct _msg_info  info;  
    uint32_t          reserved;  
    uint32_t          key;  
};
```

```
typedef union {  
    struct _io_dup i;  
} io_dup_t;
```

Описание: Это обработчик сообщений *dup()*. Как и в случае с *io\_close\_dup()*, вы вряд ли будете обрабатывать это сообщение самостоятельно. За вас это сделает базовый уровень библиотеки.

Возвращает: Код завершения, при помощи вспомогательного макроса `_RESMGR_STATUS`.

## *io\_fdinfo()*

```
int io_fdinfo(resmgr_context_t *ctp, io_fdinfo_t *msg,  
              RESMGR_OCB_T *ocb)
```

Данная функция зарезервирована QSSL для будущего использования. Вам следует инициализировать таблицу функций ввода/вывода при помощи *iofunc\_func\_init()* и не изменять данную точку входа.

## *io\_link()*

```
int io_link(resmgr_context_t *ctp, io_link_t *msg,  
            RESMGR_HANDLE_T*handle, io_link_extra_t*extra)
```

Классификация: Функция установления соединения

Обработчик по умолчанию: Нет

Вспомогательные функции: *iofunc\_link()*

Клиентская функция: *link()*

Сообщения: `_IO_CONNECT`, подтип `IO_CONNECT_LINK`

Структура данных:

```
struct _io_connect {  
    // Внутренние поля (как описано выше)  
    uint16_t path_len;  
    uint8_t  extra_type;  
    uint16_t extra_len;  
    char     path[1];  
};
```

```
struct _io_connect_link_reply {  
    uint32_t reserved1[2];  
    uint8_t  eflag;  
    uint8_t  reserved2[3];  
    uint32_t umask;  
    uint16_t nentries;  
    uint16_t path_len;  
};
```

```
typedef union {  
    struct _io_connect          connect;  
    struct _io_connect_link_reply link_reply;
```



```

} io_link_t;

typedef union _io_link_extra {
    struct _msg_info          info;
    void                      *ocb;
    char                      path[1];
    struct _io_resmgr_link_extra resmgr;
} io_link_extra_t;

```

Описание: Создает новую связь (линк) с именем, заданным в поле *path* структуры *msg*, к уже существующему имени пути, указанному в поле *path* параметра *extra* (переданного вашей функции). Для удобства поле *ocb* параметра *extra* содержит указатель на ОСВ существующего имени пути.

Возвращает: Код завершения, при помощи вспомогательного макроса *\_RESMGR\_STATUS*.

### *io\_lock()*

```

int io_lock(resmgr_context_t *ctp, io_lock_t *msg,
    RESMGR_OCB_T *ocb)

```

Данная функция зарезервирована QSSL для будущего использования. Вам следует инициализировать таблицу функций ввода/вывода, используя *iofunc\_func\_init()*, и не изменять данную точку входа.

### *io\_lock\_ocb()*

```

int io_lock_ocb(resmgr_context_t *ctp, void *reserved,
    RESMGR_OCB_T *ocb)

```

Классификация: Функция ввода/вывода (синтезируется библиотекой)

Обработчик по умолчанию: *iofunc\_lock\_ocb\_default()*

Вспомогательные функции: Нет

Клиентская функция: Все

Сообщения: Нет (синтезируются библиотекой)

Структура данных: Нет

Описание: Эта функция отвечает за блокировку атрибутной записи, на которую указывает ОСВ. Это сделано для того, чтобы гарантировать одновременный доступ не более одного потока как к самому ОСВ, так и к соответствующей атрибутной записи. Функции блокировки (и соответствующие функции разблокировки) синтезируются библиотекой

администратора ресурсов до начала обработки сообщения и после ее завершения соответственно. Более подробно это описано выше в параграфе «Составные сообщения». Вы почти никогда не будете использовать этот вызов самостоятельно; вместо этого используйте функцию POSIX-уровня по умолчанию.

Возвращает: Код завершения, при помощи вспомогательного макроса `_RESMGR_STATUS`.

### *io\_lseek()*

```
int io_lseek(resmgr_context_t *ctp, io_lseek_t *msg,  
             RESMGR_OCB_T *ocb)
```

Классификация: Функция ввода/вывода

Обработчик по умолчанию: *iofunc\_lseek\_default()*

Вспомогательные функции: *iofunc\_lseek()*

Клиентская функции: *lseek()*, *fseek()*, *rewinddir()*

Сообщения: `_IO_LSEEK`

Структура данных:

```
struct _io_lseek {  
    uint16_t type;  
    uint16_t combine_len;  
    short whence;  
    uint16_t zero;  
    uint64_t offset;  
};
```

```
typedef union {  
    struct _io_lseek i;  
    uint64_t o;  
} io_lseek_t;
```

Описание: Обрабатывает клиентскую функцию *lseek()*. Отметьте, что администратору ресурса, который обрабатывает каталоги, придется также интерпретировать сообщение `_IO_LSEEK` для операций с каталогами. Параметры *whence* и *offset* передаются от клиентской функции *lseek()*. После интерпретации параметров *whence* и *offset* клиентского сообщения подпрограмма должна скорректировать у ОCB параметр *offset* и затем вернуть новое значение *offset* или вернуть признак ошибки.

Возвращает: Код завершения, при помощи вспомогательного макроса `_RESMGR_STATUS`, а также (не обязательно) текущее смещение.

## *io\_mknod()*

```
int io_mknod(resmgr_context_t *ctp, io_mknod_t *msg,  
    RESMGR_HANDLE_T *handle, void *reserved)
```

Классификация: Функция установления соединения

Обработчик по умолчанию: Нет

Вспомогательные функции: *iofunc\_mknod()*

Клиентская функция: *mknod()*, *mkdir()*, *mkfifo()*

Сообщения: `_IO_CONNECT`, подтип `_IO_CONNECT_MKNOD`

Структура данных:

```
struct _io_connect {  
    // Внутренние поля (как описано выше)  
    uint16_t path_len;  
    uint8_t  extra_type;  
    uint16_t extra_len;  
    char     path[1];  
};  
  
struct _io_connect_link_reply {  
    uint32_t reserved1[2];  
    uint8_t  eflag;  
    uint8_t  reserved2[3];  
    uint32_t umask;  
    uint16_t nentries;  
    uint16_t path_len;  
};  
  
typedef union {  
    struct _io_connect          connect;  
    struct _io_connect_link_reply link_reply;  
} io_mknod_t;
```

Описание: Создает новую точку входа в файловую систему. Сообщение выдается для создания файла с именем, указанным в *path*, и типом, закодированным в поле *mode* (оно из «внутренних полей» структуры **struct \_io\_connect** и здесь не показано).

Реально это используется только для клиентских функций *mkfifo()* и *mkdir()*.

Возвращает: Код завершения, при помощи вспомогательного макроса `_RESMGR_STATUS`.

## *io\_mmap()*

```
int io_mmap(resmgr_context_t *ctp, io_mmap_t *msg,  
    RESMGR_OCB_T *ocb)
```

Классификация: Функция ввода/вывода

Обработчик по умолчанию: *iofunc\_mmap\_default()*

Вспомогательные функции: *iofunc\_mmap()*

Клиентская функция: *mmap()*, *munmap()*, *mmap\_device\_io()*,  
*mmap\_device\_memory()*

Сообщения: **\_IO\_MMAP**

Структура данных:

```
struct _io_mmap {  
    uint16_t          type;  
    uint16_t          combine_len;  
    uint32_t          prot;  
    uint64_t          offset;  
    struct _msg_info  info;  
    uint32_t          zero[6];  
};
```

```
struct _io_mmap_reply {  
    uint32_t zero;  
    uint32_t flags;  
    uint64_t offset;  
    int32_t  coid;  
    int32_t  fd;  
};
```

```
typedef union {  
    struct _io_mmap      i;  
    struct _io_mmap_reply o;  
} io_mmap_t;
```

Описание: Позволяет администратору процессов применять к файлам вашего администратора ресурсов операцию *mmap()*. В общем случае самостоятельно программировать эту функцию не следует (используйте обработчик по умолчанию, предоставляемый *iofunc\_func\_init()*), если только вы не хотите ее преднамеренно отключить (например, драйвер последовательного порта мог бы запросто вернуть ENOSYS, поэтому для него эта операция не имеет никакого смысла).

Данную функцию администратора ресурсов может вызвать только администратор процессов

Отметим, что побочным результатом вызова этой функции администратором процессов является создание ОСВ (то есть будет вызвана функция *iofunc\_ocb\_alloc()*, но для правильно спроектированного администратора ресурсов это не должно иметь никаких последствий.

Возвращает: Код завершения, при помощи вспомогательного макроса *\_RESMGR\_STATUS*.

### *io\_mount()*

```
int io_mount(resmgr_context_t *ctp, io_mount_t *msg,  
             RESMGR_HANDLE_T*handle, io_mount_extra_t*extra)
```

Данная функция зарезервирована QSSL для будущего использования. Вам следует инициализировать таблицу функций ввода/вывода, используя *iofunc\_func\_init()*, и не изменять данную точку входа.

### *io\_msg()*

```
int io_msg(resmgr_context_t *ctp, io_msg_t *msg,  
           RESMGR_OCB_T *ocb)
```

Классификация: Функция ввода/вывода

Обработчик по умолчанию: Нет

Вспомогательные функции: Нет

Клиентская функция: Нет (создается «вручную» и передается посредством *MsgSend()*)

Сообщения: *\_IO\_MSG*

Структура данных:

```
struct _io_msg {  
    uint16_t type;  
    uint16_t combine_len;  
    uint16_t mgrid;  
    uint16_t subtype;  
};
```

```
typedef union {  
    struct _io_msg i;  
} io_msg_t;
```

Описание: Интерфейс `_IO_MSG` является более общей, но менее переносимой вариацией на тему `ioctl()` и `devctl()`. Поле `mgrid` идентифицирует конкретный администратор — вы не должны выполнять никаких действий по запросам, не соответствующим идентификатору вашего администратора. Поле `subtype` фактически задает команду, которую клиент хочет выполнить. Любые неявно передаваемые данные следуют за входной структурой. Данные, возвращаемые клиенту, передаются сами по себе; код завершения возвращается через макрос `_RESMGR_STATUS`. Уникальный «идентификатор администратора» (manager ID) вы можете получить в QSSL.

Возвращает: Код завершения, при помощи вспомогательного макроса `_RESMGR_STATUS`.

### *io\_notify()*

```
int io_notify(resmgr_context_t *ctp, io_notify_t *msg,
              RESMGR_OCB_T *ocb)
```

Классификация: Функция ввода/вывода

Обработчик по умолчанию: Нет

Вспомогательные функции: `iofunc_notify()`, `iofunc_notify_remove()`,  
`iofunc_notify_trigger()`

Клиентская функция: `select()`, `ionotify()`

Сообщения: `_IO_NOTIFY`

Структура данных:

```
struct _io_notify {
    uint16_t      type;
    uint16_t      combine_len;
    int32_t       action;
    int32_t       flags;
    struct sigevent event;
};
```

```
struct _io_notify_reply {
    uint32_t zero;
    uint32_t flags;
};
```

```
typedef union {
    struct _io_notify      i;
```

```

    struct _io_notify_reply o;
} io_notify_t;

```

Описание: Данный обработчик отвечает за установку, опрос или удаление обработчика уведомлений. Параметры *action* (действие) и *flags* (флаги) определяют тип операции уведомления и условия; параметр *event* (событие) является структурой типа **struct sigevent**, которая определяет событие уведомления (если оно есть), которое клиент хочет получить. Событие *event* клиенту доставляется функцией *MsgDeliverEvent()* или функцией *iofunc\_notify\_trigger()*.

Возвращает: Код завершения, при помощи вспомогательного макроса *\_RESMGR\_STATUS*; флаги возвращается ответным сообщением.

### *io\_open()*

```

int io_open(resmgr_context_t *ctp, io_open_t *msg,
    RESMGR_HANDLE_T *handle, void *extra)

```

Классификация: Функция установления соединения

Обработчик по умолчанию: *iofunc\_open\_default()*

Вспомогательные функции: *iofunc\_open()*, *iofunc\_ocb\_attach()*

Клиентская функция: *open()*, *fopen()*, *sopen()* и др.

Сообщения: *\_IO\_CONNECT*, подтипы *\_IO\_CONNECT\_COMBINE*, *\_IO\_CONNECT\_COMBINE\_CLOSE* и *\_IO\_CONNECT\_OPEN*.

Структура данных:

```

struct _io_connect {
    // Внутренние поля (как описано выше)
    uint16_t path_len;
    uint8_t  extra_type;
    uint16_t extra_len;
    char     path[1];
};

```

```

struct _io_connect_link_reply {
    uint32_t reserved1[2];
    uint8_t  eflag;
    uint8_t  reserved2[3];
    uint32_t umask;
    uint16_t nentries;
    uint16_t path_len;
};

```

```
typedef union {
    struct _io_connect          connect;
    struct _io_connect_link_reply link_reply;
} io_open_t;
```

Описание: Это основная точка входа в администратор ресурсов. Она выполняет проверку, действительно ли клиент имеет соответствующие права на открытие файла, привязывает ОСВ к внутренним структурам библиотеки (посредством функций *resmgr\_bind\_ocb()* или *iofunc\_ocb\_attach()*) и возвращает *errno*. Отметим, что для данной функции релевантны не все поля структур ввода и вывода.

Возвращает: Код завершения, при помощи вспомогательного макроса *\_IO\_SET\_CONNECT\_RET*.

### *io\_openfd()*

```
int io_openfd(resmgr_context_t *ctp, io_openfd_t *msg,
    RESMGR_OCB_T *ocb)
```

Классификация: Функция ввода/вывода

Обработчик по умолчанию: *iofunc\_openfd\_default()*

Вспомогательные функции: *iofunc\_openfd()*

Клиентская функция: *openfd()*

Сообщения: *\_IO\_OPENFD*

Структура данных:

```
struct _io_openfd {
    uint16_t          type;
    uint16_t          combine_len;
    uint32_t          ioflag;
    uint16_t          sflag;
    uint16_t          reserved1;
    struct _msg_info info;
    uint32_t          reserved2;
    uint32_t          key;
};
```

```
typedef union {
    struct _io_openfd i;
} io_openfd_t;
```



Описание: Данная функция аналогична предоставляемому обработчику *io\_open()* — за исключением того, что вместо имени пути передается дескриптор уже открытого файла (в силу передачи вам параметра *ocb* в вызове функции).

Возвращает: Код завершения, при помощи вспомогательного макроса *\_RESMGR\_STATUS*.

### *io\_pathconf()*

```
int io_pathconf(resmgr_context_t *ctp, io_pathconf_t *msg,  
RESMGR_OCB_T *ocb)
```

Классификация: Функция ввода/вывода

Обработчик по умолчанию: *iofunc\_pathconf\_default()*

Вспомогательные функции: *iofunc\_pathconf()*

Клиентская функция: *fpathconf()*, *pathconf()*

Сообщения: *IO\_PATHCONF*

Структура данных:

```
struct _io_pathconf {  
    uint16_t type;  
    uint16_t combine_len;  
    short    name;  
    uint16_t zero;  
};
```

```
typedef union {  
    struct _io_pathconf i;  
} io_pathconf_t;
```

Описание: Обработчик этого сообщения отвечает за возврат значения настраиваемого параметра *name* для ресурса, связанного с данным ОСВ. Используйте функцию по умолчанию и расширьте ее дополнительными вариантами элемента *name*, соответствующими вашему устройству.

Возвращает: Код завершения, при помощи вспомогательного макроса *\_IO\_SET\_PATHCONF\_VALUE*; флаги возвращаются в ответном сообщении.

### *io\_read()*

```
int io_read(resmgr_context_t *ctp, io_read_t *msg,  
RESMGR_OCB_T *ocb)
```

Классификация: Функция ввода/вывода

Обработчик по умолчанию: *iofunc\_read\_default()*

Вспомогательные функции: *iofunc\_read()*, *iofunc\_read\_verify()*

Клиентская функция: *read()*, *readdir()*

Сообщение: `IO_READ`

Структура данных:

```
struct _io_read {  
    uint16_t type;  
    uint16_t combine_len;  
    int32_t  nbytes;  
    uint32_t xtype;  
};
```

```
typedef union {  
    struct _io_read i;  
} io_read_t;
```

Описание: Отвечает за чтение данных из ресурса. Клиент задает число байт, которое он готов прочитать, в элементе *nbytes*. Вы возвращаете данные, увеличиваете смещение в ОСВ и обновляете соответствующие поля с информацией о временах доступа.

Отметим, что элемент *xtype* может устанавливать для отдельных сообщений флаг переопределения, поэтому его надо проверять. Если вы не поддерживаете никаких расширенных флагов переопределения, вы должны вернуть `EINVAL`. Далее, в примерах функций *io\_read()* и *io\_write()*, мы рассмотрим обработку одного очень важного (и очень непростого!) флага переопределения, называемого `_IO_XTYPE_OFFSET`.

Отметим также, что сообщение `_IO_READ` приходит не только для обычных файлов, но также и для чтения содержимого каталогов. В варианте с каталогом вы должны гарантированно обеспечить возврат целого (integral) числа элементов `struct dirent`. За дополнительной информацией по возврату элементов каталога см. пример в параграфе «Возврат элементов каталога» раздела «Дополнительно».

Чтобы удостовериться, что файл был открыт в режиме, совместимом с операцией чтения, надо вызвать вспомогательную функцию *iofunc\_read\_verify()*. Также, следует вызвать функцию *iofunc\_sync\_verify()*, чтобы проверить, надо ли синхронизировать данные с носителем.

Возвращает: Число считанных байтов или код завершения, при помощи вспомогательного макроса `_IO_SET_READ_NBYTES`, а также собственно данные — ответным сообщением.

В качестве примера с возвратом только данных см. ниже раздел «Простой пример функции *io\_read()*». Более сложный пример с одновременным возвратом как данных, так и элементов каталогов, см. в параграфе «Возврат элементов каталога» раздела «Дополнительно».

### *io\_readlink()*

```
int io_readlink(resmgr_context_t *ctp, io_readlink_t *msg,  
    RESMGR_HANDLE_T *handle, void*reserved)
```

Классификация: Функция установления соединения

Обработчик по умолчанию: Нет

Вспомогательные функции: *iofunc\_readlink()*

Клиентская функция: *readlink()*

Сообщения: IO\_CONNECT, подтип IO\_CONNECT\_READLINK

Структура данных:

```
struct _io_connect {  
    // Внутренние поля (как описано выше)  
    uint16_t path_len;  
    uint8_t  extra_type;  
    uint16_t extra_len;  
    char     path[1];  
};  
  
struct _io_connect_link_reply {  
    uint32_t reserved1[2];  
    uint8_t  eflag;  
    uint8_t  reserved2[3];  
    uint32_t umask;  
    uint16_t nentries;  
    uint16_t path_len;  
};  
  
typedef union {  
    struct _io_connect          connect;  
    struct _io_connect_link_reply link_reply;  
} io_open_t;
```

Описание: Отвечает за чтение содержимого символьной связи (линка), как определено полем *path* входной структуры. Возвращаемые байты представляют собой содержимое символьной связи; возвращаемый код

состояния представляет собой число байт в ответе. Допустимый возврат должен быть сделано только для символьной связи. Все другие доступы должны вернуть код ошибки.

Возвращает: Код завершения, при помощи вспомогательного макроса `_RESMGR_STATUS`, и данные в ответном сообщении.

### *io\_rename()*

```
int io_rename(resmgr_context_t *ctp, io_rename_t *msg,  
    RESMGR_HANDLE_T *handle, io_rename_extra_t*extra)
```

Классификация: Функция установления соединения

Обработчик по умолчанию: Нет

Вспомогательные функции: *iofunc\_rename()*

Клиентская функция: *rename()*

Сообщение: `_IO_CONNECT`, подтип `_IO_CONNECT_RENAME`

Структура данных:

```
struct _io_connect {  
    // internal fields (as described above)  
    uint16_t path_len;  
    uint8_t  extra_type;  
    uint16_t extra_len;  
    char     path[1];  
};
```

```
struct _io_connect_link_reply {  
    uint32_t reserved1[2];  
    uint8_t  eflag;  
    uint8_t  reserved2[3];  
    uint32_t umask;  
    uint16_t nentries;  
    uint16_t path_len;  
};
```

```
typedef union _io_rename_extra {  
    char path[1];  
} io_rename_extra_t;
```

```
typedef union {  
    struct _io_connect          connect;
```

```
struct _io_connect_link_reply link_reply;
} io_rename_t;
```

Описание: Выполняет операцию переименования, получив на вход первоначальное имя в элементе *path* и новое имя в поле *path* переданного параметра *extra*. Замечание по реализации: для первоначального имени задается имя пути (а не ОСВ) — это делается специально для случая переименования файла, который является жесткой связью к другому файлу. Если бы был задан ОСВ, две (или более) жестких связей к одному и тому же файлу различить было бы нельзя.

Данная функция будет вызываться только для тех двух имен файлов, которые принадлежат одной и той же файловой системе (то есть одному и тому же устройству). Поэтому в проверке случаев, в которых надо было бы возвращать EXDEV, нет никакой необходимости. Это ничуть не мешает вам возвращать EXDEV — например, если вы не хотите выполнять *rename()* самостоятельно (например, операция переименования из одного каталога в другой может оказаться очень сложной). В случае возврата EXDEV командно-строковая утилита *mv* выполнит сначала *cp*, а потом *rm* (библиотечная функция *rename()* этого не сделает — она просто установит *errno* в EXDEV).

Также перед вызовом этой функции должны быть разрешены все символьные связи (где это применимо), а переданные имена путей должны быть абсолютны и относиться к файловой системе, за которую отвечает данный администратор ресурсов.

Возвращает: Код завершения, при помощи вспомогательного макроса *\_RESMGR\_STATUS*.

### *io\_shutdown()*

```
int io_shutdown(resmgr_context_t *ctp, io_shutdown_t *msg,
RESMGR_OCB_T *ocb)
```

Данная функция зарезервирована QSSL для будущего использования. Вам следует инициализировать таблицу функций ввода/вывода, используя *iofunc\_func\_init()*, и не изменять данную точку входа.

### *io\_space()*

```
int io_space(resmgr_context_t *ctp, io_space_t *msg,
RESMGR_OCB_T *ocb)
```

Классификация: Функция ввода/вывода

Обработчик по умолчанию: Нет

Вспомогательные функции: *iofunc\_space\_verify()*

Клиентская функция: *chsize()*, *fcntl()*, *ftruncate()*, *ltrunc()*

Сообщение **\_IO\_SPACE**

Структура данных:

```
struct _io_space {
    uint16_t type;
    uint16_t combine_len;
    uint16_t subtype;
    short whence;
    uint64_t start;
    uint64_t len;
};
```

```
typedef union {
    struct _io_space i;
    uint64_t o;
} io_space_t;
```

Описание: Эта функция применяется для выделения или освобождения занимаемого ресурсом пространства. Параметр *subtype* («подтип») указывает на то, следует ли это пространство выделить (если равен **F\_ALLOCS**) или освободить (если равен **F\_FREES**). Комбинация параметров *whence* («откуда») и *start* («начало») указывает, где следует начать выделение/освобождение; элемент *len* указывает размер операции.

Возвращает: Число байтов (размер ресурса), посредством вспомогательного макроса **\_RESMGR\_STATUS**.

### ***io\_stat()***

```
int io_stat(resmgr_context_t *ctp, io_stat_t *msg,
    RESMGR_OCB_T *ocb)
```

Классификация: Функция ввода/вывода

Обработчик по умолчанию: *iofunc\_stat\_default()*

Вспомогательные функции: *iofunc\_stat()*

Клиентская функция: *stat()*, *lstat()*, *fstat()*

Сообщения: **\_IO\_STAT**

Структура данных:

```
struct _io_stat {
```

```

uint16_t type;
uint16_t combine_len;
uint32_t zero;
};

```

```

typedef union (
    struct _io_stat i;
    struct stat      o;
} io_stat_t;

```

Описание: Обрабатывает сообщение, запрашивающее информацию о ресурсе, связанном с переданным ОСВ. Заметьте, что атрибутная запись содержит всю информацию, необходимую для выполнения запроса *stat()*. Вспомогательная функция *iofunc\_stat()* заполняет структуру **struct stat**, базированную на атрибутной записи. Эта вспомогательная функция также изменяет сохраненные элементы *dev/rdev* так, чтобы они были уникальны с точки зрения единичного узла (это используется для выполнения вызовов *stat()* в отношении файлов по сети). Писать этот обработчик самостоятельно особого смысла нет.

Возвращает: Код завершения, при помощи вспомогательного макроса *\_RESMGR\_STATUS*, и структуру **struct stat** — в ответном сообщении.

### *io\_sync()*

```

int io_sync(resmgr_context_t *ctp, io_sync_t *msg,
    RESMGR_OCB_T *ocb)

```

Классификация: Функция ввода/вывода

Обработчик по умолчанию: *iofunc\_sync\_default()*

Вспомогательные функции: *iofunc\_sync\_verify()*, *iofunc\_sync()*

Клиентская функция: *fsync()*, *fdatasync()*

Сообщения: *\_IO\_SYNC*

Структура данных:

```

struct _io_sync {
    uint16_t type;
    uint16_t combine_len;
    uint32_t flag;
};

```

```

typedef union {
    struct _io_sync i;

```

```
} io_sync_t;
```

Описание: Это точка входа команды `flush` (синхронизация носителя информации с буферами — например, диска с дисковым кэшем — *прим. ред.*). Вспомогательная функция `iofunc_sync()` принимает поле *flag* входного сообщения и возвращает одно из следующих значений, которые указывают, какие действия ваш администратор ресурсов должен выполнить:

- 0 — не делать ничего;
- O\_SYNC — все, что связано с файлом (включая содержимое файла, элементы каталогов, индексные дескрипторы (inodes), и т.д.) должно присутствовать на носителе и должно быть восстанавливаемым с него.
- O\_DSYNC — присутствовать на носителе и быть восстанавливаемыми с него должны только данные файла.

Отметим, что эта функция будет вызываться только в том случае, если вы согласились поддерживать функцию `sync`, установив соответствующий флаг в записи точки монтирования.

Возвращает: Код завершения, при помощи вспомогательного макроса `_RESMGR_STATUS`.

### ***io\_umount()***

```
int io_umount(resmgr_context_t *ctp, void *msg,  
              RESMGR_OCB_T *ocb)
```

Данная функция зарезервирована QSSL для будущего использования. Вам следует инициализировать таблицу функций ввода/вывода, используя `iofunc_func_init()`, и не изменять данную точку входа.

### ***io\_unblock()** [установка соединения]*

```
int io_unblock(resmgr_context_t *ctp, io_pulse_t *msg,  
               RESMGR_HANDLE_T *handle, void*reserved)
```

Классификация: Функция установления соединения (синтезируется ядром и библиотекой)

Обработчик по умолчанию: Нет

Вспомогательные функции: `iofunc_unblock()`

Клиентская функция: Нет (вызывается ядром вследствие сигнала или тайм-аута)

Сообщения: Нет (синтезируется библиотекой)

Структура данных: (см. вариант `io_unblock()` для ввода/вывода, ниже)



Описание: Это версия разблокирующего вызова в форме сообщения установления соединения, синтезируемая библиотекой в ответ на импульс от ядра, возникший в результате желания клиента разблокироваться на этапе установления соединения. См. вариант этой функции для ввода/вывода ниже.

Возвращает: Код завершения, при помощи вспомогательного макроса `_RESMGR_STATUS`.

См. подробное обсуждение стратегий разблокирования в главе «Обмен сообщениями», параграф «Применение флага `_NTO_MI_UNBLOCK_REQ`».

### *`io_unblock()` [ввод/вывод]*

```
int io_unblock(resmgr_context_t *ctp, io_pulse_t *msg,  
               RESMGR_OCB_T *ocb)
```

Классификация: Функция ввода/вывода (синтезируется ядром и библиотекой)

Обработчик по умолчанию: `iofunc_unblock_default()`

Вспомогательные функции: `iofunc_unblock()`

Клиентская функция: Нет (реакция ядра на сигнал или тайм-аут)

Сообщения: Нет (синтезируется библиотекой)

Структура данных: указатель на структуру, содержащую прерываемое сообщение

Описание: Это версия разблокирующего вызова для сообщения ввода/вывода, синтезируемая библиотекой в результате импульса от ядра, возникшего вследствие попытки клиента разблокироваться на этапе ввода/вывода. Обработчик `io_unblock()` для фазы установления соединения почти аналогичен (см. предыдущий параграф).

Общим для обеих обработчиков разблокировки (как для функций установления соединения, так и для функций ввода/вывода) является желание клиента разблокироваться с разрешения администратора ресурсов. Администратор ресурсов *обязан* ответить на клиентское сообщение, чтобы разблокировать клиента. (Мы это обсуждали в главе «Обмен сообщениями», когда говорили о флагах функции `ChannelCreate()` — в частности, о флаге `_NTO_CHF_UNBLOCK`).

Возвращает: Код завершения, при помощи вспомогательного макроса `_RESMGR_STATUS`.

Подробное обсуждение стратегий разблокирования см. в разделе «Применение флага `_NTO_MI_UNBLOCK_REQ`» в главе «Обмен

сообщениями».

### *io\_unlink()*

```
int io_unlink(resmgr_context_t *ctp, io_unlink_t *msg,  
    RESMGR_HANDLE_T*handle, void*reserved)
```

Классификация: Функция установления соединения

Обработчик по умолчанию: Нет

Вспомогательные функции: *iofunc\_unlink()*

Клиентская функция: *unlink()*

Сообщение: `_IO_CONNECT`, подтип `_IO_CONNECT_UNLINK`

Структура данных:

```
struct _io_connect {  
    // Внутренние поля (как описано выше)  
    uint16_t path_len;  
    uint8_t  extra_type;  
    uint16_t extra_len;  
    char     path[1];  
};
```

```
struct _io_connect_link_reply {  
    uint32_t reserved1[2];  
    uint8_t  eflag;  
    uint8_t  reserved2[3];  
    uint32_t umask;  
    uint16_t nentries;  
    uint16_t path_len;  
};
```

```
typedef union {  
    struct _io_connect          connect;  
    struct _io_connect_link_reply link_reply;  
} io_unlink_t;
```

Описание: Отвечает за уничтожение связей (unlinking) файла, имя пути которого передается в поле *path* структуры входящего сообщения.

Возвращает: Состояние по применению вспомогательного макроса `_RESMGR_STATUS`.

## *io\_unlock\_ocb()*

```
int io_unlock_ocb(resmgr_context_t *ctp, void *reserved,  
    RESMGR_OCB_T *ocb)
```

Классификация: Функция ввода/вывода (синтезируется библиотекой)

Обработчик по умолчанию: *iofunc\_unlock\_ocb\_default()*

Вспомогательные функции: Нет

Клиентская функция: Все

Сообщения: Нет (синтезируется библиотекой)

Структура данных: Нет

Описание: Действует противоположно вышеописанной функции *io\_lock\_ocb()*, т.е. отвечает за разблокирование атрибутной записи, на которую указывает ОСВ. Эта операция освобождает атрибутную запись, чтобы другие администраторы ресурсов могли с ней работать. Подробности см. выше в разделе «Составные сообщения».

Возвращает: Код завершения, при помощи вспомогательного макроса *\_RESMGR\_STATUS*.

## *io\_untime()*

```
int io_untime(resmgr_context_t *ctp, io_untime_t *msg,  
    RESMGR_OCB_T *ocb)
```

Классификация: Функция ввода/вывода

Обработчик по умолчанию: *iofunc\_untime\_default()*

Вспомогательные функции: *iofunc\_untime()*

Клиентская функция: *untime()*

Сообщения: *\_IO\_ETIME*

Структура данных:

```
struct _io_untime {  
    uint16_t      type;  
    uint16_t      combine_len;  
    int32_t       cur_flag;  
    struct utimbuf times;  
};
```

```
typedef union {  
    struct _io_untime i;  
} io_untime_t;
```

Описание: Устанавливает времена последнего доступа и модификации либо в «текущий момент» (если они равны нулю), либо в заданные значения. Заметьте, что согласно правилам POSIX этот обработчик сообщения может быть необходим для модификации флагов `IOFUNC_ATTR_*` в атрибутной записи. Вам почти никогда не придется самостоятельно использовать этот обработчик; вместо этого вы будете использовать вспомогательную функцию POSIX-уровня.

Возвращает: Код завершения, при помощи вспомогательного макроса `_RESMGR_STATUS`.

### *io\_write()*

```
int io_write(resmgr_context_t *ctp, io_write_t *msg,  
            RESMGR_OCB_T *ocb)
```

Классификация: Функция ввода/вывода

Обработчик по умолчанию: *iofunc\_write\_default()*

Вспомогательные функции: *iofunc\_write\_verify()*

Клиентская функция: *write()*, *fwrite()*, и т.п.

Сообщения: `_IO_WRITE`

Структура данных:

```
struct _io_write {  
    uint16_t type;  
    uint16_t combine_len;  
    int32_t nbytes;  
    uint32_t xtype;  
};
```

```
typedef union {  
    struct _io_write i;  
} io_write_t;
```

Описание: Данный обработчик отвечает за получение данных, которые клиент записал в администратор ресурсов. Обработчику передается число байт, которые клиент пытается записать, в элементе *nbytes*; данные неявно следуют за входной структурой (если параметр *xtype* не установлен в `_IO_XTYPE_OFFSET`; см. ниже параграф «Простой пример функции *io\_write()*»). Согласно реализации, потребуется повторное считывание от клиента части сообщения с данными при помощи функции *resmgr\_msgreadv()* или ей эквивалентной. Код завершения дает число байт, фактически записанных, либо устанавливает признак ошибки в *errno*.

Отметьте, что чтобы удостовериться, что файл был открыт в режиме, совместимом с записью, следует вызвать вспомогательную функцию *iofunc\_write\_verify()*. Также следует вызывать функцию *iofunc\_sync\_verify()* для проверки необходимости синхронизации данных с носителем.

Возвращает: Код завершения, при помощи вспомогательного макроса *\_IO\_SET\_WRITE\_NBYTES*.

Пример см. ниже в параграфе «Простой пример функции *io\_write()*».

## Примеры

Этот раздел — своего рода «кулинарная книга» для программистов. Здесь я приведу ряд готовых примеров, которые вы сможете непосредственно использовать в качестве базиса для ваших проектов. Это не совсем готовые администраторы ресурсов — вы должны будете дополнить их функциями работы с пулами потоков и «каркасом» диспетчеризации (о котором речь ниже), а также удостовериться, что ваши версии функций-обработчиков помещаются в соответствующие таблицы функций *после* вызова *iofunc\_func\_init()*, чтобы корректно переопределить значения по умолчанию!

Я начну с ряда простых примеров, демонстрирующих базовые функциональные возможности обработчиков различных сообщений, таких как:

- *io\_read()*
- *io\_write()*
- *io\_devctl()* (без передачи данных)
- *io\_devctl()* (с передачей данных)

Затем, в разделе «Дополнительно», мы рассмотрим обработчик *io\_read()*, который обеспечивает возврат элементов каталога.

### ***Базовый каркас администратора ресурсов***

Приведенный ниже пример можно использовать в качестве шаблона для многопоточного администратора ресурсов. (Шаблон однопоточного администратора ресурсов мы уже рассматривали — это было в разделе «Библиотека администратора ресурсов», когда мы обсуждали администратор */dev/null*).

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>

static resmgr_connect_funcs_t connect_func;
static resmgr_io_funcs_t io_func;
static iofunc_attr_t attr;
```

```

main(int argc, char **argv) {
    thread_pool_attr_t pool_attr;
    thread_pool_t      *tpp;
    dispatch_t         *dpp;
    resmgr_attr_t       resmgr_attr;
    resmgr_context_t    *ctp;
    int                 id;
    if ((dpp = dispatch_create()) == NULL) {
        fprintf(stderr,
            "%s: Ошибка выделения контекста диспетчеризации\n",
            argv[0]);
        return (EXIT_FAILURE);
    }
    memset(&pool_attr, 0, sizeof(pool_attr));
    pool_attr.handle = dpp;
    pool_attr.context_alloc = resmgr_context_alloc;
    pool_attr.block_func = resmgr_block;
    pool_attr.handler_func = resmgr_handler;
    pool_attr.context_free = resmgr_context_free;

    // 1) Настроить пул потоков
    pool_attr.lo_water = 2;
    pool_attr.hi_water = 4;
    pool_attr.increment = 1;
    pool_attr.maximum = 50;
    if ((tpp =
        thread_pool_create(&pool_attr, POOL_FLAG_EXIT_SELF))
        == NULL) {
        fprintf(stderr,
            "%s: Ошибка инициализации пула потоков\n",
            argv[0]);
        return (EXIT_FAILURE);
    }
    iofunc_func_init(_RESMGR_CONNECT_NFUNCS, &connect_func,
        _RESMGR_IO_NFUNCS, &io_func);
    iofunc_attr_init(&attr, S_IFNAM | 0777, 0, 0);

    // 2) Переопределить функции установления соединения
    // и функции ввода/вывода, как надо

```

```

memset(&resmgr_attr, 0, sizeof(resmgr_attr));
resmgr_attr.nparts_max = 1;
resmgr_attr.msg_max_size = 2048;

// 3) Замените «/dev/whatever» на нужный префикс
if ((id =
    resmgr_attach(dpp, &resmgr_attr, "/dev/whatever",
        _FTYPE_ANY,
        0, &connect_func, &io_func, &attr)) == -1) {
    fprintf(stderr, "%s: Ошибка регистрации префикса\n",
        argv[0]);
    return (EXIT_FAILURE);
}
// Отсюда возврата не будет
thread_pool_start(tpp);
}

```

Дополнительную информацию об интерфейсе диспетчеризации (т.е., о функции *dispatch\_create()*), см. в справочном руководстве по Си-библиотеке (C Library Reference).

### Этап 1

Здесь мы используем функции пула потоков для создания пула, который должен будет обслуживать сообщения в нашем администраторе ресурсов. Вообще говоря, я бы рекомендовал вам начать однопоточного администратора ресурсов, как мы это делали ранее в примере с администратором */dev/null*. Как только базовая функциональность у вас заработает, вы сможете *затем* добавить многопоточность. Вам нужно будет задать параметры *lo\_water*, *hi\_water*, *increment* и *maximum* структуры *pool\_attr*, как это было описано в главе «Процессы и потоки» в обсуждениях функций пула потоков.

### Этап 2

Здесь мы дополняем администратор ресурсов *нашими* функциями. Эти функции представляют собой функции-обработчики сообщений, о которых мы только что говорили (например, *io\_read()*, *io\_devctl()*, и т.п.). Например, чтобы добавить свой собственный обработчик для сообщения



`_IO_READ`, указывающий на функцию `my_io_read()`, мы должны были бы добавить в программу такую строку:

```
io_func.io_read = my_io_read;
```

Это переопределит элемент таблицы, инициализированный вызовом `iofunc_func_init()` и содержащий функцию POSIX-уровня по умолчанию, заменив его указателем на вашу функцию `my_io_read()`.

### Этап 3

Вы, вероятно, не захотите, чтобы ваш администратор ресурсов назывался `/dev/whatever` (букв. — «`/dev/абы_что`» — прим. ред.), так что вам придется выбрать для него соответствующее имя. Отметим, что привязка атрибутной записи (параметр `attr`) к регистрируемому префиксу осуществляется вызовом `resmgr_attach()` — если бы нам было надо, чтобы наш администратор обрабатывал несколько устройств, нам пришлось бы вызывать `resmgr_attach()` несколько раз, каждый раз с новой атрибутной записью, чтобы на этапе выполнения можно было отличить зарегистрированные префиксы друг от друга.

### Простой пример функции `io_read()`

Чтобы проиллюстрировать, как ваш администратор ресурса мог бы возвращать клиенту данные, рассмотрим простейший администратор ресурса, который всегда возвращает одну и ту же строковую константу «Здравствуй, мир!\n». Даже в таком простом случае необходимо учесть ряд моментов, как-то:

- согласование размера клиентской области данных с количеством данных, подлежащих возврату;
- обработка EOF;
- поддерживание контекстной информации (индекс `lseek()`);
- обновление POSIX-информации `stat()`.

### Учет размеров областей данных

В нашем случае администратор ресурсов возвращает фиксированную строку длиной в 17 байт, то есть размер доступных данных точно известен и постоянен. Эти аналогично случаю с дисковым файлом, доступным

только для чтения и содержащим рассматриваемую строку. Единственное реальное отличие состоит в том, что этот «файл» обеспечивается в нашей программе строкой:

```
char *data_string = "Здравствуй, мир!\n";
```

С другой стороны, клиент может запросить чтение любого объема данных — один байт, 17 байт или более. Это должно отразиться на характеристиках вашей реализации *io\_read()* ее умением согласовывать размер запрашиваемых клиентом данных с размером данных, имеющихся в наличии.

### ***Обработка EOF***

Особым случаем согласования размеров областей данных является обработка EOF для строки фиксированной длины. Как только клиент считал заключительный символ «\n», дальнейшие его попытки считать данные должны возвращать EOF.

### ***Поддерживание контекстной информации***

И «учет размеров областей данных», и «обработка EOF» требуют, чтобы в ОСВ, передаваемом вашей функции *io\_read()*, поддерживалась контекстная информация — в частности, поле *offset*.

### ***Обновление информации POSIX***

И еще одно заключительное соображение: при чтении данных из ресурса должна обновляться POSIX-переменная времени доступа *atime* («access time» — «время доступа»). Это делается для того, чтобы клиентская функция *stat()* могла обнаружить, что к устройству кто-то обращался.

### ***Собственно код***

Ниже приведена программа, в которой учтены все вышеперечисленные моменты. Ниже мы ее последовательно проанализируем.

/\*

```

* io.read1.c
*/

#include <stdio.h>
#include <errno.h>
#include <sys/neutrino.h>
#include <sys/iofunc.h>

// наша строка с данными
char* data_string = "Здравствуй, мир!\n";

int io_read(resmgr_context_t* ctp, io_read_t* msg,
iofunc_ocb_t* ocb) {
    int sts;
    int nbytes;
    int nleft;
    int off;
    int xtype;
    struct _xtype_offset* xoffset;

    // 1) Проверить, открыто ли устройство на чтение
    if ((sts ==
        iofunc_read_verify(ctp, msg, ocb, NULL)) != EOK) {
        return sts;
    }

    // 2) проверить и обработать переопределение XTYPE
    xtype = msg->i.xtype & _IO_XTYPE_MASK;
    if (xtype == _IO_XTYPE_OFFSET) {
        xoffset = (struct _xtype_offset*)(msg->i + 1);
        off = xoffset->offset;
    } else if (xtype == _IO_XTYPE_NONE) {
        off = ocb->offset;
    } else { // Неизвестный тип; игнорировать
        return ENOSYS;
    }

    // 3) Сколько байт осталось?
    nleft = ocb->attr->nbytes - off;

```

```

// 4) Сколько байт мы можем отдать клиенту?
nbytes = min(nleft, msg->i.nbytes);

// 5) Если возвращаем данные, отдать их клиенту
if (nbytes) {
    MsgReply(ctp->rcvid, nbytes, data_string+off, nbytes);
    // 6) Установить значение "atime" для POSIX stat()
    ocb->attr->flags |=
        IOFUNC_ATTR_ATIME | IOFUNC_ATTR_DIRTY_TIME;
    // 7) Если индекс lseek() не равен _IO_XTYPE_OFFSET,
    // увеличить его на число считанных байт
    if (xtype == _IO_XTYPE_NONE) {
        ocb->offset += nbytes;
    }
} else {
    // 8) Не возвращаем данные, просто разблокировать клиента
    MsgReply(ctp->rcvid, EOK, null, 0);
}

// 9) Сказать библиотеке, что мы уже ответили сами
return _RESMGR_NOREPLY;
}

```

### Этап 1

Здесь мы убедились, что клиентский вызов *open()* действительно запросил открытие устройства на чтение. Если бы клиент открыл устройство только на запись, а затем попытался выполнить чтение, это следовало бы расценивать как ошибку. В этом случае вспомогательная функция *iofunc\_read\_verify()* возвратила бы нам (затем мы — библиотеке, а библиотека — клиенту) EBADF, а не EOK.

### Этап 2

Здесь мы проверили, указал ли клиент индивидуальное для данного сообщения переопределение типа (*xtype-override*) (например, потому что если мы открыли устройство в неблокирующем режиме, то это указало бы, что для данного конкретного запроса мы хотим задать блокирующее

поведение). Отметим, что блокирующий аспект переопределения типа может быть отражён в последнем параметре функции *iofunc\_read\_verify()*, однако, поскольку мы приводим здесь упрощенный пример, мы передаем NULL, указывая этим, что этот вопрос нас не волнует.

Более важно, однако, посмотреть, как обрабатываются конкретные модификаторы *xtype*. Очень интересен, например, модификатор `_IO_XTYPE_OFFSET`, который, если присутствует, указывает на то, что принятое от клиента сообщение содержит смещение, и что операция чтения не должна изменять «текущую позицию файла» для данного файлового дескриптора (так делает, например, функция *pread()*). Если модификатор `_IO_XTYPE_OFFSET` не указан, то операция чтения может смело модифицировать «текущую позицию файла». Мы используем переменную *xtype* для сохранения *xtype*, содержавшегося в принятом сообщении, и переменную *off* для представления текущего смещения, которое мы должны будем использовать при обработке. Далее, на этапе 7, вы увидите еще кое-какие действия по обработке модификатора `_IO_XTYPE_OFFSET`.

Если присутствует иное переопределение *xtype*, чем `_IO_XTYPE_OFFSET` (и это не пустая команда `_IO_XTYPE_NONE`), мы отказываемся обрабатывать запрос и возвращаем ENOSYS. Это просто означает, что мы не знаем, как обрабатывать такую ситуацию, и поэтому возвращаем клиенту признак ошибки.

### Этапы 3 и 4

Чтобы вычислить, сколько байт мы можем реально вернуть клиенту, мы выполняем этапы 3 и 4, в которых выясняется, сколько байт доступно у устройства (разность между полным объемом устройства, полученным из `ocb->attr->nbytes`, и текущим смещением в устройстве). Узнав, сколько байт осталось, мы выбираем наименьшее значение между размером этого остатка и количеством байт, которые клиент хочет прочитать. Например, у нас может остаться семь байт, а клиент захочет прочитать только два. В этом случае мы возвратим клиенту только два байта. И наоборот, если клиент захочет прочитать 4096 байт, а у нас осталось только семь, мы сможем вернуть ему только семь байт.

### Этап 5

Теперь, вычислив, сколько байт мы намерены вернуть клиенту, нам нужно сделать ряд вещей в зависимости от того, возвращаем мы данные или нет. Если да, то мы просто отвечаем клиенту с данными сразу после проверки на этапе 5. Обратите внимание, что для возврата данных с корректного смещения мы используем `data_string + off` (смещение *off* вычисляется в зависимости от наличия переопределения типа). Отметьте также второй параметр функции *MsgReply()* — в документации он упоминается как «*status*» («код завершения»), но в этом случае мы используем его для возврата числа байт. Мы делаем так, потому что реализация клиентской функции *read()* знает, что значение, возвращаемое ее функцией *MsgSendv()* (а это, кстати, как раз и есть параметр *status* функции *MsgReply()*) представляет собой число реально прочитанных байт — это общеизвестное соглашение.

## Этап 6

Поскольку мы возвращаем данные от устройства, мы знаем, что к устройству производился доступ. Мы устанавливаем биты `IOFUNC_ATTR_ETIME` и `IOFUNC_ATTR_DIRTY_TIME` в поле *flags* атрибутной записи. Это служит напоминанием для функции *io\_stat()* о том, что время доступа стало недействительным, и перед выполнением ответа его следует скорректировать по системным часам. Если бы нам очень хотелось, мы могли бы записать текущее время в поле *etime* атрибутной записи и сбросить флаг `IOFUNC_ATTR_DIRTY_TIME`; однако, это было бы не очень-то эффективно, поскольку мы предполагаем получить от клиента значительно большее количество запросов типа *read()*, чем запросов типа *stat()*. Впрочем, ваши условия могут диктовать иначе.

☞ Так какое же время видит клиент, когда он вызывает-таки функцию *stat()*? Функция *iofunc\_stat\_default()*, предоставляемая библиотекой администратора ресурсов, посмотрит на поле *flags* атрибутной записи, чтобы проверить, являются времена доступа (поля *etime*, *ctime* и *mtime*) корректными или нет. Если нет (как это было бы после вызова *io\_read()* с возвратом данных), *iofunc\_stat\_default()* устанавливает нужные из них в значение текущего времени.

## Этап 7

Теперь мы увеличиваем смещение *lseek()* на число возвращенных клиенту байт, но делаем это только в том случае, если не обрабатываем модификатор `_IO_XTYPE_OFFSET`. Это гарантирует, что в случае отсутствия флага `_IO_XTYPE_OFFSET`, если клиент вызовет функцию *lseek()* для определения текущей позиции, или (более важный случай) если клиент вызовет *read()* для чтения еще нескольких байт, смещение в ресурсе будет корректным. Если `_IO_XTYPE_OFFSET` установлен, мы оставляем содержащееся в *osb* смещение в покое.

### Этап 8

Сопоставьте этот этап с этапом 6. Здесь мы только разблокируем клиента и не выполняем больше никаких действий. Также обратите внимание, что функции *MsgReply()* не передается никакой области данных, потому что в этом случае данные мы не возвращаем.

### Этап 9

И наконец, на этапе 9 мы выполняем действия, не зависящие от того, возвращаем мы данные клиенту или нет. Поскольку мы уже сами разблокировали клиента при помощи *MsgReply()*, мы, конечно же, не хотим, чтобы это попыталась сделать еще и библиотека администратора ресурсов. Поэтому мы сообщаем ей, что мы уже сделали это сами, возвратом `_RESMGR_NOREPLY`.

### Эффективное применение других функций обмена сообщениями

Как вы помните из главы «Обмен сообщениями», мы упоминали еще несколько функций обмена сообщениями, а именно — функции *MsgWrite()*, *MsgWritev()* и *MsgReplyv()*. Повод, в связи с которым я снова упоминаю здесь эти функции, состоит в том, что ваша функция *io\_read()* может быть превосходным местом для их применения. В простом примере, показанном выше, мы возвращали непрерывный массив данных из постоянного места в памяти. В реальной же жизни вам может понадобиться вернуть, скажем, множество фрагментов данных из различных выделенных вами буферов. Классическим примером такого случая является циклический буфер, который часто применяется, например, в драйверах последовательных

устройств. Часть данных может быть размещена в конце буфера, другая часть — в начале. В этом случае для возврата обеих частей данных вам понадобилось бы передать *MsgReplyv()* двухэлементный вектор ввода/вывода (IOV), где первый элемент содержал бы адрес (и длину) «нижней» части данных, а второй — адрес (и длину) «верхней» части. Или же, если вы ожидаете прибытия данных частями, вы могли бы вместо этого использовать функции *MsgWrite()* или *MsgWritev()* для записи данных в адресное пространство клиента по мере их поступления, а затем выдать заключительный вызов *MsgReply()* или *MsgReplyv()*, чтобы разблокировать клиента. Как мы уже показали выше, функция *MsgReply()* может и не передавать никаких данных — вы можете использовать ее просто для того, чтобы разблокировать клиента.

### Простой пример функции *io\_write()*

Это был простой пример функции *io\_read()*; давайте теперь перейдем к функции *io\_write()*. Основной камень преткновения, связанный с *io\_write()*, — получить доступ к данным. Поскольку библиотека администратора ресурсов считывает лишь незначительную часть сообщения от клиента, переданные клиентом данные (они идут сразу после заголовка `_IO_WRITE`) могут быть приняты функцией *io\_write()* только частично. Простой пример — представьте себе клиента, записывающего один мегабайт. Библиотекой администратора ресурсов будут считаны только заголовок сообщения и несколько байт данных. Остальная часть мегабайта остается по-прежнему доступной на клиентской стороне — администратор ресурсов при желании может к ней обращаться.

Реально рассмотрения заслуживают только два случая:

- все содержимое сообщения клиентской функции *write()* было считано библиотекой администратора ресурсов полностью; или
- этого не произошло.

Судьбоносное решение, однако, состоит в ответе на следующий вопрос: «Какие проблемы сопряжены с попыткой сохранить полученную с первым сообщением часть данных?» Ответ такой: овчинка не стоит выделки. Тому есть ряд причин:

- обмен сообщениями (операции копирования на уровне ядра) выполняется очень быстро;
- проверка, получены ли данные целиком или частично, влечет определенные накладные расходы;



- дополнительные накладные расходы связаны с попыткой «сохранения» первой, уже прибывшей, части данных, в свете того факта, что ожидаются дополнительные.

По-моему, первые два пункта говорят сами за себя. Третий же пункт заслуживает пояснения. Давайте предположим, что клиент переслал большую порцию данных, и мы *приняли-таки* решение о том, что было бы неплохо попробовать сохранить ту часть данных, которая уже получена. К сожалению, эта часть оказалось очень небольшой. Это означает, что вместо одного непрерывного массива байт у нас теперь будет большой кусок и маленький «довесок». Иными словами, только ради этого маленького кусочка нам придется значительно усложнять работу с данными, что может неприятно сказаться на эффективности кода. Это потенциальная головная боль, не делайте так!

Реальным ответом на поставленный вопрос будет просто заново считать данные в заранее подготовленные буферы. В нашем простом примере функции *io\_write()* я буду просто каждый раз выделять буфер при помощи *malloc()*, считывать в него данные, а затем освобождать его функцией *free()*. Разумеется, есть и куда более эффективные способы выделения буферов и управления ими!

Еще один тонкий момент в этом примере функции *io\_write()* — это обработка модификатора `_IO_XTYPE_OFFSET` (и связанных с этим данных; здесь она выполняется несколько иначе, чем в примере *io\_read()*).

```
/*
 * io_writel.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/neutrino.h>
#include <sys/iofunc.h>

void process_data(int offset, void *buffer, int nbytes) {
    // Сделать что-нибудь с данными
}

int io_write(resmgr_context_t *ctp, io_write_t *msg,
            iofunc_ocb_t *ocb) {
    int sts;
    int nbytes;
    int off;
```

```

int start_data_offset;
int xtype;
char *buffer;
struct _xtype_offset *xoffset;

// Проверить, открыто ли устройство на запись
if ((sts =
    iofunc_write_verify(ctp, msg, ocb, NULL)) != EOK) {
    return (sts);
}

// 1) Проверить и обработать переопределение
XTYPE xtype = msg->i.xtype & _IO_XTYPE_MASK;
if (xtype == _IO_XTYPE_OFFSET) {
    xoffset = (struct _xtype_offset*)(&msg->i + 1);
    start_data_offset = sizeof(msg->i) + sizeof(*xoffset);
    off = xoffset->offset;
} else if (xtype == _IO_XTYPE_NONE) {
    off = ocb->offset;
    start_data_offset = sizeof(msg->i);
} else {
    // Известный тип; игнорировать
    return (ENOSYS);
}

// 2) Выделить достаточно большой буфер для данных
nbytes = msg->i.nbytes;
if ((buffer = malloc(nbytes)) == NULL) {
    return (ENOMEM);
}

// 3) Считать данные от клиента (возможно, повторно)
if (resmgr_msgread(ctp, buffer, nbytes,
    start_data_offset) == -1) {
    free(buffer);
    return (errno);
}

// 4) Сделать что-нибудь с данными
process_data(off, buffer, nbytes);

```

```

// 5) Освободить память буфера
free(buffer);

// 6) Установить, сколько байт должна возвращать
// клиентская функция «write»
_IO_SET_WRITE_NBYTES(ctp, nbytes);

// 7) Если данные записаны, обновить структуры
// данных POSIX и смещение OCB
if (nbytes) {
    ocb->attr->flags |=
        IOFUNC_ATTR_MTIME | IOFUNC_ATTR_DIRTY_TIME;
    if (xtype == _IO_XTYPE_NONE) {
        ocb->offset += nbytes;
    }
}

// 8) Пусть библиотека сама ответит, что все в порядке
return (EOK);
}

```

Как вы видите, некоторые начальные действия идентичны таковым из примера функции *io\_read()* — функция *iofunc\_write\_verify()* аналогична функции *iofunc\_read\_verify()*, и проверка переопределения *xtype* выполняется точно также.

### *Этап 1*

Здесь мы выполняем обработку переопределения *xtype*, в значительной степени аналогичную примеру с *io\_read()* — за исключением того, что смещение не сохраняется в поле структуры входящего сообщения. Причина этого состоит в том, что обычной практикой для определения начального адреса поступающих от клиента данных является использование размера структуры входящего сообщения. Мы предпринимаем дополнительные усилия, чтобы удостовериться, что смещение начала данных (*doffset*) в коде обработки *xtype* является корректным.

### *Этап 2*

Здесь мы выделяем буфер, достаточный для размещения в нем данных. Число байт, которые клиент собирается записать, представлено нам в поле *nbytes* объединения *msg*, оно заполняется автоматически Си-библиотекой клиента в коде функции *write()*. Отметим, что у нас недостаточно памяти для обработки запроса *malloc()*, мы возвращаем клиенту *ENOMEM*, чтобы он знал, почему его запрос потерпел неудачу.

### Этап 3

Здесь мы применяем вспомогательную функцию *resmgr\_msgread()* для считывания всего объема данных от клиента непосредственно в только что выделенный для этого буфер. В большинстве случаев здесь вполне сошла бы функция *MsgRead()*, но в случаях, когда сообщение является частью составного сообщения, функция *resmgr\_msgread()* выполни для нас еще и соответствующие «магические» действия (о том, почему это надо, см. раздел «Составные сообщения»). Параметры функции *resmgr\_msgread()* довольно очевидны: мы передаем ей указатель на внутренний контекстный блок (*ctp*), буфер, в который мы хотим поместить данные (*buffer*), и число байт, которые мы хотим считать (поле *nbytes* объединения *msg*). Последний параметр — это смещение, которое мы вычислили ранее, на этапе 1. Смещение реально позволяет пропустить заголовок, помещенный туда функцией *write()* клиентской Си-библиотеки, и сразу перейти к данным. Здесь возникает два интересных момента:

- мы могли бы взять произвольное смещение, чтобы считывать любые фрагменты данных в любом порядке, как нам заблагорассудится;
- мы могли бы использовать функцию *resmgr\_msgreadv()* (обратите внимание на символ «V» в названии) для считывания данных от клиента в вектор ввода/вывода, возможно, описывающий несколько разных буферов, подобно тому, как мы делали с буферами кэша при обсуждении файловых систем в главе «Обмен сообщениями».

### Этап 4

Здесь вы можете делать с данными все, что вашей душе угодно — я, например, вызвал некую условную функцию *process\_data()* и передал ей буфер и его размер.

## Этап 5

Критически важный этап! Упустить его из виду очень просто, но это неизбежно приведет к «утечкам памяти». Обратите также внимание, как мы позаботились об освобождении памяти в случае неудачи на этапе 3.

## Этап 6

Здесь мы используем макрос `_IO_SET_WRITE_NBYTES()` для сохранения числа записанных байт, которое затем будет передано назад клиенту в качестве значения, возвращаемого функцией `write()`. Важно отметить, что вы должны вернуть фактическое число байт! От этого зависит судьба клиента.

## Этап 7

Пришло время навести лоск для функций `stat()`, `lseek()` и последующих `write()`, аналогично тому, как мы это делали в нашей `io_read()` (и опять мы изменяем смещение в `osb` только в том случае, если это не сообщение типа `_IO_XTYPE_OFFSET`). Однако, поскольку мы выполняем запись в устройство, мы используем при этом константу `IOFUNC_ATTR_MTIME` вместо константы `IOFUNC_ATTR_ETIME`. Флаг `MTIME` означает «время модификации» (`modification time`) — функция `write()` определенно его изменяет.

## Этап 8

Последний этап прост: мы возвращаем константу `EOK`, которая сообщает библиотеке администратора ресурсов, что она должна ответить клиенту. Здесь наша работа закончена. Библиотека администратора ресурсов использует в ответе данные о числе записанных байт, которые мы сохранили с помощью макроса `IO_SET_WRITE_NBYTES()`, и разблокирует клиента. Клиентская функция `write()` возвратит число байт, записанное нашим устройством.

## Простой пример функции `io_devctl()`

Клиентский вызов *devctl()* формально определен так:

```
#include <sys/types.h>
#include <unistd.h>
#include <devctl.h>
```

```
int devctl(int fd, int dcmd, void *dev_data_ptr,
           size_t nbytes, int *dev_info_ptr);
```

Прежде чем рассматривать эту функцию с позиций администратора ресурсов, надо сначала понять, что это за зверь. Функция *devctl()* применяется для «нестандартных» и «управляющих» операций. Например, вы можете записывать данные в звуковую плату (реальные оцифрованные звуковые фрагменты, которые звуковая плата должна будет конвертировать в аналоговый аудиосигнал) и принять решение об изменении числа каналов от одного (моно) до двух (стерео) или об изменении частоты дискретизации данных от стандарта CD (44.1 кГц) к стандарту DAT (48 кГц). Такие вещи было бы правильно делать при помощи функции *devctl()*. При написании администратора ресурсов вы можете решить, что вам вообще не нужны никакие *devctl()*, и что всю необходимую функциональность можно свести к стандартным функциям *read()* и *write()*. С другой стороны, вы можете захотеть использовать как вызовы *devctl()* наряду с вызовами *read()* и *write()*, так и только *devctl()* — это будет зависеть от вашего устройства.

Функция *devctl()* принимает 5 аргументов:

<i>fd</i>	Дескриптор файла администратора ресурсов, которому вы посылаете команду <i>devctl()</i> .
<i>dcmd</i>	Собственно команда — комбинация из двух разрядов направления обмена данными и 30 разрядов команды (см. ниже).
<i>dev_data_ptr</i>	Указатель на область данных, которые передаются, принимаются или и то, и другое.
<i>nbytes</i>	Размер области данных, на которую указывает <i>dev_data_ptr</i> .
<i>dev_info_ptr</i>	Переменная для дополнительной информации, установку которой может выполнить администратор ресурса.

Двумя старшими разрядами команды *dcmd* кодируется направление обмена данными, если он вообще имеет место. Подробности см. выше в описании функций ввода/вывода (параграф «*io\_devctl()*»).

Когда администратор ресурсов принимает сообщение *\_IO\_DEVCTL*, оно обрабатывается вашей функцией *io\_devctl()*. Ниже представлен очень простой пример, который предполагается использовать для настройки

каналов и частоты дискретизации для аудиоустройства, как упоминалось выше.

```
/*
 * io_devctl1.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/neutrino.h>
#include <sys/iofunc.h>

#define DCMD_AUDIO_SET_CHANNEL_MONO    1
#define DCMD_AUDIO_SET_CHANNEL_STEREO  2
#define DCMD_AUDIO_SET_SAMPLE_RATE_CD  3
#define DCMD_AUDIO_SET_SAMPLE_RATE_DAT 4

int io_devctl(resmgr_context_t *ctp, io_devctl_t *msg,
iofunc_ocb_t *ocb) {
    int sts;

    // 1) Проверить, не является ли это обычным
    // POSIX-совместимым devctl()
    if ((sts =
        iofunc_devctl_default(ctp, msg, ocb)) !=
        _RESMGR_DEFAULT) {
        return (sts);
    }

    // 2) Узнать, что за команда, и отработать ее
    switch (msg->i.dcmd) {
    case DCMD_AUDIO_SET_CHANNEL_MONO:
        audio_set_nchannels(1);
        break;
    case DCMD_AUDIO_SET_CHANNEL_STEREO:
        audio_set_nchannels(2);
        break;
    case DCMD_AUDIO_SET_SAMPLE_RATE_CD:
        audio_set_samplerate(44100);
        break;
```

```

case DCMD_AUDIO_SET_SAMPLE_RATE_DAT:
    audio_set_samplerate(48000);
    break;
// 3) Если мы не знаем такой команды, отвергнуть ее
default:
    return (ENOSYS);
}

// 4) Сказать клиенту, что все отработано
memset(msg->o, 0, sizeof(msg->o));
SETIOV(ctp->iiov, &msg->o, sizeof(msg->o));
return (_RESMGR_NPARTS(1));
}

```

## Этап 1

На первом этапе мы снова видим применение вспомогательной функции, на этот раз — функции *iofunc\_devctl\_default()*, которая используется для выполнения всей обработки по умолчанию для *devctl()*. Если вы не поставляете свою версию *io\_devctl()*, а только инициализируете таблицы функций ввода/вывода и установления соединения при помощи *iofunc\_func\_init()*, будет вызвана именно *iofunc\_devctl\_default()*. Мы включаем ее в нашу версию *io\_devctl()*, потому что мы хотим, чтобы она обработала для нас все стандартные POSIX-варианты вызова *devctl()*. Затем мы проверяем возвращаемое значение; если это не *\_RESMGR\_DEFAULT*, значит, функция *iofunc\_devctl\_default()* «обработала» запрос, и нам остается только вернуть это значение, выдав его за «наше».

Если возвращенное значение является константой *\_RESMGR\_DEFAULT*, это говорит нам, что вспомогательная функция *не* обработала запрос, и что мы должны выяснить, является ли он одним из «наших».

## Этап 2

Эта проверка выполняется на этапе 2 при помощи инструкции **switch/case**. Мы просто проверяем значение *dcmd*, которое клиентский код указал во втором параметре функции *devctl()*, на предмет совпадения с какой-нибудь из «наших» команд. Обратите внимание, что для выполнения



фактической «работы» для клиента мы вызываем фиктивные функции *audio\_set\_nchannels()* и *audio\_set\_samplerate()*. Здесь важно отметить, что мы преднамеренно избегаем обсуждения области данных функции *devctl()*. Вы можете подумать: «А что если я хочу установить частоту дискретизации в некое значение *n*? Как это сделать?» На этот вопрос мы ответим в следующем примере *io\_devctl()*, который представлен ниже.

### Этап 3

Этот этап — дань концепции защитного программирования. Мы возвращаем код ошибки *ENOSYS*, чтобы сообщить клиенту, что мы не распознали его запрос.

### Этап 4

Наконец, мы обнуляем возвратную структуру и устанавливаем на нее одноэлементный вектор ввода/вывода. Затем мы возвращаем библиотеке администратора ресурсов единицу (1) через макрос *\_RESMGR\_NPARTS()*, сообщая ей тем самым, что мы возвращаем одноэлементный вектор ввода/вывода. Это и будет возвращено клиенту. Как вариант, мы могли бы применить макрос *\_RESMGR\_PTR()*:

```
// Вместо этого
// 4) Сказать клиенту, что это сработало
memset(&msg->o, 0, sizeof(msg->o));
SETIOV(&ctp->iov, &msg->o, sizeof(msg->o));
return (_RESMGR_NPARTS(1));

//Мы могли бы сделать так:
// 4) Сказать клиенту, что это сработало
memset(&msg->o, 0, sizeof(msg->o));
return (_RESMGR_PTR(ctp, &msg->o, sizeof(msg->o)));
```

Причиной тому, что мы здесь обнулили возвращаемую структуру (вспомните, в примерах *io\_read()* и *io\_write()* мы этого не делали) является то, что в данном случае возвращаемая структура имеет реальное содержимое! (В случае с *io\_read()* мы возвращали только собственно данные и число считанных байт — никакой «возвращаемой структуры» не было; в случае же с *io\_write()* единственным возвращаемым значением было число записанных байт.)

## Пример функции *io\_devctl()*, имеющей дело с данными

В предыдущем примере *io\_devctl()* мы подняли вопрос о том, как устанавливать произвольные значения частоты дискретизации. Очевидно, создание большого количества констант `DCMD_AUDIO_SET_SAMPLE_RATE_*` было бы не самым оптимальным решением — у нас бы просто не хватило разрядности поля *dcmd*.

С клиентской стороны мы будем использовать указатель на частоту дискретизации, *dev\_data\_ptr*, которую мы просто передадим как целое, поэтому поле *nbytes* будет содержать число байт в целом числе (для 32-разрядных машин это 4). Будем предполагать, что для этих целей определена константа `DCMD_AUDIO_SET_SAMPLE_RATE`.

Также неплохо было бы уметь считывать текущее значение частоты дискретизации. Для этого мы также будем использовать переменные *dev\_data\_ptr* и *nbytes*, как описано выше, но в обратном направлении — администратор ресурсов запишет *nbytes* данных в ячейку памяти, на которую указывает *dev\_data\_ptr*, вместо чтения данных из этой ячейки. Предположим, что для этого определена константа `DCMD_AUDIO_GET_SAMPLE_RATE`.

Давайте посмотрим, что в таком случае происходит в функции *io\_devctl()* администратора ресурсов (того, что обсуждалось в предыдущем примере, мы здесь касаться не будем):

```
/*
 * io_devctl2.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <devctl.h>
#include <sys/neutrino.h>
#include <sys/iofunc.h>

#define DCMD_AUDIO_SET_SAMPLE_RATE 1
#define DCMD_AUDIO_GET_SAMPLE_RATE 2

int io_devctl(resmgr_context_t *ctp, io_devctl_t *msg,
  iofunc_ocb_t *ocb) {
  int sts;
```

```

void *data;
int  nbytes;
if ((sts =
    iofunc_devctl_default(ctp, msg, ocb)) !=
    _RESMGR_DEFAULT) {
    return (sts);
}

// 1) Установить указатель на область данных сообщения
data = _DEVCTL_DATA(msg);

// 2) Установить число возвращаемых байт в 0
nbytes = 0;
// Проверить все команды; покажем только те, которые нам
// здесь интересны
switch (msg->i.cmd) {
    ...

// 3) Обработать команду SET
case DCMD_AUDIO_SET_SAMPLE_RATE:
    audio_set_samplerate(*(int*)data);
    break;
// 4) Обработать команду GET
case DCMD_AUDIO_GET_SAMPLE_RATE:
    *(int*)data = audio_get_samplerate();
    nbytes = sizeof(int);
    break;
    ...
}

// 5) Возвратить данные (если есть) клиенту
memset(&msg->o, 0, sizeof(msg->o));
msg->o.nbytes = nbytes;
SETIOV(ctp->iiov, &msg->o, sizeof(msg->o) + nbytes);
return (_RESMGR_NPARTS(1));
}

```

В «шапке» мы декларировали указатель типа `void*` по имени *data* («данные»), которые мы будем использовать в качестве универсального указателя на область данных. Если вы обратитесь к приведенному выше описанию *io\_devctl()*, то вы увидите, что структура данных состоит из объединения заголовков входной и выходной структур, за которым неявно следует область данных. На этапе 1 указатель на эту область данных возвращается макросом *\_DEVCTL\_DATA()*.

## Этап 2

Здесь мы должны указать, сколько байт мы собираемся вернуть клиенту. Я для удобства обнулil переменную *nbytes перед* выполнением каких-либо действий — теперь мне не придется принудительно обнулять ее в каждой ветви *switch/case*.

## Этап 3

Пришло время для команды «set» («установить»). Мы вызываем фиктивную функцию *audio\_set\_samplerate()* и передаем ей значение частоты дискретизации, полученное разыменованием указателя *data* (который мы коварно выставили указателем на целое число... нет, никакого коварства, обычный для Си прием приведения типов). Это ключевой механизм, потому что это и есть наш способ «интерпретации» области данных (клиентского указателя *dev\_data\_ptr*) в соответствии с командой. В более сложном случае вы, наверное, выполнили бы приведение его типа к какой-нибудь структуре побольше вместо простого целого числа. Очевидно, что описания этой структуры на стороне как клиента, так и администратора ресурсов, должны быть идентичными, поэтому лучшим местом для описания такой структуры является заголовочный файл, в котором хранятся ваши командные константы *DCMD\_\**

## Этап 4

Обработка команды «get» («получить») на этапе 4 во многом аналогична (по части приведения типов), кроме того, что на этот раз мы записываем данные в структуру вместо считывания из нее. Заметьте, что мы также присваиваем переменной *nbytes* число байт, которые мы хотим

возвратить клиенту. В случае более сложного доступа к данным вы должны были бы вернуть размер области данных (т.е. если бы эта область была бы структурой, вам нужно было бы вернуть ее размер).

### Этап 5

Наконец, для возврата данных клиенту мы должны вспомнить, что клиент ожидает не только возвращаемые данные (если таковые имеются), но и заголовочную структуру, за которой идут эти данные. Поэтому на этом этапе мы обнуляем заголовочную структуру и устанавливаем число байт (поле *nbytes*) в число байт, которые мы намерены вернуть (вспомните, мы обнуляли это значение ранее). Затем мы создаем одноэлементный вектор ввода/ вывода с указателем на заголовок и расширяем размер заголовка на число возвращаемых байт. В конце мы просто сообщаем библиотеке администратора ресурсов, что мы возвращаем клиенту одноэлементный вектор ввода/вывода.

### Важное замечание

Вспомните рассуждения про следующую за заголовком область данных из примера *io\_write()*, приведенного выше. Мы утверждали, что байты, расположенные сразу после заголовка, *могут как быть полноценными, так и нет* (то есть возможны случаи, когда область данных со стороны клиента была считана лишь частично) — в зависимости от того, сколько данных считала библиотека администратора ресурсов. Затем мы говорили о том, что было бы неэффективно пытаться «сэкономить» лишнюю операцию обмена сообщениями и «повторно использовать» область данных. Однако, в случае с *devctl()* все обстоит несколько иначе, особенно если количество передаваемых данных достаточно невелико (как было и в наших примерах). Здесь у нас есть неплохой шанс того, что данные от клиента были-таки считаны в область данных целиком, и тогда повторное их считывание будет напрасной тратой сил. Узнать, сколько у вас доступно пространства, очень просто: поле *size* («размер») структуры *ctp* содержит число байт, доступных для вас, начиная с параметра *msg*. Размер доступной области данных, расположенной за буфером сообщений, вычисляется как разность между размером буфера сообщений и полем *size* структуры *ctp*:

```
data_area_size = ctp->size - sizeof(*msg);
```

Отметим, что этот размер будет действителен также и в случае *возврата* данных клиенту (как при команде DCMD\_AUDIO\_GET\_SAMPLE\_RATE).

Для всего, что превосходит по размеру выделенную область, вам придется получать данные от клиента так же, как мы это делали в примере с *io\_write()* (см. выше), а также выделить буфер для возврата данных клиенту.

## Дополнительно

Теперь, после того как мы овладели «основами» построения администраторов ресурсов, пришло время рассмотреть более сложные вопросы. К ним относятся:

- расширение ОСВ;
- расширение атрибутной записи;
- блокирование в пределах администратора ресурсов;
- возврат элементов каталога.

## Расширение ОСВ

В ряде случаев у вас может возникнуть необходимость расширения ОСВ. Процедура эта является относительно безболезненной. Обычно ОСВ расширяют дополнительными флагами, характеризующими каждый конкретный *open()*. Один такой флаг можно было бы использовать с обработчиком *io\_unblock()* для кэширования значения флага ядра `_NTO_MI_UNBLOCK_REQ` (подробнее см. параграф «Применение флага `_NTO_MI_UNBLOCK_REQ`» в главе «Обмен сообщениями»).

Для расширения блока ОСВ вам нужно будет обеспечить две дополнительных функции: одну для выделения ОСВ, и одну — для его освобождения. Затем вы должны будете привязать эти две функции к записи точки монтирования. (Да-да, совершенно верно — вам понадобится запись точки монтирования, даже если только для этого.) И наконец, вы должны будете определить ваш собственный тип ОСВ, чтобы все прототипы в программе были корректны.

Давайте рассмотрим сначала описание типа ОСВ, а затем уже поглядим, как переопределяются функции:

```
#define IOFUNC_OCB_T struct my_ocr
#include <sys/iofunc.h>
```

Это сообщает включаемому файлу `<sys/iofunc.h>`, что именованная константа `IOFUNC_OCB_T` теперь указывает на вашу новую усовершенствованную структуру ОСВ.

☞ *Очень важно* иметь в виду, что ваш «расширенный» ОСВ должен содержать «стандартный» ОСВ в качестве своего первого элемента! Это так, потому что вспомогательная библиотека POSIX везде передает указатель на то, что она считает

стандартным ОСВ — о вашем расширенном ОСВ ей ничего не известно, так что первый элемент данных, расположенный по этому указателю, должен соответствовать стандартному ОСВ.

Вот наш расширенный ОСВ:

```
typedef struct my_ocb {
    iofunc_ocb_t normal_ocb;
    int          my_extra_flags;
    ...
} my_ocb_t;
```

А вот код, иллюстрирующий, как переопределяются функции выделения и освобождения ОСВ в записи точки монтирования:

```
// Декларации
iofunc_mount_t mount;
iofunc_funcs_t mount_funcs;

// Задать в записи точки монтирования
// наши функции выделения/освобождения
// _IOFUNC_NFUNCS взята из .h-файла
mount_funcs.nfuncs = _IOFUNC_NFUNCS;

// Новая функция выделения ОСВ
mount_funcs.ocb_calloc = my_ocb_calloc;

// Новая функция освобождения ОСВ
mount_funcs.ocb_free = my_ocb_free;

// Настроить запись точки монтирования
memset(&mount, 0, sizeof(mount));
```

После этого остается только привязать запись точки монтирования к атрибутной записи:

```
...
attr.mount = &mount;
```

Функции *my\_ocb\_calloc()* и *my\_ocb\_free()* отвечают за выделение обнуленного расширенного ОСВ и освобождения ОСВ, соответственно. Вот их прототипы:

```
IOFUNC_OCB_T* my_ocb_calloc(resmgr_context_t *ctp,
    IOFUNC_ATTR_T *attr);

void my_ocb_free(IOFUNC_OCB_T *ocb);
```



Это означает, что функции *my\_osb\_malloc()* передаются одновременно и внутренний контекст администратора ресурсов, и атрибутная запись. Функция отвечает за возврат обнуленного ОСВ. Функция *my\_osb\_free()* получает ОСВ и отвечает за освобождение выделенной под него памяти.

Для этих двух функций имеются два интересных применения (которые ничем не связаны с выполнением расширения блока ОСВ):

- контроль распределения/освобождения блока ОСВ;
- обеспечение более эффективного распределения/освобождения

### ***Контроль за ОСВ***

В этом случае вы можете просто «подключиться» к функциям распределения/освобождения и контролировать использование ОСВ (например, вам может быть необходимо ограничить суммарное количество ОСВ). Это может оказаться полезным, если вы не перехватываете функцию *io\_open()*, но создание (и, возможно, удаление) ОСВ все-таки хотите контролировать.

### ***Более эффективное распределение***

Другое применение для переопределения встроенных библиотечных функций распределения/освобождения ОСВ может заключаться в том, что вы можете захотеть хранить ОСВ в свободном списке вместо использования библиотечных *calloc()* и *free()*. Если вы распределяете и освобождаете ОСВ с большой частотой, это может оказаться более эффективно.

### **Расширение атрибутной записи**

Вы можете захотеть расширить атрибутную запись в случаях, когда вам необходимо хранить дополнительную информацию об устройствах. Поскольку атрибутные записи создаются «по каждому устройству», это означает, что любая дополнительная информация, которую вы сохраните там, будет доступна для всех ОСВ, относящихся к этому устройству (поскольку ОСВ содержит указатель на атрибутную запись). В расширенных атрибутных записях часто хранятся такие параметры как скорость передачи данных по последовательному каналу, и т.п.

Расширять атрибутную запись намного проще, чем ОСВ, потому что атрибутные записи в любом случае распределяются и освобождаются вашим кодом.

Вам нужно будет выполнить тот же трюк с переопределением атрибутной записи в заголовочных файлах, как мы это делали ранее при расширении ОСВ:

```
#define IOFUNC_ATTR_T struct my_attr
#include <sys/iofunc.h>
```

Затем вы фактически определяете содержимое ваших расширенных атрибутных записей. Отметьте, что расширенная атрибутная запись *должна* включать в себя стандартную атрибутную запись *первым* элементом — аналогично случаю с расширением ОСВ (и по тем же самым причинам).

## Блокирование в пределах администратора ресурсов

До настоящего момента мы избегали разговоров о возможности блокирования в пределах администратора ресурсов. Мы предполагали, что у нас есть функция-обработчик (например, *io\_read()*), и что данные будут доступны немедленно. А что если нам придется блокироваться в ожидании данных? Например, выполнение *read()* применительно к последовательному порту может потребовать блокирования до приема символа. Очевидно, что мы не можем предсказать, сколько может продолжаться такое ожидание.

Блокирование в пределах администратора ресурсов базируется на тех же самых принципах, которые мы обсуждали в главе «Обмен сообщениями» — в конце концов, администратор ресурса фактически является сервером, который обрабатывает ряд четко определенных сообщений. Когда прибывает сообщение, соответствующее клиентскому запросу *read()*, оно прибывает вместе с идентификатором отправителя (receive ID), и клиент блокируется. Если у администратора ресурсов есть данные, он просто возвращает их клиенту, как мы уже видели в различных приведенных ранее примерах. Однако, если данные недоступны, администратор ресурсов должен будет удерживать этого клиента в заблокированном состоянии (конечно, если клиент для этой операции определил блокирующий режим), чтобы иметь возможность продолжить обработку других сообщений. Реально это означает, что поток администратора ресурсов, который принял сообщение от клиента, *не должен блокироваться* в ожидании данных — в противном случае это может закончиться для администратора ресурсов огромным числом

заблокированных потоков, каждый из которых ожидал бы данные от некоего устройства.

Правильным решением для этой проблемы является сохранение идентификатора отправителя полученного сообщения в какой-нибудь очереди и возврат из вашего обработчика константы `_RESMGR._NOREPLY`. Это укажет библиотеке администратора ресурсов, что обработка сообщения закончена, но клиента пока разблокировать не надо.

Несколько позже, по готовности данных, вы сможете выбрать идентификатор отправителя нужного клиента и сконструировать ответное сообщение с данными. После этого можно отвечать клиенту.

Вы могли бы также расширить эту концепцию добавлением таймаутов, как мы это делали в главе «Часы, таймеры и периодические уведомления» (параграф «Поддерживаемые сервером тайм-ауты»). Если говорить вкратце, там по истечении некоторого интервала времени клиентский запрос считался «просроченным», и сервер отвечал по сохраненному по идентификатору отправителя неким сообщением об ошибке.

## Возврат элементов каталога

В приведенном ранее примере функции `io_read()` мы уже видели, как происходит возврат данных. Как было упомянуто в описании функции `io_read()` (в разделе «Алфавитный список функций установления соединения и ввода-вывода»), `io_read()` можно возвращать и элементы каталога тоже. Поскольку это может понадобиться далеко не всем, я решил рассмотреть этот вопрос здесь отдельно.

Прежде всего давайте посмотрим, *почему и когда* вашей `io_read()` могло бы понадобиться возвращать элементы каталога, а не «сырые» данные.

Если вы дискретно объявляете элементы в пространстве имени путей, и эти элементы *не* помечены флагом `_RESMGR_FLAG_DIR`, тогда вам не придется возвращать элементы каталога из функции `io_read()`. Если рассматривать это как «файловую систему», то ваш объект будет «файлом». Если же, с другой стороны, вы *указываете* `_RESMGR_FLAG_DIR`, то будет создан объект типа «каталог». Никто, кроме вас, не знает ничего о содержимом этого каталога, поэтому вы должны будете предоставить эти данные. Это и есть ответ на вопрос, почему функции `io_read()` может понадобиться возвращать элементы каталогов.

## Вообще говоря...

Вообще говоря, возврат элементов каталога — это почти то же самое, что и возврат «сырых» данных, за исключением того, что:

- вы должны вернуть *целое* число структур типа `struct dirent`;
- эти структуры `struct dirent` должны быть заполнены.

Первый пункт означает, что вы не можете вернуть, например, семь с половиной структур `struct dirent`. Если восемь структур не вписываются в выделенное пространство, то вы должны будете вернуть только семь элементов.

Второй пункт достаточно очевиден. Он упомянут здесь только потому, что заполнение структуры `struct dirent` может быть несколько «хитрее», чем «сырой» подход к данным в случае с «обычной» `io_read()`.

## Структура `struct dirent` и ее друзья

Давайте взглянем на структуру `struct dirent`, поскольку это именно то, что возвращает функция `io_read()` в случае чтения каталога. Мы также кратко рассмотрим клиентские вызовы, имеющие дело с элементами каталогов, поскольку у них со структурой `struct dirent` существует ряд интересных взаимоотношений.

Чтобы работать с каталогами, клиент использует функции `closedir()`, `opendir()`, `readdir()`, `rewinddir()`, `seekdir()` и `telldir()`.

Обратите внимание на сходство с «нормальными» функций для файлов (и совпадение применяемых типов сообщений):

Функция для работы с каталогами	Функция для работы с файлами	Сообщение
<code>closedir()</code>	<code>close()</code>	<code>IO_CLOSE_DUP</code>
<code>opendir()</code>	<code>open()</code>	<code>_IO_CONNECT</code>
<code>readdir()</code>	<code>read()</code>	<code>_IO_READ</code>
<code>rewinddir()</code>	<code>lseek()</code>	<code>_IO_LSEEK</code>
<code>seekdir()</code>	<code>lseek()</code>	<code>_IO_LSEEK</code>
<code>telldir()</code>	<code>tell()</code>	<code>_IO_LSEEK</code>

Если мы на мгновение вообразим, что функции `opendir()` и `closedir()` будут обработаны для нас автоматически, мы сможем сконцентрироваться

только на сообщениях типа `_IO_READ` и `_IO_LSEEK` и на соответствующих им функциях.

### Смещения

Сообщение `_IO_LSEEK` и соответствующая ему функция применяются для «поиска» (или «перемещения») в пределах файла. С каталогом происходит та же история. Вы можете переместиться к «первому» элементу каталога (как явно задав смещение функции `seekdir()`, так и вызвав функцию `rewinddir()`) переместиться к любому произвольному элементу (используя функцию `seekdir()`), или же узнать свою текущую позицию в списке элементов каталога (вызвав функцию `telldir()`).

Однако, «хитрость» при работе с каталогами состоит в том, что *смещения задаете вы сами, и управление ими тоже всецело лежит на вас*. Это означает, что вы можете назначать смещения элементов в каталоге равными как «0», «2» и так далее, так и «0», «64», «128» и так далее. Единственно, что здесь необходимо предусмотреть — чтобы обработчики `io_lseek()` и `io_read()` одинаково трактовали эти смещения.

В приведенном ниже примере мы предположим, что используется простой подход с номерами «0», «1», «2», и т.д. (Вы могли бы использовать «0», «64», «128», и т.д., если бы эти числа соответствовали, например, неким смещениям на носителе. Выбор за вами!)

### Содержимое

Ну вот, остается «просто» заполнить `struct dirent` «содержимым» нашего каталога. Структура `struct dirent` выглядит так (взято из `<dirent.h>`):

```
struct dirent {
    ino_t      d_ino;
    off_t      d_offset;
    uint16_t   d_reclen;
    uint16_t   d_namelen;
    char       d_name[1];
};
```

Коротко о ее полях:

`d_ino` «Индексный дескриптор» («inode») — уникальный для точки

монтирования порядковый номер, который не может быть нулевым (нуль указывал бы на то, что элемент, соответствующий данному индексному дескриптору, является свободным/пустым).

*d\_offset* Смещение в каталоге, о котором мы только что говорили. В нашем примере это будут обычные числа типа «0», «1», «2», и т.д.

*d\_reclen* Размер структуры **struct dirent** целиком, включая любые добавляемые в нее расширения. Заполнители для выравнивания при вычислении размера учитываются.

*d\_namelen* Число символов в поле *d\_name*, не включая признак конца строки NULL.

*d\_name* Имя элемента каталога, которое должно завершаться признаком конца строки — NULL.

При возврате структур типа **struct dirent** код возврата, который передается клиенту, представляет собой число возвращенных байт.

### *Пример*

Давайте для примера создадим администратора каталогового ресурса */dev/atoz*. Этот администратор регистрирует «файлы» с именами от */dev/atoz/a* до */dev/atoz/z*, чтобы команда *cat*, примененная к любому из них, возвращала соответствующие их именам заглавные буквы. Чтобы понять, как это должно работать, вот пример командно-строковой сессии:

```
# cd /dev

# ls
atoz  null  ptyp2 socket ttyp0 ttyp3
enet0 ptyp0 ptyp3 text   ttyp1 zero
mem   ptyp1 shmem tty    ttyp2

# ls -ld atoz
dr-xr-xr-x 1 root 0 26 Sep 05 07:59 atoz

# cd atoz
# ls
a  e  i  m  q  u  y
b  f  j  n  r  v  z
```

```
c g k o s w
d h l p t x
```

```
# ls -l e
-r--r--r-x 1 root 0 1 Sep 05 07:59 e
```

```
# cat m
M# cat q
Q#
```

Приведенный пример показывает, что в каталоге `/dev` есть каталог `atoz`, и что к нему можно применить команду `ls` и выполнить в него `cd`. Данный каталог `/dev/atoz` имеет размер «26» — мы так задали в нашей программе. Сменив текущий каталог на `atoz` и выполнив еще одну `ls`, получаем его содержимое — файлы с именами от `a` до `z`. Выполнив `ls` для отдельного файла — в данном случае для файла `e` — мы видим, что файл доступен по чтению всем (часть «`-r--r--r--`») и имеет размер, равный 1 байту. Наконец, выполнение нескольких `cat` показывает, что файлы действительно имеют заявленное содержимое. (Отметим, что поскольку файлы содержат только один байт и не содержат символа новой строки, после вывода символа строка не переводится, и приглашение командного интерпретатора оказывается на той же самой строке, что и вывод `cat`.)

Теперь, когда мы знаем нужные характеристики, давайте посмотрим на код. Он разбит на следующие функции:

`main()` и декларации

Основная функция; здесь мы все инициализируем и запускаем наш администратор ресурса.

`my_open()`

Обработчик сообщения `_IO_CONNECT`.

`my_read()`

Обработчик сообщения `_IO_READ`.

`my_read_dir()` и `my_read_file()`

Выполняют фактическую работу функции `my_read()`.

`dirent_size()` и `dirent_fill()`

Сервисные функции для работы со структурой `struct dirent`.

Заметьте, что при том, что разбит на короткие секции, перемежаемые текстовыми пояснениями, архив с полным исходным текстом в виде единого файла можно взять на веб-сайте компании PARSE Software Devices (<http://www.parse.com/>), он называется `atoz.c`.

## *main() и декларации*

Первый приведенный раздел программы представляет собой функцию *main()* и ряд деклараций. Для удобства объявлен макрос *ALIGN()*, который используется функциями *dirent\_size()* и *dirent\_fill()* для выравнивания.

Массив *atoz\_attrs* содержит атрибутные записи, используемые в этом примере для «файлов». Мы объявляем массив из *NUM\_ENTS* элементов, потому что у нас есть *NUM\_ENTS* (то есть 26) файлов — от «a» до «z». Атрибутная запись, применяемая непосредственно для каталога (то есть для */dev/atoz*, объявлена в теле функции *main()* и называется просто *attr*. Обратите внимание на различное содержимое у этих двух типов атрибутных записей:

Файловая атрибутная запись:

Маркируется как обычный файл (константа *S\_IFREG*) с режимом доступа 0444 (это означает, что доступ по чтению имеет каждый, но доступа по записи нет ни у кого). Размер равен «1» — файл содержит только один байт, а именно прописную букву, соответствующую своему имени. Индексные дескрипторы (*inodes*) этих файлов пронумерованы от «1» до «26» включительно (было бы удобнее взять числа от «0» до «25», но цифра «0» зарезервирована).

Каталоговая атрибутная запись:

Маркируется как файл типа «каталог» (константа *S\_IFDIR*) с режимом доступа 0555 (это означает, что доступ по чтению и поиску имеет каждый, но доступа по записи нет ни у кого). Размер определен как «26» — это просто число, взятое по количеству элементов в каталоге. Индексный дескриптор выбран равным «27» — это число заведомо не используется больше ни в одной атрибутной записи.

Обратите внимание, что мы переопределили только поле *open* структуры *connect\_func* и поле *read* структуры *io\_func*. Для всех остальных полей сохранены POSIX-значения по умолчанию.

Наконец, обратите внимание, как мы создали имя */dev/atoz*, используя *resmgr\_attach()*. Наиболее важным здесь является то, что мы применили флаг *\_RESMGR\_FLAG\_DIR*, который сообщает администратору процессов, что он может разрешать запросы на эту точку монтирования и ниже.

```
/*
 * atoz.c
 *
 * /dev/atoz с использованием библиотеки администратора
ресурсов
```



```

*/
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <errno.h>
#include <dirent.h>
#include <limits.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>

#define ALIGN(x) (((x) +3) & ~3)
#define NUM_ENTS 26

static iofunc_attr_t atoz_attrs[NUM_ENTS];

int main (int argc, char **argv) {
    dispatch_t *dpp;
    resmgr_attr_t resmgr_attr;
    resmgr_context_t *ctp;
    resmgr_connect_funcs_t connect_func;
    resmgr_io_funcs_t io_func;
    iofunc_attr_t attr;
    int i;

    // Создать структуру диспетчеризации
    if ((dpp = dispatch_create()) == NULL) {
        perror("Ошибка dispatch_create\n");
        exit(EXIT_FAILURE);
    }

    // Инициализировать структуры данных
    memset(&resmgr_attr, 0, sizeof(resmgr_attr));
    resmgr_attr.nparts_max = 1;
    resmgr_attr.msg_max_size = 2048;

    // Назначить обработчики по умолчанию
    iofunc_func_init(_RESMGR_CONNECT_NFUNCS, &connect_func,
        _RESMGR_IO_NFUNCS, &io_func);

    // Создать и инициализировать атрибутивную запись для

```

```

// каталога...
iofunc_attr_init(&attr, S_IFDIR | 0555, 0, 0);
attr.inode = NUM_ENTS + 1; // 1-26 зарезервированы для
                          // файлов от «a» до «z»
attr.nbytes = NUM_ENTS; // 26 элементов в каталоге

// ...и для имен от «a» до «z»
for (i = 0; i < NUM_ENTS; i++) {
    iofunc_attr_init(&atoz_attrs[i], S_IFREG | 0444, 0, 0);
    atoz_attrs[i].inode = i + 1;
    atoz_attrs[i].nbytes = 1;
}
// Добавить наши функции; нам интересны только io_open
// и io_read
connect_func.open = my_open;
io_func.read = my_read;

// Зарегистрировать префикс
if (resmgr_attach(dpp, &resmgr_attr, "/dev/atoz",
    _FTYPE_ANY, _RESMGR_FLAG_DIR, &connect_func,
    &io_func, &attr) == -1) {
    perror("Ошибка resmgr_attach\n");
    exit(EXIT_FAILURE);
}

// Выделить контекст
ctp = resmgr_context_alloc(dpp);

// Ждать сообщений в вечном цикле
while (1) {
    if ((ctp = resmgr_block(ctp)) == NULL) {
        perror("Ошибка resmgr_block\n");
        exit(EXIT_FAILURE);
    }
    resmgr_handler(ctp);
}
}

```

<i><b>my_open()</b></i>
-------------------------

При том, что функция *my\_open()* очень невелика, в ней есть ряд критических мест. Обратите внимание, как мы принимаем решение о том, был ли ресурс открыт как «файл» или как «каталог», на основе только длины имени пути. Мы можем себе это позволить, потому что знаем, что других каталогов, кроме основного, в этом администраторе ресурсов нет. Если вы захотите расположить ниже вашей точки монтирования дополнительные каталоги, вам придется применить более сложный механизм анализа поля *path* структуры *msg*. В нашем простом примере, если в имени пути ничего нет, то мы знаем, что это каталог. Также обратите внимание на чрезвычайно упрощенную проверку корректности имени пути: мы просто проверяем, что у нас действительно только один символ, и что он лежит в диапазоне от «а» до «z» включительно. Опять же, в случае более сложного администратора ресурсов вам пришлось бы выполнять синтаксический анализ имени, следующего за зарегистрированной точкой монтирования.

Теперь о наиболее важной особенности. Обратите внимание, как мы использовали для выполнения всей нашей работы функции POSIX-уровня по умолчанию! Функция *iofunc\_open\_default()* обычно размещается в таблице функций установления соединения в той же самой ячейке, которую сейчас занимает наша функция *my\_open()*. Это означает, что они принимают одинаковый набор аргументов!

Все, что мы должны сделать — это решить, *какую* атрибутную запись мы хотим связать с ОСВ, создаваемой функцией по умолчанию: либо каталоговую (в этом случае мы передаем *attr*), либо одну из 26 имеющихся у нас файловых (тогда мы передаем соответствующий элемент *atoz\_attrs*). Это ключевой момент, поскольку обработчик, который вы помещаете в слот *open* в таблице функций установления соединения, действует как «швейцар» по отношению ко всем последующим запросам к вашему администратору ресурса.

```
static int my_open(resmgr_context_t *ctp, io_open_t *msg,
    iofunc_attr_t *attr, void *extra) {
    if (msg->connect.path[0] == 0) {
        // Каталог (/dev/atoz)
        return (iofunc_open_default(ctp, msg, attr, extra));
    } else if (msg->connect.path[1] == 0 &&
        (msg->connect.path[0] >= 'a' &&
        msg->connect.path[0] <= 'z')) { // Файл (/dev/atoz/[a-z])
        return
            (iofunc_open_default(ctp, msg, atoz_attrs +
                msg->connect.path[0] - 'a', extra));
    }
```

```

    } else {
        return (ENOENT);
    }
}

```

### ***my\_read()***

В функции *my\_read()*, чтобы решить, какие операции надо выполнить, мы анализируем поле `mode` атрибутной записи. Если макрос *S\_ISDIR()* говорит, что это каталог, мы вызываем функцию *my\_read\_dir()*; если макрос *S\_ISREG()* говорит, что это файл, мы вызываем функцию *my\_read\_file()*. (Отметим, что если мы не можем разобрать, что это такое, мы возвращаем клиенту `EBADF`, указывая ему этим, что что-то здесь не так.)

Приведенный ниже код ничего не знает о наших специальных устройствах, да ему, собственно, и дела до них нет — он просто принимает решение на основе стандартных общеизвестных данных.

```

static int my_read(resmgr_context_t *ctp, io_read_t *msg,
    iofunc_ocb_t *ocb) {
    int sts;

    // Использовать вспомогательную функцию для проверки
    // корректности
    if ((sts =
        iofunc_read_verify(ctp, msg, ocb, NULL)) != EOK) {
        return (sts);
    }

    // Решить, надо ли читать «файл» или «каталог»
    if (S_ISDIR(ocb->attr->mode)) {
        return (my_read_dir(ctp, msg, ocb));
    } else if (S_ISREG(ocb->attr->mode)) {
        return (my_read_file(ctp, msg, ocb));
    } else {
        return (EBADF);
    }
}

```

### ***my\_read\_dir()***

Вот тут-то все веселье и начинается. С точки зрения высокого уровня, мы выделяем буфер (он называется *reply\_msg*), в котором собираемся разместить результат операции. Затем мы используем *dp*, чтобы «прогуляться» по буферу, заполняя его по ходу дела элементами **struct dirent**. Вспомогательная подпрограмма *dirent\_size()* применяется, чтобы определить, есть ли у нас место в буфере для еще одного элемента. Вспомогательная подпрограмма *dirent\_fill()* кладет элемент в буфер. (Отметим, что эти подпрограммы не входят в библиотеку администратора ресурсов; мы обсудим их ниже.)

На первый взгляд этот код может показаться неэффективным; мы используем функцию *sprintf()* для создания двухбайтового имени файла (символ имени файла и признак конца строки NULL) в буфере длиной *\_POSIX\_PATH\_MAX* (то есть 256) байт. Это делается для того, чтобы сделать код по возможности универсальным.

Наконец, обратите внимание, что мы используем поле *offset* в ОСВ для указания, для какого конкретного файла мы в данный момент генерируем структуру **struct dirent**. Это означает, что мы также должны корректировать поле *offset* всякий раз, когда возвращаем данные.

Возврат данных клиенту осуществляется «обычным» способом при помощи функции *MsgReply()*. Заметьте, что поле состояния (status) функции *MsgReply()* используется для указания числа отправленных клиенту байт.

```
static int my_read_dir(resmgr_context_t *ctp,
    io_read_t *msg, iofunc_ocb_t *ocb) {
    int nbytes;
    int nleft;
    struct dirent *dp;
    char *reply_msg;
    char fname[_POSIX_PATH_MAX];
    // Выделить буфер для ответа
    reply_msg = calloc(1, msg->i.nbytes);
    if (reply_msg == NULL) {
        return (ENOMEM);
    }

    // Назначить выходной буфер
    dp = (struct dirent *)reply_msg;

    // Осталось «nleft» байт
    nleft = msg->i.nbytes;
    while (ocb->offset < NUM_ENTS) {
```

```

// Создать имя файла
sprintf(fname, "%c", ocb->offset + "a");
// Проверить, насколько велик результат
nbytes = dirent_size(fname);
// Есть место?
if (nleft - nbytes >= 0) {
    // Заполнить элемент каталога и увеличить указатель
    dp =
        dirent_fill(dp, ocb->offset + 1, ocb->offset, fname);
    // Увеличить смещение ОСВ
    ocb->offset++;
    // Учесть, сколько байт мы использовали
    nleft -= nbytes;
} else {
    // Места больше нет, остановиться
    break;
}
}

// Возвращаемся обратно к клиенту
MsgReply(ctp->rcvid, (char*)dp - reply_msg, reply_msg,
(char*)dp - reply_msg);

// Освободить буфер
free(reply_msg);
// Сказать библиотеке, что мы уже ответили сами
return (_RESMGR_NOREPLY);
}

```

### ***my\_read\_file()***

В функции *my\_read\_file()* мы видим код, почти аналогичный простому примеру функции чтения, который приведен выше в данном разделе. Единственная странная вещь, которую мы здесь делаем — поскольку мы знаем, что возвращается всегда только один байт данных, значит, если параметр *nbytes* не равен нулю, то он должен быть равен единице (и ничему другому). Таким образом, мы можем создавать данные, подлежащие возврату, непосредственным заполнением символьной переменной *string*. Обратите внимание, как мы используем поле *inode* атрибутной записи для

определения, какие данные возвращать. Это обычный прием для администраторов, обслуживающих несколько ресурсов. Дополнительным трюком было бы расширить атрибутную запись (мы говорили об этом в разделе «Расширение атрибутной записи») и хранить непосредственно в ней либо сами данные, либо указатель на них.

```
static int my_read_file(resmgr_context_t *ctp,
    io_read_t *msg, iofunc_ocb_t *ocb) {
    int nbytes;
    int nleft;
    char string;

    // Тут нет никаких xtype...
    if ((msg->i.xtype & _IO_XTYPE_MASK) != _IO_XTYPE_NONE) (
        return (ENOSYS);
    }

    // Выяснить, сколько байт осталось...
    nleft = ocb->attr->nbytes - ocb->offset;

    // ...и сколько мы можем вернуть клиенту
    nbytes = min(nleft, msg->i.nbytes);
    if (nbytes) {
        // Создать ответную строку
        string = ocb->attr->inode - 1 + "A";
        // Возвратить ее клиенту
        MsgReply(ctp->rcvid, nbytes, &string + ocb->offset,
            nbytes);
        // Обновить флаги и смещение
        ocb->attr->flags |=
            IOFUNC_ATTR_ATIME | IOFUNC_ATTR_DIRTY_TIME;
        ocb->offset += nbytes;
    } else {
        // Возвращать нечего, индицировать конец файла
        MsgReply(ctp->rcvid, EOK, NULL, 0);
    }

    // Уже ответили сами
    return (_RESMGR_NOREPLY);
}
```

### *dirent\_size()*

Вспомогательная подпрограмма *dirent\_size()* просто вычисляет число байт, необходимое для структуры **struct dirent**, с учетом ограничений по выравниванию. Опять же, для нашего простого администратора ресурсов здесь имеет место небольшое искусственное упрощение, потому что мы точно знаем, каков размер каждого элемента каталога — все имена файлов имеют длину ровно один байт. Однако, как бы там ни было, это все равно полезная служебная подпрограмма.

```
int dirent_size(char *fname) {  
    return (ALIGN(sizeof(struct dirent) - 4 + strlen(fname)));  
}
```

### *dirent\_fill()*

И, наконец, вспомогательная подпрограмма *dirent\_fill()* применяется для помещения передаваемых ей значений (а именно — полей *inode*, *offset* и *fname*) в также передаваемый ей элемент каталога. В порядке щедрости она также возвращает указатель на адрес (с учетом выравнивания), с которого должен начинаться следующий элемент каталога.

```
struct dirent* dirent_fill(struct dirent *dp, int inode,  
    int offset, char *fname) {  
    dp->d_ino = inode;  
    dp->d_offset = offset;  
    strcpy(dp->d_name, fname);  
    dp->d_namelen = strlen(dp->d_name);  
    dp->d_reclen =  
        ALIGN(sizeof(struct dirent) - 4 + dp->d_namelen);  
    return ((struct dirent*)((char*)dp + dp->d_reclen));  
}
```



## Резюме

Написание администратора ресурсов — наиболее сложная задача из описанных в этой книге.

Администратор ресурсов — это сервер, который воспринимает ряд четко определенных сообщений. Эти сообщения можно подразделить на две категории:

Сообщения установления соединения

Относятся к операциям с именами путей, с их помощью можно устанавливать контекст для дальнейшей работы.

Сообщения ввода/вывода

Всегда следуют за сообщениями установления соединения и указывают, какую реальную работу нужно сделать для клиента (например, *stat()*).

Действия администратора ресурса управляются функциями пула потоков (мы обсуждали их в главе «Процессы и потоки») и функциями интерфейса диспетчеризации.

QSSL в библиотеке администратора ресурсов предоставляет ряд вспомогательных POSIX-функций, которые выполняют львиную долю работы по обслуживанию клиентских сообщений установления соединения и ввода/вывода, поступающих в администратор ресурсов.

Существует ряд структур данных, относящихся к клиентам и объявляемым администраторами ресурсов устройствам, которые необходимо принимать во внимание:

OCB (блок открытого контекста)

Выделяется при каждом «открытии» ресурса; содержат контекст для клиента (например, текущее смещение *lseek()*).

Атрибутная запись

Выделяется для каждого устройства; содержит информацию об устройстве (например, размер устройства, режимы доступа, и т.д.).

Запись точки монтирования

Распределяется на базисе по каждому администратору ресурса и содержит полную информацию о характеристиках администратора ресурса.

Клиент общается с администраторами ресурсов посредством обмена сообщениями, разрешив имя пути (с помощью функции *open()* и других вызовов) в соответствующий нужному администратору дескриптор узла, идентификатор процесса, идентификатор канала и обработчик (*handle*).

И наконец, вы наделяете администратор ресурса необходимой вам функциональностью, переопределяя некоторые из вызовов в таблицах функций установления соединения и функций ввода/вывода.

# **Приложение А**

## **Переход с QNX4 на QNX/Neutrino**

## Переход с QNX4 на QNX/Neutrino

В данном приложении мы рассмотрим предыдущую операционную систему разработки QSSL, QNX4, и сравним ее с QNX/Neutrino. Данное приложение будет вам интересно главным образом в том случае, если вы уже являетесь пользователем QNX4 и хотите узнать.

- что такого замечательного в QNX/Neutrino?
- какие сложности связаны с переносом программного обеспечения в QNX/Neutrino?

а также если вы разрабатываете (или портируете) программное обеспечение для обеих операционных систем.

## Сходства

Давайте начнем со общих черт этих двух операционных систем:

- архитектура на основе обмена сообщениями;
- распределенный обмен сообщениями в сети;
- реальное время;
- микроядерная архитектура;
- защита памяти на уровне процессов;
- POSIX-совместимость;
- относительно простая модель «драйвера устройства»;
- встраиваемость.

Заметьте, что хоть часть вышеперечисленных базовых свойств действительно подобны в этих двух ОС, в целом QNX/Neutrino обеспечивает более расширенную поддержку. Например, поддержки POSIX в QNX/Neutrino больше, чем в QNX4, — просто потому что многие из стандартов этой серии на момент выхода QNX4 были еще только в стадии разработки. И хотя в процессе разработки QNX/Neutrino в состоянии разработки находилась гораздо меньшая их часть, постоянно появляются новые. Эта бесконечная игра в догонялки.

## Улучшения

Теперь, когда вы выяснили, что общего между этими двумя ОС, давайте посмотрим, каковы преимущества QNX/Neutrino перед QNX4:

- большее число поддерживаемых стандартов POSIX;

- лучше встраивается;
- ядро лучше конфигурируется для других аппаратных платформ;
- поддержка многопоточности;
- более простая модель драйвера устройства;
- переносимая архитектура (в настоящее время поддерживаются, кроме x86, процессоры MIPS и PPC) (а теперь уже и ARM, StrongARM и SuperH-4 — прим. ред.),
- поддерживает SMP;
- лучше документирована.

При том, что некоторые из этих усовершенствований «вне сравнения», поскольку аналогов для них в QNX4 нет, а значит, нет и проблем совместимости (например, многопоточность POSIX не поддерживалась в QNX4), некоторые из проблем потребовали внесения в ОС кардинальных изменений. Сначала я вкратце упомяну, какие классы изменений были необходимы, а затем мы подробно рассмотрим возникшие в связи с этим проблемы совместимости и предложения о том, как переносить программы в QNX/Neutrino (или заставить программу работать в обеих ОС).

### ***Встраиваемость***

В QNX/Neutrino коренным образом пересмотрена стратегия встраивания. В первоначальном варианте QNX4 встраиваемость обеспечивалась только частично. Затем появилась QNX/Neutrino, которая с самого начала разрабатывалась как встраиваемая. В качестве премии, QNX4 подверглась некоторой модернизации на основе опыта, полученного с QNX/Neutrino, и теперь является в гораздо более значительной степени встраиваемой, чем ранее. Как бы там ни было, по части встраиваемости QNX/Neutrino и QNX4 отличаются, как день и ночь. QNX4 не содержит никакой реальной поддержки таких вещей как:

- исходящие вызовы ядра (kernel callouts) (прерывание, таймер);
- настройка стартового кода;
- образная файловая система.

а вот в QNX/Neutrino все это есть. Подробное описание методик встраивания QNX/Neutrino приведено в книге «Построение встраиваемых систем», входящей в комплект документации.

### ***Поддержка многопоточности***

В QNX4 есть функция, называемая *tfork()*, которая позволяет вам организовать «многопоточность», создавая процесс с сегментами кода и данных, отображенными в то же адресное пространство, что и у родительского. Создание процесса и потом приведение его к «потокopodobному» виду дает иллюзию создания потока. И хотя в системе обновлений QSSL содержится библиотека поддержки потоков для QNX4, само ядро потоки непосредственно не поддерживает.

В QNX/Neutrino для организации многопоточности применяется POSIX-модель «*pthread*». Это означает, что там вы увидите (и уже видели в данной книге) знакомые вызовы функций типа *pthread\_create()*, *pthread\_mutex\_lock()*, и т.п.

### Обмен сообщениями

При том что воздействие потоков на обмен сообщениями может показаться минимальным, использование потоков привело к фундаментальному изменению реализации механизма обмена сообщениями (не самой *концепции* SEND/RECEIVE/REPLY, а именно ее *реализации*).

В QNX4 сообщения адресовались идентификатору процесса. Чтобы отправить сообщение, нужно было просто найти идентификатор процесса-адресата и выполнить *Send()*. Чтобы сервер мог принять сообщение в QNX4, он должен был вызвать *Receive()*. Это блокировало его до прибытия сообщения. Затем сервер отвечал с помощью функции *Reply()*.

В QNX/Neutrino обмен сообщениями устроен так же, только при этом используются другие имена функций. Что изменилось, так это сам механизм. Теперь, прежде чем вызывать стандартные функции обмена сообщениями, клиент должен сперва создать соединение с сервером. А сервер, прежде чем он сможет выполнять стандартные функции обмена сообщениями, должен сначала создать канал.

Заметьте, что существовавшая в QNX4 функция *Creceive()*, которая выполняла неблокирующий вызов *Receive()*, в QNX/Neutrino отсутствует. Мы вообще не одобряем такие «опрашивающие» функции, особенно когда можно запустить поток; впрочем, если вы настаиваете на выполнении неблокирующего вызова *MsgReceive()*, посмотрите главу «Часы, таймеры и периодические уведомления», раздел «Тайм-ауты ядра». Вот соответствующий пример кода в качестве краткого пояснения:

```
TimerTimeout(CLOCK_REALTIME, _NTO_TIMEOUT_RECEIVE,  
             NULL, NULL, NULL);
```

```
rcvid = MsgReceive(...
```

### ***Импульсы и события***

В QNX4 было нечто по имени «прокси». Прокси лучше всего описывается как «законсервированное» (т.е. неизменяемое) сообщение, которое можно отослать владельцу этой (почему-то исторически сложилось, что в русском языке термин «прокси» женского рода; возможно, из-за распространенного сленгового произношения «прокся» #:o) — *прим. ред.*) прокси от имени процесса или ядра (например, по срабатыванию таймера или из обработчика прерывания). Прокси является неблокирующей для отправителя и принимается точно также, как и любое другое сообщение. Распознать сообщение прокси (то есть отличить его от обычного сообщения) можно было либо проанализировав его содержимое (не самый надежный способ, поскольку процесс тоже может передать что-нибудь с подобным содержимым), либо проверив идентификатор процесса, от которого оно было получено. Если идентификатор процесса совпадает с идентификатором прокси — значит, это прокси, потому что идентификаторы процессов и прокси берутся из того же самого пула номеров и не пересекаются.

(На самом деле, документация по QNX4 вносит в понятие «прокси» страшную путаницу, которая требует пояснения. Если бы прокси являлась именно *сообщением*, как это записано в определении прокси, обороты типа «получить сообщение от прокси» были бы лишены смысла. Само по себе слово «проху» переводится как «промежуточный агент». Если внимательно прочитать главу «IPC via proxies» книги «QNX4 System Architecture», становится ясно, что прокси — это не само сообщение, а специализированный «*квазипроцесс*», не обладающий ресурсами, но имеющий идентификатор (который, разумеется, по понятным причинам не перекрывается с другими идентификаторами процессов) и способный к обмену специализированными сообщениями. Применяя к прокси с определенным идентификатором функцию *Trigger()*, процесс в QNX4 фактически делает этому квазипроцессу специализированный *Send()*, в ответ на который мгновенно следует *Reply()*, поэтому отправитель (вызвавший *Trigger()*) и не блокируется. Затем прокси отправляет своему процессу-владельцу свое предопределенное сообщение; если владелец прокси в этот момент не является RECEIVE-блокированным, сообщение становится в очередь. Таким образом, правильнее было бы говорить

«переключить прокси» вместо «отправить прокси» (о вызове *Trigger()*) и «получить *сообщение* прокси» вместо «получить прокси» — прим. ред.).

QNX/Neutrino расширяет концепцию прокси введением «импульсов». Импульсы по-прежнему являются неблокирующими сообщениями, и их по-прежнему можно переслать либо от потока к потоку, либо от служебной функции ядра (например, от того же таймера или обработчика прерывания) к потоку. Различие состоит в том, что в то время как прокси имели фиксированное содержимое, импульсы имеют фиксированную длину, но содержимое их может быть изменено отправителем в любой момент. Например, обработчик прерываний (ISR) может сохранить в импульсе какие-либо данные и затем отправить этот импульс потоку.

В QNX4 некоторые сервисы были наделены способностью послать как сигнал, так и прокси, в то время как другие сервисы были наделены способностью послать либо только одно, либо только другое. Мало того, обычно это выполнялось несколькими различными способами. Например, чтобы доставить сигнал, вы должны были применить функцию *kill()*. Чтобы доставить прокси или сигнал по срабатыванию таймера, вы должны были использовать отрицательный номер сигнала (для указания на то, что это прокси) или положительный номер сигнала (для указания на то, что это сигнал). И, наконец, обработчик прерываний мог доставлять *только* прокси.

В QNX/Neutrino все это было абстрагировано в расширение POSIX-структуры `struct sigevent`. Все, что использует и возвращает `struct sigevent`, может использовать как сигнал, так и импульс.

На самом деле, функциональность `struct sigevent` была расширена вплоть до возможности создания потока! Мы говорили об этом в Главе «Часы, таймеры и периодические уведомления» в разделе «Уведомление созданием потока».

### *Модель драйвера устройства*

В самых ранних версиях QNX (семейство QNX2) программирование драйверов устройств было чем-то из области черной магии. В QNX4 это тоже поначалу была загадочная вещь, но затем, в конце концов, появилось несколько примеров программ. В QNX/Neutrino этому вопросу посвящены книги и учебные курсы. И, как оказалось, модели драйверов в QNX/Neutrino и QNX4 на архитектурном уровне достаточно сходны. В то время как в QNX4 была крошечная путаница вокруг того, что такое «функции установления соединения» и что такое «функции ввода-вывода», в



QNX/Neutrino все четко разделено. Также под QNX4 вы (разработчик драйвера устройства) должны были выполнять большую часть работы (программирование основного цикла обработки сообщений, сопоставление контекста каждому сообщению ввода-вывода, и т.д.) самостоятельно. В QNX/Neutrino все это в значительной степени упрощено введением библиотеки администратора ресурсов.

### ***Поддержка MIPS и PPC***

Одно из самых существенных отличий QNX/Neutrino от QNX4 в отношении встраиваемости состоит в том, что QNX/Neutrino поддерживает процессоры MIPS и PPC (Power PC). QNX4 изначально была «дома» на IBM PC с их BIOS и «очень» стандартным набором аппаратных средств, QNX/Neutrino же одинаково по-домашнему чувствует себя на разных платформах с BIOS (или монитором ПЗУ) или без нее, а также на нестандартной аппаратуре, комплектация которой выбирается изготовителем (и часто без учета требований ОС). Это означает, что ядро QNX/Neutrino должно было обеспечивать поддержку исходящих вызовов (callouts), чтобы вы могли, например, задать свой тип контроллера прерываний и работать на этих аппаратных средствах без необходимости приобретения лицензии на исходный текст операционной системы.

Другая группа изменений, которые вы заметите при переносе приложений из QNX4 в QNX/Neutrino, особенно на другие платформы, касается того, что процессоры MIPS и PPC уж больно вычурны по части выравнивания. Вы не можете обращаться к N-байтовому объекту иначе чем по адресу, кратному N. При работе на x86 (с выключенным флагом выравнивания) вы бы волей-неволей обратились к памяти. Изменив вашу программу так, чтобы все структуры были правильно выровнены (для не-x86 процессоров), вы также заметите, что ваша программа после этого на x86 станет выполняться быстрее, потому что x86 быстрее обращается к выровненным данным.

Другое, что так часто не дает людям жить — это порядок следования байт, прямой (big-endian) или обратный (little-endian). (Кому интересна этимология этих терминов, загляните в английский оригинал «Путешествий Гулливера» Джонатана Свифта :- ) — прим. ред.). У процессора x86 возможен только один порядок следования байт, и он обратный. Процессоры MIPS и PPC допускают оба порядка следования байт, т. е. могут работать как с прямым, так и с обратным. Кроме того, процессоры MIPS и PPC являются процессорами типа RISC (с

сокращенным набором команд), что означает, что некоторые операции типа `|=` в Си (установка бита) могут и не быть атомарными. Это иногда приводит к жутким последствиям! Список вспомогательных функций, которые обеспечивают атомарность выполняемой операции, приведен в файле `<atomic.h>`.

### *Поддержка SMP*

Существующие версии QNX4 работают только на однопроцессорных системах, в то время как QNX/Neutrino уже на момент публикации первого издания этой книги обеспечивала поддержку SMP по меньшей мере на архитектуре x86. SMP дает значительные преимущества, особенно в многопоточной ОС, но это одновременно и гораздо более серьезный пистолет для простреливания ноги (кто любопытен, поищите в Интернет по ключевой фразе «shoot yourself in the foot» — *прим. ред.*).

Например, в однопроцессорной машине обработчик прерывания (ISR) может вытеснить поток, но *наоборот никогда не бывает*. В однопроцессорной машине бывает полезно представить себе, что потоки якобы выполняются одновременно, хотя в действительности это не так.

В блоке SMP поток и обработчик прерывания могут работать одновременно, да и несколько потоков тоже могут работать одновременно. Так что SMP — это не только превосходная рабочая станция, но и неплохое средство тестирования программного обеспечения — если вы сделали какие бы то ни было «плохие» предположения о защите в многопоточной среде, в SMP-системе они однажды обязательно выплывут.

☞ Для иллюстрации того, насколько это верно, один пример. Одна из ошибок в ранней внутренней версии поддержки SMP проявлялась в «окне» длиной в один машинный цикл! То, что для однопроцессорной машины было запрограммировано как атомарная операция «чтение/модификация/запись», в SMP-блоке стало допускать вмешательство в ход операции второго процессора, выполняющего «сравнение/обмен».

## Философия переноса программ

Давайте теперь взглянем на все «сверху». Здесь мы рассмотрим:

- обмен сообщениями и систему «клиент/сервер»;
- обработчики прерываний (ISR).

### Анализ обмена сообщениями

В QNX4 клиент мог найти сервер двумя способами:

- используя глобальное пространство имен;
- выполнение *open()* в отношении администратора ввода/вывода.

### *«Клиент/сервер» с использованием глобального пространства имен*

Если взаимоотношения «клиент/сервер, которые вы переносите, базируются на глобальном пространстве имен, тогда клиент использует:

*qnx\_name\_locate()*

а сервер регистрирует свое имя при помощи:

*qnx\_name\_attach()*

В этом случае у вас есть два выбора. Вы можете либо попробовать сохранить вариант с глобальными именами, либо модифицировать клиента и сервер так, чтобы они работали подобно стандартному администратору ресурсов.

Я рекомендую вам последний вариант, поскольку именно этот вариант характерен для QNX/Neutrino — сводить все к администраторам ресурсов, а не пытаться навесить кусок администратора ресурсов на службу глобальных имен.

Модификация будет достаточно проста. Скорее всего, клиентская сторона вызывает функцию, либо возвращающую идентификатор серверного процесса, либо создающую виртуальный канал («VC» — Virtual Circuit) от клиентского узла к удаленному узлу сервера. В обоих случаях как идентификатор процесса, так и идентификатор виртуального канала к удаленному процессу определяются при помощи *qnx\_name\_locate()*. «Магическим амулетом», связывающим клиента с сервером, здесь является специальная разновидность идентификатора процесса (мы считаем идентификатор виртуального канала идентификатором процесса, поскольку

он берется из того же пула номеров и со всех точек зрения выглядит как идентификатор процесса).

Преодолеть основное различие можно было бы, возвращая вместо идентификатора процесса идентификатор соединения. Поскольку клиент в QNX4, вероятно, не анализирует идентификаторы процессов (да и зачем? Так, просто число), вы могли бы «обмануть» его, применив к «глобальному имени» функцию *open()*. В этом случае, однако, глобальное имя должно было бы быть точкой монтирования, зарегистрированной администратором ресурса в качестве своего «идентификатора». Вот, например, типовой пример клиента QNX4, взятый из моей серверной библиотеки CLID:

```
/*
 * CLID_Attach(ServerName)
 *
 * Эта подпрограмма отвечает за установление соединения
 * с сервером CLID.
 *
 * Возвращает PID сервера CLID или идентификатор
 * виртуального канала к нему.
 */

// Сюда запишется имя - для других библиотечных вызовов
static char CLID_serverName(MAX_CLID_SERVER_NAME + 1);

// Сюда запишется идентификатор сервера
CLID static int clid_pid = -1;

int CLID_Attach(char *serverName) {
    if (ServerName == NULL) {
        sprintf(CLID_serverName, "/PARSE/CLID");
    } else {
        strcpy(CLID_serverName, serverName);
    }
    clid_pid = qnx_name_locate(0, CLID_serverName,
        sizeof(CLID_ServerIPC), NULL);
    if (clid_pid != -1) {
        CLID_IPC(CLID_MsgAttach); // Послать сообщение ATTACH
        return (clid_pid);
    }
    return (-1);
}
```

Вы могли бы изменить это на следующее:

```
/*
 * CLID_Attach(serverName), версия для QNX/Neutrino
 */
int CLID_Attach(char *serverName) {
    if (ServerName == NULL) {
        sprintf(CLID_serverName, "/PARSE/CLID");
    } else {
        strcpy(CLID_serverName, serverName);
    }
    return (clid_pid = open(CLID_serverName, O_RDWR));
}
```

И клиент ничего бы не заметил.

☞ Два замечания по реализации. В качестве зарегистрированного префикса администратора ресурса я просто оставил имя по умолчанию («/PARSE/CLID»). Вероятно, лучше было бы взять имя «/dev/clid», но насколько вы хотите следовать POSIX — это ваше личное дело. В любом случае, это незначительное изменение, и оно мало связано с тем, что здесь обсуждается.

Второе замечание касается того, что я по-прежнему назвал дескриптор файла *clid\_pid*, хотя реально ему бы теперь следовало называться *clid\_fd*. Это, опять же, вопрос стиля и касается только того, сколько различий вы хотите иметь между версиями кода для QNX4 и QNX/Neutrino.

В любом случае, того чтобы данная программа была переносима в обе ОС, вам придется выделить код соединения с сервером в отдельную функцию — как я это сделал выше с функцией *CLID\_Attach()*.

В какой-то момент клиент должен будет выполнить собственно операцию отправки сообщения. Здесь все становится несколько сложнее. Поскольку отношения клиент/сервер *не* основаны на отношениях с администраторами ввода/вывода, клиент обычно создает «нестандартные» сообщения. Снова пример из CLID-библиотеки («незащищенный» клиентский вызов здесь — *CLID\_AddSingleNPANXX()*, я также включил функции *checkAttach()* и *CLID\_IPC()* для того, чтобы продемонстрировать фактическую передачу сообщений и логику проверок):

```
/*
```

```

    * CLID_AddSingleNPANXX(npa, nxx)
*/

int CLID_AddSingleNPANXX(int npa, int nxx) {
    checkAttach();
    CLID_IPCData.npa = npa;
    CLID_IPCData.nxx = nxx;
    CLID_IPC(CLID_MsgAddSingleNPANXX);
    return (CLID_IPCData.returnValue);
}

/*
 * CLID_IPC(номер_сообщения_IPC)
 *
 * Эта подпрограмма вызывает сервер с глобальным буфером
 * CLID_IPCData и заносит в него номер сообщения,
 * переданный ей в качестве аргумента.
 *
 * Если сервера нет, эта подпрограмма установит
 * поле returnValue в CLID_NoServer. Остальные
 * поля остаются как есть.
*/
void CLID_IPC(int IPCMessage) {
    if (clid_pid == -1) {
        CLID_IPCData.returnValue = CLID_NoServer;
        return;
    }
    CLID_IPCData.serverFunction = IPCMessage;
    CLID_IPCData.type = 0x8001;
    CLID_IPCData.subtype = 0;
    if (Send(clid_pid, &CLID_IPCData, &CLID_IPCData,
        sizeof(CLID_IPCData), sizeof(CLID_IPCData))) {
        CLID_IPCData.returnValue = CLID_IPCError;
        return;
    }
}

void checkAttach() {
    if (clid_pid == -1) {
        CLID_Attach(NULL);
    }
}

```

```
}  
}
```

Как вы видите, функция *checkAttach()* применяется для проверки существования соединения с сервером CLID. Если бы соединения не было, это было бы подобно запросу *read()* по несуществующему дескриптору файла. В моем варианте программы функция *checkAttach()* создает соединение автоматически. Это как если бы функция *read()* определила, что дескриптор файла некорректен, и сама создала бы корректный. Еще один вопрос стиля.

Обмен специализированными сообщениями происходит в функции *CLID\_IPC()*. Она берет значение глобальной переменной *CLID\_IPCData* и пробует переслать его серверу, используя функцию *QNX4 Send()*.

Специализированные сообщения могут быть обработаны о из двух способов. Можно:

1. транслировать их в стандартные вызовы функций POSIX основанные на файловых дескрипторах;
2. инкапсулировать их в сообщение типа *devctl()*, либо в специализированное сообщение, используя тип *\_IO\_MSG*.

В обоих случаях вы перестраиваете клиента на обмен сообщениями стандартными для администраторов ресурсов средствами. Как? У вас нет файлового дескриптора? Есть только идентификатор соединения? Или наоборот? Ну, это как раз не проблема. В QNX/Neutrino *дескриптор файла* в действительности является *идентификатором соединения*!

### ***Трансляция сообщений в стандартные вызовы POSIX на основе файловых дескрипторов***

В случае CLID-сервера это не вариант. Не существует стандартного POSIX-вызова на основе файлового дескриптора, который мог бы «добавить к администратору ресурса CLID пару NPA/NXX». Однако, существует стандартный механизм *devctl()*, так что если ваши отношения клиент/сервер требуют такой формы, смотрите ниже.

Прежде чем броситься реализовывать этот подход (трансляцию в стандартные сообщения на основе файловых дескрипторов), давайте остановимся и подумаем, где это может оказаться полезным. В аудиодрайвере QNX4 вы могли бы использовать нестандартные сообщения для передачи аудиоданных администратору и от него. При ближайшем рассмотрении здесь, для задачи блочной передачи данных, вероятно, наиболее бы подошли функции *read()* и *write()*. Установку частоты

оцифровки, с другой стороны, можно было бы гораздо удачнее реализовать с применением функции *devctl()*.

Хорошо, но ведь не каждое взаимодействие клиент/сервер сводится к блочной передаче данных (тот же сервер CLID — тому пример).

### Трансляция сообщений в вызовы *devctl()* или *\_IO\_MSG*

Итак, возник вопрос — как выполнять операции управления? Самый простой способ состоит в применении POSIX-вызова *devctl()*. Наш пример из библиотеки CLID примет вид:

```
/*
 * CLID_AddSingleNPANXX(npa, nxx)
 */
int CLID_AddSingleNPANXX(int npa, int nxx) {
    struct clid_addnpanxx_t msg;
    checkAttach(); // Оставить или нет — дело вкуса
    msg.npa = npa;
    msg.nxx = nxx;
    return
        (devctl(clid_pid, DCMD_CLID_ADD_NPANXX,
                &msg, sizeof(msg), NULL));
}
```

Как видите, операция относительно безболезненная. (Для тех, кто не любит *devctl()* за то, что приходится отправлять в обе стороны блоки данных одного и того же размера, см. ниже обсуждение сообщений *\_IO\_MSG*.) Опять же, если вы пишете программу, которая должна работать в обеих операционных системах, вам следует выделить функцию обмена сообщениями в отдельный библиотечный модуль и предоставить несколько вариантов реализации, в зависимости от применяемой операционной системы.

Реально мы убили двух зайцев:

1. отказались от глобальной переменной и стали собирать сообщения на основе стековой переменной — это делает нашу программу безопасной в многопоточной среде (thread- safe);
2. передали структуру данных нужного размера вместо структуры данных максимального размера, как мы это делали в предыдущем примере с QNX4.

Заметьте, что нам пришлось определить константу *DCMD\_CLID\_ADD\_NPANXX* — в принципе, мы могли бы для этих же



целей применить константу `CLID_MsgAddSingleNPANXX` (сделав соответствующее изменение в заголовочном файле), но я просто хотел подчеркнуть тот факт, что эти две константы не являются одинаковыми.

Второй убитый заяц заключался в том, что мы передали «структуру данных нужного размера». На самом деле, мы тут немножко приврали. Обратите внимание на то, что функция `devctl()` имеет только один параметр размера (четвертый, который мы установили в `sizeof(msg)`). Как *на самом деле* происходит пересылка данных? Второй параметр функции `devctl()` содержит команду для устройства (поэтому и «DCMD»). Двумя старшими битами команды кодируется направление, которое может быть одним из четырех:

1. «00» — передачи данных нет;
2. «01» — передача от драйвера клиенту;
3. «10» — передача от клиента драйверу;
4. «11» — двунаправленная передача.

Если вы не передаете данные (то есть достаточно просто команды) или передаете их в одном направлении, то применение функции `devctl()` — прекрасный выбор. Интересен тот вариант, когда вы передаете данные в обоих направлениях. Интересен он тем, что, поскольку у функции `devctl()` только один параметр размера, обе пересылки данных (как драйверу, так и от драйвера) передадут весь буфер данных целиком! Это хорошо в том частном случае, когда размеры буферов «ввода» и «вывода» одинаковы, но представьте себе, что буфер принимаемых драйвером данных имеет размер в несколько байт, а буфер передаваемых данных гораздо больше. Поскольку у нас есть только один параметр размера, мы вынуждены будем каждый раз передавать драйверу полный буфер данных, хотя требовалось передать всего несколько байт!

Эта проблема может быть решена применением «своих собственных» сообщений на основе общего механизма управляющих последовательностей, поддерживаемого в сообщениях типа `_IO_MSG`.

Сообщение типа `_IO_MSG` было предусмотрено для того, чтобы дать вам возможность вводить свои собственные типы сообщений, не конфликтуя при этом со «стандартными» типами сообщений администраторов ресурсов, поскольку для администраторов ресурсов сам тип сообщения `_IO_MSG` уже является «стандартным».

Первое, что вы должны сделать при использовании сообщений типа `_IO_MSG` — это определить ваши «специальные» сообщения. В этом примере мы определим два таких типа и последуем стандартной модели сообщений администратора ресурсов: один тип будет сообщением ввода, другой — вывода.

```

typedef struct {
    int data_rate;
    int more_stuff;
} my_input_xyz_t;

typedef struct {
    int old_data_rate;
    int new_data_rate;
    int more_stuff;
} my_output_xyz_t;

typedef union {
    my_input_xyz_t i;
    my_output_xyz_t o;
} my_message_xyz_t;

```

Здесь мы определили новый тип — объединение (union) из сообщений ввода и вывода — и назвали этот тип **my\_message\_xyz\_t**. Закономерность в имени идентификатора заключается в том, что это сообщение относится к службе «xyz» какова бы она ни была. Сообщение ввода имеет тип **my\_input\_xyz\_t**, а сообщение вывода — **my\_output\_xyz\_t**. Отметьте, что и «ввод», и «вывод» определяются с позиции администратора ресурса: «ввод» — это данные, поступающие в администратор ресурса, а «вывод» — это данные, поступающие из него (обратно клиенту).

Нам надо придумать какой-то вызов API для клиента — мы, конечно, можем принудить клиента «вручную» заполнять структуры **my\_input\_xyz\_t** и **my\_output\_xyz\_t**, но я не рекомендовал бы так делать по той причине, что API призван «отвязать» реализацию передаваемого сообщения от функциональности. Давайте предположим, что API клиента у нас такой:

```

int adjust_xyz(int *data_rate, int *odata_rate,
    int *more_stuff);

```

Теперь мы имеем хорошо документированную функцию *adjust\_xyz()*, которая выполняет нечто полезное для клиента. Заметьте, что для передачи данных мы использовали указатели на целые числа — это просто пример реализации. Вот текст функции *adjust\_xyz()*:

```

int adjust_xyz(int *dr, int *odr, int *ms) {
    my_message_xyz_t msg;
    int sts;
    msg.i.data_rate = *dr;
    msg.i.more_stuff = *ms;
    sts =

```

```

    io_msg(global_fd, COMMAND_XYZ, &msg, sizeof(msg.i),
           sizeof(msg.o));
    if (sts == EOK) {
        *odr = msg.o.old_data_rate;
        *ms = msg.o.more_stuff;
    }
    return (sts);
}

```

Это пример применения функции `io_msg()` (ее мы скоро опишем — это не стандартный библиотечный вызов!). Функция `io_msg()` колдует над сборкой сообщения `_IO_MSG`. Чтобы уйти от проблемы функции `devctl()` с наличием только одного параметра размера, мы дали функции `io_msg()` два таких параметра: один — для ввода (`sizeof(msg.i)`), другой — для вывода (`sizeof(msg.o)`). Заметьте, что мы обновляем значения `*odr` и `*ms` *только в том случае*, когда функция `io_msg()` возвращает EOK. Это обычный прием, и здесь он полезен потому, что передаваемые аргументы не изменятся, если команда не завершится успешно. (Это предохраняет клиентскую программу от необходимости держать копию передаваемых данных на случай несрабатывания функции.)

Последнее, что я сделал в функции `adjust_xyz()`, — это зависимость от переменной `global_fd`, содержащей дескриптор файла администратора ресурса. Есть, опять же, множество способов обработки этого:

- Скрыть дескриптор файла внутри функции `io_msg()` (это было бы полезно, если бы вы пожелали избавиться от необходимости передавать дескриптор файла с каждым вызовом; это хорошо в случаях, когда вы собираетесь обмениваться сообщениями только с одним администратором ресурса, и не подходит в качестве универсального решения).

- Передавать от клиента дескриптор файла каждому вызову функции библиотеки API (полезно, если клиент хочет разговаривать с администратором ресурса еще и другими способами, например, стандартными POSIX-вызовами на основе файловых дескрипторов типа `read()`, или если клиент должен уметь общаться с несколькими администраторами ресурсов).

Вот текст функции `io_msg()`:

```

int io_msg(int fd, int cmd, void *msg, int isize,
           int osize) {
    io_msg_t io_message;
    iov_t rx_iov[2];
    iov_t tx_iov[2];
    int sts;

```

```

// set up the transmit IOV
SETIOV(tx_iov + 0, &io_msg.o, sizeof(io_msg.o));
SETIOV(tx_iov + 1, msg, osize);
// set up the receive IOV
SETIOV(rx_iov + 0, &io_msg.i, sizeof(io_msg.i));
SETIOV(rx_iov + 1, msg, isize);
// set up the _IO_MSG itself
memset(&io_message, 0, sizeof(io_message));
io_message.type = _IO_MSG;
io_message.mgrid = cmd;
return (MsgSendv(fd, tx_iov, 2, rx_iov, 2));
}

```

Отметьте несколько вещей.

В функции *io\_msg()* для «инкапсуляции» специального сообщения (передаваемого в параметре «*msg*») в структуру *io\_message* использован двухкомпонентный вектор ввода-вывода IOV.

Структура *io\_message* была предварительно обнулена, и в ней был задан тип сообщения (*\_IO\_MSG*), а также инициализировано поле *cmd* (это будет использовано администратором ресурса для определения типа посылаемого сообщения).

В качестве кода завершения функции *io\_msg()* использовался непосредственно код завершения *MsgSendv()*.

Единственная «забавная» вещь, которую мы тут сделали, касается поля *mgrid*. QSSL резервирует для данного поля диапазон значений со специальным поддиапазоном для «неофициальных» драйверов. Этот поддиапазон ограничен значениями от *\_IOMGR\_PRIVATE\_BASE* до *IOMGR\_PRIVATE\_MAX* соответственно. Если вы разрабатываете глубоко встраиваемую систему и хотите быть уверены, что ваш администратор ресурса не получит никаких неподходящих сообщений, то смело можете использовать значения из этого специального диапазона. С другой стороны, если вы разрабатываете в большей степени «настольную» или «обычную» систему, вы можете захотеть точно проконтролировать, будут ли вашему администратору ресурса приходят несоответствующие сообщения или нет. В этом случае вам нужно будет обратиться в QSSL за значением *mgrid*, зарезервированным специально для вас — никто, кроме вас, не должен использовать это номер. Посмотрите файл `<sys/iomgr.h>`, там представлены используемые в настоящее время диапазоны. В нашем вышепредставленном мы могли предположить, что *COMMAND\_XYZ* базирована на *\_IOMGR\_PRIVATE\_BASE*:

```
#define COMMAND_XYZ (_IOMGR_PRIVATE_BASE + 0x0007)
```

или QSSL назначила нам специальный поддиапазон:

```
#define COMMAND_XYZ ( IOMGR_ACME_CORP + 0x0007)
```

### «Клиент/сервер» с использованием администратора ввода/вывода

А что если клиент, которого вы переносите, использует администратор ввода/вывода? Как адаптировать его для QNX/ Neutrino? Ответ прост: мы уже это сделали. Установив интерфейс на основе файловых дескрипторов, мы уже используем администратор ресурса. В QNX/Neutrino вам почти *никогда* не придется использовать интерфейс «сырых» сообщений. Почему?

1. Вам пришлось бы самим беспокоиться о сообщении `_IO_CONNECT`, поступившим с клиентским вызовом `open()`, или искать способ поиска администратор ресурса, альтернативный использованию `open()`.

2. Вам пришлось бы искать способ сопоставить клиенту конкретный контекстный блок в администраторе ресурса. Это, конечно, не ракетная техника, но поработать придется.

3. Вам придется инкапсулировать все ваши сообщения вручную вместо использования стандартных POSIX-функций, которые бы сделали эту работу за вас.

4. Ваш администратор ресурса не будет работать с приложениями на основе `stdin/stdout`. В примере с аудиодрайвером вы не смогли бы просто так выполнить `mp3_decode spud.mp3 >/dev/audio`, потому что `open()`, скорее всего, не сработала бы (а если бы и сработала, то не сработала бы `write()`, и так далее).

### Прокси

В QNX4 единственным способом передачи неблокирующего сообщения было создание прокси — это делалось с помощью функции `qnx_proxy_attach()`. Эта функция возвращает идентификатор прокси (проху ID), (он выбирается из того же самого пространства номеров, что и идентификаторы процессов), к которому вы затем можете применить функцию `Trigger()` или вернуть его из функции обработки прерывания (см. ниже).

В QNX/Neutrino вы бы вместо этого настроили структуру `struct sigevent` на генерацию «импульса», а потом либо использовали бы

функцию *MsgDeliverEvent()* для доставки события, либо привязали бы событие к таймеру или обработчику прерывания.

Обычный прием распознавания прокси-сообщений QNX4 (полученных с помощью *Receive()* или *Creceive()*) — сравнить идентификатор процесса, возвращенный функцией приема сообщения, с ожидаемым идентификатором прокси. Если совпадают — значит, это прокси. Как вариант, идентификатор процесса можно было игнорировать и обрабатывать сообщение как «стандартное». К сожалению, это несколько усложняет проблему переноса программ.

### *Анализ прокси по идентификаторам*

Если вы сравниваете полученный от функции приема идентификатор процесса со списком ожидаемых идентификаторов прокси, обычно вы игнорируете содержимое прокси. В конце концов, коль скоро содержимое прокси нельзя изменить после ее создания, какой прок с анализа сообщения, о котором вы уже знаете, что это одна из ваших прокси? Вы можете возразить, что это для удобства — помещаем в прокси нужные сообщения, а затем обрабатываем все сообщения одним стандартным декодером. Если это ваш случай, см. ниже «Анализ прокси по содержимому».

Поэтому, в QNX4 ваш код выглядел бы примерно так:

```
pid = Receive(0, &msg, sizeof(msg));
if (pid == proxyPidTimer) {
    // Сработал наш таймер, сделать что-нибудь
} else if (pid == proxyPidISR) {
    // Сработал наш ISR, сделать что-нибудь
} else {
    // Не наша прокси — возможно, обычное
    // клиентское сообщение. Сделай что-нибудь.
}
```

В QNX/Neutrino он заменился бы на следующий:

```
rcvid = MsgReceive(chid, &msg, sizeof(msg), NULL);
if (rcvid == 0) { // 0 значит, что это импульс
    switch (msg.pulse.code) {
        case MyCodeTimer:
            // Сработал наш таймер, сделать что-нибудь
            break;
        case MyCodeISR:
```

```

    // Сработал наш ISR, сделать что-нибудь
    break;
default:
    // Неизвестный код импульса
    break;
}
} else {
    // rcvid - не ноль, значит, это обычное
    // клиентское сообщение. Сделать что-нибудь.
}

```

Отметим, что это пример для случая, когда вы обрабатываете сообщения самостоятельно. Но поскольку мы рекомендуем использовать библиотеку администратора ресурсов, на самом деле ваша программа выглядела бы примерно так:

```

int main(int argc, char **argv) {
    ...
    // Выполнить обычные инициализации
    pulse_attach(dpp, 0, MyCodeTimer, my_timer_pulse_handler,
        NULL);
    pulse_attach(dpp, 0, MyCodeISR, my_isr_pulse_handler,
        NULL);
    ...
}

```

На этот раз мы предписываем библиотеке администратора ресурсов ввести две проверки из предыдущего примера в основной цикл приема сообщений и вызывать две наши функции обработки (*my\_timer\_pulse\_handler()* и *my\_isr\_pulse\_handler()*) всякий раз, когда обнаруживаются нужные коды. Гораздо проще.

### ***Анализ прокси по содержимому***

Если вы анализируете содержимое прокси (фактически игнорируя, что это прокси, и обрабатывая их как сообщения), то вы автоматически имеете дело с тем, что в QNX4 на прокси ответить нельзя. В QNX/Neutrino ответить на импульс тоже нельзя. Это означает, что у вас уже есть код, который либо анализирует идентификатор, возвращаемый функцией приема, и определяет, что это прокси, и отвечать не надо, либо смотрит на содержимое сообщения и по нему определяет, надо отвечать на это сообщение или нет.

К сожалению в QNX/Neutrino произвольные данные в импульс не запишешь. Импульс имеет четко определенную структуру, и обойти это нельзя. Умным решением здесь было бы «имитировать» сообщение от прокси при помощи импульса и таблицы. Таблица содержала бы сообщения, которые раньше передавались посредством прокси. Получив импульс, вы использовали бы поле *value* в качестве индекса к этой таблице, выбрали бы из таблицы соответствующее сообщение и «притворились», что получено именно оно.

## Обработчики прерываний

Обработчики прерываний в QNX4 могли либо вернуть идентификатор прокси (указывая этим, что надо переключить прокси и таким образом уведомить ее владельца о прерывании), либо вернуть нуль (что означало бы, что в дальнейшем ничего делать не требуется). В QNX/Neutrino механизм почти идентичен — за исключением того, что вместо возвращения идентификатора прокси вы возвращаете указатель на **struct sigevent**. Генерируемое событие может быть либо импульсом (ближайший аналог прокси), либо сигналом, либо созданием потока — как выберете, так и будет.

Также в QNX4 вы *обязаны были* иметь обработчик прерывания — даже в том случае, если он должен был только вернуть идентификатор прокси. В QNX/Neutrino вы можете привязать **struct sigevent** к вектору прерывания, используя *InterruptAttachEvent()*, и это событие будет генерироваться при каждой активизации данного вектора.



## Резюме

Перенос приложений из QNX4 в QNX/Neutrino или поддержка программы, функционирующей в обеих операционных системах, — это возможно, если придерживаться следующих правил:

- абстрагировать, абстрагировать и еще раз абстрагировать;
- развязывать, развязывать и еще раз развязывать.

Ключ в том, чтобы отказаться от использования конкретного «дескриптора», который «связывал» бы клиента с сервером, и не привязываться к конкретному механизму обнаружения сервера. Если вы абстрагируете операции обнаружения сервера и установления соединения в отдельный набор функций, то вы сможете производить условную компиляцию для любой платформы, на которую вы пожелаете перенести ваш код.

Аналогичные рассуждения применимы и к транспортировке сообщений — всегда абстрагируйте клиентский API от «понимания» того, как сообщение доставляется от клиента к серверу, в некоторый универсальный API, который уже бы базировался на конкретном транспортном API. Затем этот конкретный транспортный API можно будет условно откомпилировать для любой платформы.

Перенос *сервера* из QNX4 в QNX/Neutrino является более трудной задачей вследствие того, что QNX4-серверы были сделаны «вручную» и никогда не следовали четкой структуре подобно тому, как это принято в библиотеке администраторов ресурсов в QNX/Neutrino. Впрочем, в общем случае, если вы переносите что-то сильно аппаратно-зависимое (например, аудиодрайвер или блок-ориентированный дисковый драйвер), основная часть портируемого кода никак не связана с операционной системой, и ой как связана с собственно аппаратурой. Подход, который я обычно использовал для таких случаев, состоит в том, чтобы запрограммировать каркас «драйвера» и обеспечить набор четко определенных аппаратно-зависимых функций. Каркас будет отличаться для разных операционных систем, но аппаратно-зависимые функции получатся на удивление переносимыми.

# **Приложение Б**

## **Скорая помощь**

## Обращайтесь за помощью к профессионалам

Независимо от того, насколько вы хороший программист, иногда возникают моменты, когда вы:

- сталкиваетесь с проблемой, которую не можете решить;
- выявили ошибку и хотите сообщить о ней и/или найти обходной путь;
- сталкиваетесь с необходимостью консультации по реализации проекта.

В этой главе мы рассмотрим ресурсы, необходимые при разрешении подобных проблем.

### Итак, у вас проблема...

Мы обсудим первые две проблемы вместе, потому что подчас бывает трудно определить, с которой из них имеешь дело.

Предположим, что что-то вдруг перестало работать или работает не так, как ожидается. Что делать?

### RTFM

Читайте документацию! (Оригинальный перевод аббревиатуры RTFM широко известен, поэтому приводить его здесь не будем; любопытные могут затянуть в словарь — например, <http://www.instantweb.com/foldoc/> — прим. ред.) Несмотря на то, что это может показаться элементарным и очевидным первым шагом, число людей, которые этого не делают, просто поразительно!

Все нижеперечисленные справочные руководства по QNX/Neutrino доступны он-лайн:

- «*Building Embedded Systems*» («Построение встраиваемых систем»);
- «*Development Tools*» («Средства разработки»);
- «*Library Reference*» («Справочник по библиотекам»);
- «*System Architecture*» («Системная архитектура»);
- «*Technotes*» («Технические замечания»);
- «*Utilities Reference*» («Справочник по утилитам»).

**«*Building Embedded Systems*» («Построение встраиваемых систем»)**

В этой книге содержится вся информация, необходимая для встраивания QNX/Neutrino — то есть чтобы создать систему на основе QNX/Neutrino и заставить ее работать. В ней есть главы по среде разработки программных продуктов (как компилировать, компоновать и отлаживать программы для QNX/Neutrino), построению образа ОС (как создать образ, как встроить этот образ глубоко встраиваемую систему, как заставить его работать на нужной платформе), а также кое-какие общие рекомендации по проектированию.

### ***«Development Tools» («Средства разработки»)***

Это справочник по различным средствам разработки, о которых упоминается в книге *«Построение встраиваемых систем»* — там есть документация по GNU-компилятору и сопутствующим инструментальным средствам, описания средств формирования образов, и т.д. Это прекрасный источник информации для тех случаев, когда инструмент ведет себя не так, как вы от него ожидаете, или когда нужно понять, как работает тот или иной инструмент.

### ***«Library Reference» («Справочник по библиотекам»)***

Это руководство по вызовам Си-библиотеки QNX/Neutrino — от А до Z. Применительно к вызовам функций это — истина в высшей инстанции. Я часто ссылался на это издание в предыдущих главах (например, за более подробной информацией о конкретной функции — скажем, о ее редко используемых аргументах).

### ***«System Architecture» («Системная архитектура»)***

«Высокоуровневый» документ по системной архитектуре QNX/Neutrino, описывающий ее «сверху» и в то же время приводящий достаточно подробное описание реализации, из которого вы сможете понять, что из себя представляют компоненты системы, и как они связаны между собой.

### ***«Technotes» («Технические замечания»)***

Серия книг «*Технические замечания*» содержит описание специфических особенностей QNX/Neutrino и может меняться от версии к версии. Посмотрите онлайн-версию этих документов, чтобы узнать, что характерно для имеющейся у вас версии QNX/Neutrino. Например, для релиза QNX/Neutrino 2.00 BETA 5 в серии «Технические замечания» были рассмотрены следующие темы:

- *Переход с Neutrino 1.X на версию 2.0;*
- *Интерфейс диспетчеризации в QNX/Neutrino 2.0.*

### **«Utilities Reference» («Справочник по утилитам»)**

Этом алфавитный справочник по доступным командно-строковым утилитам. Он описывает все утилиты, не относящиеся к средствам разработки, то есть те, которые не охвачены справочником «*Средства разработки*» — **grep**, **cat**, **echo**, и т.д.

### **Свяжитесь с технической поддержкой**

Когда вы определитесь, что дело не в вашем недопонимании вызова функции или утилиты и не в опечатке, добро пожаловать за помощью в королевство технической поддержки QSSL. Бояться нечего — большинство заказчиков очень довольны уровнем технической поддержки, которую они получают от QSSL.

Обращаться за помощью в группу технической поддержки QSSL можно двояко: либо по телефону, либо через телеконференции QUICS (еще раз отмечу: эта информация устарела, на настоящий момент QUICS упразднена в пользу QDN (<http://qdn.qnx.com>, <nntp://inn.qnx.com>) — прим. ред.).

Прежде чем мы решим, каким способом лучше воспользоваться, хотелось бы упомянуть ряд подготовительных действий, выполнив которые, вы сможете свести время получения ответа на ваш вопрос к минимуму.

### **Опишите проблему**

Как правило, заказчики пытаются решить проблему самостоятельно, пробуя различные варианты, которые им приходят на ум. Это великолепно.

Только, к сожалению, это часто приводит к тому, что заказчик приходит в уныние и вывешивает в телеконференцию сообщение типа:

«Я тут запустил TCP/IP, пытаюсь соединиться с Windows-машиной, а оно не работает.

В чем дело?!?»

Первый ответ службы технической поддержки будет выглядеть примерно так (бьюсь об заклад, у них даже есть специальный шаблон для этого):

Что значит «не работает»? Вы имеете в виду TCP/IP на стороне QNX или на стороне Windows? Какой модуль TCP/IP не работает? Что вы пытаетесь сделать? Какие у вас версии QNX и TCP/IP? Какая у вас версия Windows и какой там TCP/IP?

Мораль сей басни такова: если у вас проблема, то вам, вероятно, надо ее решить быстро. Если вам нужен ответ на вопрос быстро, постарайтесь привести максимум информации о возникшей у вас проблеме в первом же запросе, чтобы специалисты технической поддержки QSSL сразу же смогли попробовать воспроизвести вашу проблемную ситуацию.

Существует ряд вещей, которые служба технической поддержки спрашивает всегда. К ним относятся:

- точное описание сбоя;
- версии программных продуктов;
- конфигурация системы;
- аппаратная платформа (x86, PPC, и т.д.)

### ***Точная информация***

Чтобы предоставить эту информацию, опишите, что вы ожидали, и что произошло на самом деле. Вышеприведенный пример выглядел бы гораздо лучше в таком варианте:

Я пытаюсь подсоединиться по telnet из QNX/Neutrino 2.00A к Windows-машине и сразу же после приглашения login получаю сообщение «Connection closed by foreign host».

### ***Версии***

Следующее, что следует указать, — это информация о версиях различных модулей, которые вы используете. Ее можно взять от команд **ls** и **cksum**. В случае с нашим примером хорошо бы бы сообщить службе техподдержки, какая у вас версия команды **telnet**, стека TCP/IP, и т.д.

```
# cd /usr/nto/x86
# ls -l bin/telnet dll/devn-ne2000.so dll/npi-ttcpip.so
-rwxr-xr-x 1 root root 71588 May 19 11:45 bin/telnet*
-rwxr-xr-x 1 root root 680928 May 19 11:05 dll/devn-ne2000.so*
-rwxr-xr-x 1 root root 83765 Jun 02 06:39 dll/npi-ttcpip.so*
# cksum bin/telnet dll/devn-ne2000.so dll/npi-ttcpip.so
1574162245 71588 bin/telnet
3564123752 680928 dll/devn-ne2000.so
2582029317 83765 dll/npi-ttcpip.so
```

Это даст сотрудникам техотдела примерную картину дат, размеров и контрольных сумм ряда объектов, которые, возможно, связаны с проблемой. Отметьте строку «**cd /usr/nto/x86**» — она говорит, что я использую средства разработки в версии для x86.

Если у вас есть подозрение, что проблема может быть связана с особенностями платформы, вам, безусловно, следует указать ее название, торговую марку (brand) и соответствующие применяемые в ней чипсеты.

Также служба технической поддержки, как правило, требует описания конфигурации вашей системы — особенно когда есть подозрение, что проблема связана с недостатком памяти, лицензированием, и т.п. Вы должны попытаться дать полную информацию относительно объема установленной памяти, количестве выполняющихся процессов, приблизительной степени загрузки, и т.д.

Чем более подробную информацию вы предоставите, тем быстрее вам смогут помочь.

### *Если у вас бета...*

Если вы используете бета-версию программного обеспечения (то есть вы подписаны на бета-программу QSSL), вся вышеперечисленная информация критична, потому ваши версии программного обеспечения, скорее всего, будут отличаться от официально выпущенных. Однако, имейте в виду, что в общем случае поддержка бета-версий по телефону в QSSL не предусмотрена, так что здесь единственный способ получить информацию — написать в телеконференцию или, если разработчик

запрашивает прямой контакт с вами, поговорить с разработчиком напрямую.

В любом случае одним из лучших решений являются телеконференции, потому что они дают возможность и другим участникам видеть, какие есть проблемы, и как с ними быть (то есть если найдена ошибка, то какие есть обходные пути). Как бы там ни было, вышеупомянутая информация критична для определения того, какие именно программные продукты из бета-релиза вы применяете.

Также, если вы обмениваетесь информацией с разработчиком, то имейте в виду, что у него, как правило, есть еще тысяча дел, и он вряд ли сможет ответить на ваш запрос сразу. Дружеский «ping» спустя несколько дней молчания делу не повредит, а вот повторное требование ответа через 15 минут после запроса определенно не прибавит вам новых друзей.

Что часто проявляется при работе с бета-версиями, так это то, что люди забывают ставить обновления. Механизм же бета-тестирования работает так, что пропуск обновления запросто может заставить вашу систему вести себя странно. Некоторые новые драйверы или администраторы ресурсов могут вести себя со своими клиентами совершенно иначе, чем в предыдущих версиях.

В этом случае вы должны быть уверены (потому что вас обязательно спросят!), что все обновления установлены, и в нужном порядке.

### ***Воспроизведите проблему***

Один из первых вопросов персонала технической поддержки обычно звучит так: «А оно происходит исключительно с бухты-барахты, или вы можете это повторить намеренно?»

Это не праздное любопытство. Если проблема проявляется редко, она столь же серьезна, как и проблема, которая проявляется регулярно. Главное — понять, с какой стороны подойти.

Обычно в случае редко проявляющейся проблемы персонал технической поддержки рекомендует сконфигурировать операционную систему и другие компоненты так, чтобы когда проблема проявится вновь, остался какой-нибудь отчет о состоянии системы или был вызван отладчик — чтобы можно было потом разобраться, что произошло.

Если проблемная ситуация легко воспроизводится, то технический персонал захочет воспроизвести ее на месте, чтобы продемонстрировать разработчику на «живой» системе. «Смотри-ка! Все умирает, стоит мне сде...»



## Сузьте круг поиска

Даже если проблема воспроизводима, персонал технической поддержки, скорее всего, не будет в восторге от перспективы перевернуть 6000 строк вашей Си-кода в поисках затаившейся в его недрах ошибки.

В большинстве случаев, которые мне доводилось видеть, местоположение ошибки можно было сузить примерно до 20–30 строк *максимум*. Большие файлы могут быть действительно полезны для анализа в тех случаях, когда вы подозреваете, что ошибка связана с размерами объектов, а не с библиотеками или ядром. Например, у некоторых утилит может быть задан размер массива по умолчанию, и попытка увеличить его размер ведет к проблеме. В этом случае персонал технической поддержки может запросить у вас **tar**-архив, в котором содержится *все*. К счастью, создавать **tar**-архивы несложно. Например, если каталог вашего проекта называется **/src/projects/xyzyy**, и техперсонал хочет посмотреть все, что там находится, вы можете сделать следующее:

```
# cd /src/projects
# tar cvf xyzyy.tar xyzyy
```

Это «высосет» все содержимое каталога **xyzyy** (включая все подкаталоги!) в файл с именем **xyzyy.tar**. Если результирующий **tar**-файл получится большим, вы сможете сэкономить время на его загрузку и место на диске, если сожмете его с помощью утилиты **gzip**:

```
# gzip -9v xyzyy.tar
xyzyy.tar: 60.2% — replaced with xyzyy.tar.gz
```

Вы сможете затем переслать полученный файл **xyzyy.tar.gz** персоналу техподдержки (лучше, конечно, по FTP, чем электронным письмом :-).

## Как сообщить об ошибке

Если вы убеждены в том, что нашли ошибку (например, утилита «падает» по SIGSEGV), вы можете не выставлять ее на всеобщее обсуждение в телеконференцию, а напрямую сообщить об ошибке в QSSL при помощи команды **bug** на QUICS. (Этот способ также устарел. Теперь для сообщения об ошибках используется специальная веб-форма на QDN — [http://qdn.qnx.com/report/problem\\_report.html](http://qdn.qnx.com/report/problem_report.html) — прим. ред.)

## Другие источники информации

Кроме QUICS, существует ряд других информационных ресурсов, куда можно обращаться за помощью, информацией или услугами. Ниже приведен далеко не полный список, но по крайней мере у вас будет с чего начать.

`www.qnx.com`

Веб-сайт компании QSSL, особенно его раздел техподдержки (`qdn.qnx.com` — *прим. ред.*), наполнен полезной информацией и файлами.

`comp.os.qnx`

Это обычная телеконференция USENET. Она была создана для пользователей QNX, чтобы они могли вместе обсуждать проблемы, особенности, решения, и т.п.

Персонал QSSL не всегда следит за этой конференцией (хотя некоторые там все-таки прячутся). Вообще это конференция для тех, у кого есть вопросы, но нет продуктов QNX. В эту конференцию периодически вывешивается сборник ЧаВо (Часто задаваемые Вопросы, или Frequently Asked Questions — FAQ — *прим. ред.*), из которого можно узнать много полезного о продуктах.

Если вам нужен *гарантированный* ответ, то лучшим выбором была и остается QUICS.

## Каталог «третьих» фирм — продукты и консалтинг

Компания QSSL издает Каталог «третьих» фирм (Third Party Directory), в котором перечислены все сторонние компании и их продукты для QNX (например, многопортовые платы последовательного интерфейса, аппаратные средства стандарта X.25, программное обеспечение), а также представлен список организаций и лиц, оказывающих консалтинговые услуги при разработке проектов.

Данный каталог доступен от QSSL, для получения копии обращайтесь к своему торговому представителю.

## *QNXnews*

Компания QSSL издает ежеквартальный журнал, называемый QNXnews. Он содержит статьи о заказчиках компании QSSL, примеры

применения ими QNX в своих продуктах и решениях, а также объявления от QSSL и «третьих» фирм о новых продуктах и услугах.

Подписка на этот журнал также доступна в QSSL. Обращайтесь к вашему торговому представителю для получения дополнительной информации

### *Обучение*

И, наконец, некоторые компании предлагают курсы обучения по программным продуктам QNX — за подробностями обращайтесь в отдел обучения QSSL

## Глоссарий

### *Neutrino*

Цитата из веб-странички сайта Обсерватории по изучению нейтрино, расположенной в Садбери (Sudbury Neutrino Observatory) (см. <http://www.sno.phy.queensu.ca>):

«Нейтрино — это мельчайшие, возможно невесомые, нейтральные элементарные частицы, которые взаимодействуют с веществом через слабые ядерные силы взаимодействия. Незначительность этих сил взаимодействия наделяет нейтрино свойством свободного прохождения через вещество — говорят, что материя является почти прозрачной для нейтрино. Солнце и все другие звезды в результате процессов синтеза и распада в пределах ядра генерируют мощные потоки нейтрино. Поскольку нейтрино взаимодействуют редко, они проникают через Солнце и Землю (и вас) беспрепятственно. Другими источниками нейтрино являются взрывающиеся звезды (сверхновые), реликтовые нейтрино (от рождения вселенной) и атомные электростанции (при работе которых много энергии топлива уносится потоком нейтрино). Например, Солнце генерирует более чем две сотни триллионов триллионов триллионов нейтрино каждую секунду, а взрыв сверхновой может выдать поток нейтрино в тысячи раз больше, чем наше Солнце, которое произведет его в течение всех 10 миллиардов лет своей жизни. Поток в миллиарды нейтрино каждую секунду пронизывает Ваше тело и всего только одна или две частицы из частиц с наибольшей энергией рассеиваются в Вас за всю Вашу жизнь.»

### *PDP-8*

Старинный компьютер, создан в период между 1965 и 1970 гг. компанией Digital Equipment Corporation (DEC; в настоящее время — Compaq) и имел самую продвинутую (в оригинале было «coolest» — *прим. ред.*) по тем временам переднюю панель. Также это был первый компьютер, который мне довелось программировать. Если у вас есть такой (особенно PDP-8/1) или его фрагменты, инструкции и т.д., известите меня об этом по электронной почте (адрес: [rk@parse.com](mailto:rk@parse.com)) — я их коллекционирую! К

сожалению, эта замечательная 12-разрядная машина не работает под Neutrino. :(

### ***pthread***

Собирательное имя для набора вызовов функций *pthread\_\**(). Подавляющее большинство этих вызовов определены стандартом POSIX (Portable Operating System Interface for Unix) и используется при работе с потоками.

### ***QNX Software Systems Limited***

Компания-разработчик операционных систем QNX2, QNX4 и QNX/Neutrino (QNX6).

### ***QSSL***

Аббревиатура от QNX Software Systems Limited.

### ***абсолютный таймер (absolute timer)***

Таймер, истекающий в фиксированный момент времени, например, 20 января 2005 года в 9 часов 42 минуты утра по восточному поясному времени (EDT — Eastern Daylight Time). Сравните с определением «относительный таймер».

### ***администратор ресурса (resource manager)***

Сокращенно также «resmgr». Это серверный процесс, предоставляющий произвольному клиенту ряд строго определенных сервисов на основе файловых дескрипторов. Администратор ресурса поддерживает некоторый ограниченный набор сообщений, которые соответствуют стандартным клиентским библиотечным функциям Си, таким как *open()*, *write()*, *lseek()*, *devctl()* и т.д.

### ***асинхронный (asynchronous)***

Понятие, используемое для того, чтобы указать на то, что данная операция не синхронизирована по отношению к другой операции. Например, временные метки, которые генерируются микросхемой системного таймера, как говорят, «асинхронны» в отношении потока, запрашивающего задержку на некоторое время, поскольку запрос на задержку никак не синхронизирован с возникающим прерыванием отсчета таймера. Сравните с определением «синхронный».

### ***атомарная операция (atomic operation)***

Операция, которая является «неделимой», то есть такая, которая не может быть прервана любой другой операцией. Атомарные операции критично важны, особенно в подпрограммах обработки прерываний и многопоточных программах, поскольку последовательность событий типа «проверить и установить», которая осуществляется в одном потоке, должна быть гарантирована от прерывания другим потоком. Любую последовательность можно сделать атомарной с точки зрения защищённости от вмешательства других потоков, применяя мутексы или — в обработчиках прерываний — функции *InterruptLock()* и *InterruptUnlock()*. См. также заголовочный файл `<atomic.h>`.

### ***атрибутная запись (attribute structure)***

Структура, используемая в пределах администратора ресурса и содержащая информацию, относящуюся к устройству, которое администратор ресурса декларирует в пространстве имен путей. Если администратор ресурса декларирует несколько устройств (например, администратор последовательного порта может объявить `/dev/ser1` и `/dev/ser2`), он будет поддерживать соответствующее число атрибутивных записей. Сравните с определением ОСВ.

### ***барьер (barrier)***

Объект синхронизации на уровне потоков, которому соответствует некое значение счетчика. Потоки, запрашивающие блокировку по барьеру (функция *barrier\_wait()*), блокируются до тех пор, пока число потоков, запросивших блокировку, не станет равным указанному значению; как только это произойдет, все эти потоки будут разблокированы. Сопоставьте это с работой семафора.

***блок управления открытым контекстом (open context block, сокр. ОСВ)***

Структура данных, используемая администратором ресурсов и содержащая информацию по каждому клиентскому запросу типа *open()*. Если клиент открыл несколько файлов, то для каждого дескриптора файла, который этот клиент имеет у соответствующих администраторов ресурсов, будет существовать соответствующий блок управления открытым контекстом (ОСВ). Сравните с атрибутивной записью.

***блокирование (blocking)***

Средство синхронизации потоков по отношению к другим потокам или событиям. В заблокированном состоянии (которых имеется порядка дюжины) поток не расходует процессорное время — он находится в ожидании в списке, поддерживаемом в пределах ядра. Когда происходит ожидаемое потоком событие, поток разблокируется и снова становится способным использовать процессор.

***вектор ввода/вывода (I/O Vector, сокр. IOV)***

Структура, в которой каждый элемент содержит указатель и длину. Обычно применяются не одиночные векторы ввода/вывода, а массивы векторов — такой массив структур из указателей и длин определяет список фрагментов сообщения для операции фрагментации/дефрагментации (*scatter/gather*), позволяющей выполнять обмен сообщениями намного эффективнее (в противном случае, чтобы сформировать один непрерывный буфер, данные пришлось бы копировать по отдельности).

***взаимная блокировка (deadlock)***

Аварийная ситуация, которая возникает, когда два потока взаимно заблокированы, ожидая друг от друга ответ. Это состояние можно легко воспроизвести: просто заставьте два потока отправить друг другу сообщение — с этого момента оба потока перейдут в состояние ожидания ответа. Поскольку оба потока заблокированы, они не имеют возможности ответить, следовательно, наблюдается тупиковая ситуация. Для исключения взаимной блокировки клиенты и серверы должны придерживаться иерархического принципа обмена.

### ***виртуальная память (virtual memory)***

Система виртуальной памяти — это система, в которой виртуальное адресное пространство может, но не обязательно, быть преобразовано на основе взаимно-однозначного соответствия с физическим адресным пространством. Типовым примером этого (на момент написания книги в QNX/Neutrino это не поддерживается) является система «со страничной организацией памяти», где в случае недостатка ОЗУ некоторые фрагменты адресного пространства процессов могут быть выгружены на диск. Что QNX/Neutrino действительно поддерживает, так это динамическое распределение стековых страниц.

### ***виртуальный адрес (virtual address)***

Адрес, которому не обязательно соответствует физический адрес. В QNX/Neutrino все потоки работают в режиме виртуальной адресации, когда виртуальные адреса транслируются в физические при помощи диспетчера памяти. Сравните с понятиями «физический адрес» и «виртуальная память».

### ***выравнивание (alignment)***

Выравнивание — характеристика операции, при которой доступ к N-байтовому элементу данных должен выполняться только по адресу, кратному N. Например, чтобы обратиться к 4-байтовому целому числу, адрес этого целого числа должен быть кратным 4 байтам (например, 0x2304B008, а не 0x2304B009). В архитектуре некоторых процессоров (CPU) при попытке невыровненного доступа генерируется ошибка



выравнивания (alignment fault). В архитектуре других процессоров (например, x86) невыровненный доступ осуществляется медленнее, чем выровненный доступ.

#### ***диспетчер памяти (Memory Management Unit, сокр. MMU)***

Аппаратный блок (обычно интегрированный с центральным процессором), который обеспечивает трансляцию виртуальных адресов в физические и может использоваться для реализации системы виртуальной памяти. В QNX/Neutrino главным преимуществом применения диспетчера памяти является возможность обнаружить момент, когда поток обращается к виртуальному адресу, который не отображен в адресное пространство соответствующего процесса.

#### ***диспетчеризация FIFO (FIFO scheduling)***

При диспетчеризации FIFO (First In — First Out) поток будет использовать процессор до тех пор, пока поток с более высоким приоритетом не перейдет в состояние готовности, или пока поток добровольно не освободит процессор. Если не существует потоков с более высокими приоритетами, и поток добровольно не освобождает процессор, он будет выполняться вечно. Сопоставьте с карусельной диспетчеризацией.

#### ***идентификатор отправителя (receive ID)***

Когда сервер принимает сообщение от клиента, функции сервера *MsgReceive()* или *MsgReceivev()* возвращают идентификатор отправителя (часто сокращенно в программах называемый «*rcvid*»). Этот идентификатор *rcvid* затем используется по отношению к заблокированному клиенту как дескриптор, позволяя серверу отправить клиенту ответ с данными, там самым разблокировав его. После использования *rcvid* для ответа клиенту, он перестает иметь значение для всех вызовов функций, кроме функции *MsgDeliverEvent()*.

#### ***идентификатор соединения (connection ID, сокр. CID)***

Дескриптор, возвращаемый функцией *ConnectAttach()* (на клиентской стороне) и используемый для всех операций обмена данными между клиентом и сервером. Идентификатор соединения аналогичен дескриптору файла в терминах стандартной библиотеки языка Си — иными словами, когда функция *open()* в QNX/Neutrino возвращает дескриптор файла, реально возвращается идентификатор соединения.

### ***иерархический принцип обмена (send hierarchy)***

Принятая в QNX/Neutrino концепция, в силу которой отправляемые сообщения передаются в одном направлении, а ответы на сообщения — в другом. Основной целью реализации иерархического принципа обмена является необходимость исключения состояния взаимной блокировки потоков. Иерархический принцип реализуется назначением клиентам и серверам «уровней иерархии» и обеспечения того, чтобы сообщения передавались только на более высокий уровень иерархии. Это исключает ситуации взаимной блокировки, когда два потока посылают сообщения друг другу, потому что такая ситуация нарушила бы принцип — поток не должен отправлять сообщения другому, если тот находится на нижнем уровне иерархии.

### ***импульс (pulse)***

Неблокирующее сообщение, получаемое аналогично обычному сообщению. Это сообщение является неблокирующим для отправителя; получатель же может ожидать его применением стандартных функций обмена сообщениями (*MsgReceive()* и *MsgReceivev()*) или же, если необходимо ждать именно импульса, то при помощи функции *MsgReceivePulse()*. В то время как большинство сообщений обычно посылаются от клиента к серверу, импульсы обычно пересылаются в противоположном направлении, чтобы не нарушать иерархический принцип обмена (это вызвало бы взаимную блокировку). Сравните с сигналом.

### ***исходящий вызов ядра (kernel callout)***

Операционная система QNX/Neutrino может быть настроена для функционирования на различных аппаратных средствах без необходимости в лицензии на исходный код. Для этого необходимо предусмотреть в начальном загрузчике возможность обработки исходящих вызовов ядра. Механизм исходящих вызовов ядра позволяет разработчику добавлять в систему свой код, «знающий» о специфике оборудования — например, как опрашивать контроллер прерывания о том, какое прерывание произошло, или о том, как настроить таймер на регулярную генерацию прерываний, и т.п. Это очень подробно изложено в книге «Building Embedded Systems» («Построение встраиваемых систем»).

### *канал (channel)*

Абстрактный объект, через который сервер принимает сообщения. Это тот же самый объект, к которому клиент подключается, чтобы отправить сообщение серверу. При создании канала с помощью *ChannelCreate()* возвращается идентификатор канала («channel ID», сокращенно «chid») — тот самый идентификатор канала, который назначается администратором ресурса каждой объявляемой им точке монтирования.

### *карусельная диспетчеризация (round robin (RR) scheduling)*

При карусельной диспетчеризации поток использует процессор до тех пор, пока либо не будет готов к работе поток с более высоким приоритетом, либо пока этот поток добровольно не освободит процессор, либо пока не истечет выделенный данному потоку квант времени. Если потоков с более высоким приоритетом нет, поток добровольно не освобождает процессор, и не существует других потоков с тем же самым приоритетом, поток будет выполняться вечно. Если удовлетворены все вышеизложенные условия, за исключением того, что становится готов к работе другой поток с таким же самым приоритетом, то предыдущий поток освободит процессор после того, как истечет выделенный ему квант времени — таким образом, другой поток будет иметь шанс на обслуживание. Сравните с диспетчеризацией FIFO.

### *клиент (client)*

Архитектура обмена сообщениями в QNX/Neutrino имеет клиент-серверную структуру. Клиент является тем, кто запрашивает услугу у определенного сервера. Обычно клиент запрашивает услуги, используя функции, ориентированные на работу со стандартными файловыми дескрипторами (например, *lseek()*). Эти функции являются синхронными в том отношении, что вызов, сделанный клиентом, не возвращает ответ до тех пор, пока не будет завершена обработка запроса сервером. Любой поток может являться одновременно и клиентом, и сервером.

***мутекс (mutex, от mutual exclusion — «взаимное исключение»)***

Объект, применяемый для упорядочения последовательности доступа потоков к ресурсу — так, чтобы к ресурсу, определяемому мутексом, в конкретный момент времени имел доступ только один поток. Например, используя мутекс всякий раз при обращении к некоторой переменной, вы гарантируете, что только один поток в данный момент времени имеет к ней доступ, тем самым предотвращая гонки. См. также «атомарная операция».

***обмен сообщениями (message-passing)***

Операционная система QNX/Neutrino имеет в своей основе модель обмена сообщениями, в которой все сервисы предоставляются синхронно, передачей сообщения от клиента к серверу и обратно. Клиент посылает сообщение серверу и блокируется. Сервер принимает сообщение от клиента, выполняет обработку запроса и затем отвечает на сообщение клиента, разблокируя его.

***обработчик прерывания (interrupt service routine)***

Подпрограмма, которой ядро передает управление (в привилегированном режиме) в результате аппаратного прерывания. Эта подпрограмма не имеет права выполнять системные вызовы и должна обеспечить возврат управления как можно скорее, поскольку ее приоритет реально выше, чем у любого потока в системе. Обработчики прерываний в QNX/ Neutrino могут возвращать структуру типа **struct sigevent**, которая указывает, какое событие, если нужно, следует сгенерировать.

### ***ответить на сообщение (reply to a message)***

Сервер отвечает на сообщение клиента, чтобы доставить клиенту результаты обработки его запроса.

### ***относительный таймер (relative timer)***

Таймер с моментом истечения, определяемым как смещение от текущего момента времени, например, «через 5 минут». Сравните с абсолютным таймером.

### ***передать сообщение (send a message)***

Поток может передавать сообщения другому потоку. Для передачи сообщения применяется семейство функций *MsgSend\*()*; передающий сообщение поток блокируется до тех пор, пока принимающий поток не ответит на это сообщение (см. «обмен сообщениями»). Поток, который передает сообщение, считается клиентом.

### ***периодический таймер (periodic timer, repetitive timer)***

Абсолютный или относительный таймер, который после истечения времени отсчета автоматически перезагружается, и продолжает делать так до явной отмены. Полезен для приема регулярных уведомлений.

### ***поток управления, «нить» (thread)***

Одиночный диспетчеризуемый поток управления. Реализация потоков обеспечивается непосредственно ядром Neutrino и соответствуют вызовам функций POSIX *pthread\*()*. Поток может быть синхронизирован с другими потоками (если таковые имеются) путем применения различных примитивов синхронизации, таких как мутексы, условные переменные, семафоры и т.д. Потоки подвергаются диспетчеризации по типу FIFO или RR (карусельного типа).

### ***принять сообщение (receive a message)***

Поток может принять сообщение, вызвав функцию *MsgReceive()* или *MsgReceivev()*. Если сообщений нет, поток заблокируется в ожидании сообщения (см. «обмен сообщениями»). Поток, который принимает сообщение, считается сервером.

### ***процесс (process)***

Недиспетчеризируемый объект, занимающий память и вмещающий в себя один или более потоков.

### ***разблокировать (unblock)***

Ранее заблокированный поток будет разблокирован, когда условие, на основе которого он был заблокирован, будет удовлетворено. Например, поток может быть заблокирован в ожидании сообщения. Как только ему будет послано сообщение, он будет разблокирован.

### ***семафор (semaphore)***

Примитив синхронизации потоков, с которым ассоциируется счетчик. Потоки могут вызывать функцию *sem\_wait()*, и не будут при этом блокироваться, если в момент запроса счетчик имел ненулевое значение. Вызывая функцию *sem\_wait()*, поток уменьшает значение счетчика. Если поток вызывает *sem\_wait()* в тот момент, когда счетчик равен нулю, поток блокируется, пока некоторый другой поток не вызовет функцию *sem\_post()*, увеличивая тем самым значение счетчика. Сравните с барьером.

### ***сервер, серверный процесс (server)***

Сервер представляет собой обычный процесс, работающий в кольце пользователя и предоставляющий клиентам определенные сервисы (обычно на основе файловых дескрипторов). Серверы обычно являются администраторами ресурсов, и существует обширная специализированная

библиотека функций, созданная компанией QSSL и содержащая готовые реализации многих функциональных возможностей администраторов ресурсов. Работа сервера состоит в том, чтобы принимать сообщения от клиентов, обрабатывать запросы и затем отвечать на сообщения, тем самым разблокируя клиентов. Любой поток может одновременно являться и клиентом, и сервером.

### ***сигнал (signal)***

Механизм, относящийся к ранним UNIX-системам, который применялся для отправки асинхронного уведомления о событиях от одного потока к другому. Сигналы не блокируют отправителя. Получатель сигнала может решить сам, обрабатывать ли сигнал синхронным способом, путем его явного ожидания. Сравните с импульсом.

### ***синхронный (synchronous)***

Это понятие применяется для указания на то, что данная операция синхронизирована по отношению к другой операции. Например, в процессе обмена сообщениями, когда сервер выполняет *MsgReply()* для ответа клиенту, говорят, что деблокирование клиента синхронно по отношению к операции ответа. Сравните с определением «асинхронный».

### ***соединение (connection)***

Понятие, говорящее о подключении клиента к каналу. Соединение устанавливается либо непосредственно клиентом (вызовом функции *ConnectAttach()*), либо «третьей» стороной от имени клиента (в случае вызова библиотечной функции *open()*). В любом случае, возвращаемый идентификатор соединения пригоден для использования в качестве дескриптора для всех операций обмена данными между клиентом и сервером.

### ***условная переменная (condition variable)***

Объект синхронизации, применяемый для множества потоков и характеризуемый как «точка встречи», в которой несколько потоков могут быть заблокированы в ожидании некоего сигнала (не путать с понятием сигнала в UNIX!). При поступлении сигнала один или более потоков разблокируются.

### *физический адрес (physical address)*

Адрес, который выставляется ЦП на шину, связанную с подсистемой памяти. Поскольку QNX/Neutrino работает в режиме виртуальной адресации, это означает, что диспетчер памяти должен транслировать виртуальные адреса, которые используются потоками, в физические адреса, пригодные для использования подсистемой памяти. Сравните с виртуальной адресацией и виртуальной памятью.

### *фрагментация/дефрагментация сообщений (scatter/gather)*

Применяется для определения операций обмена сообщениями, в которой множество различных фрагментов собираются ядром (на стороне либо клиента, либо сервера) (дефрагментация), после чего сообщение разбивается на фрагменты (их может быть и другое число, нежели было при дефрагментации) с другой стороны (фрагментация). Эта операция чрезвычайно полезна, когда, например, необходимо добавить к данным клиента заголовок прежде, чем их отправят серверу. Клиенту в таком случае следует определить вектор ввода/вывода (IOV), который должен содержать указатель на заголовок и его длину в виде первого элемента, а также указатель на данные и их длину в качестве второго элемента. Тогда ядро «дефрагментирует» эти данные и перешлет их серверу как один непрерывный объект. Действия на стороне сервера будут аналогичными.

Спасибо, что скачали книгу в [бесплатной электронной библиотеке Royallib.com](http://Royallib.com)

[Оставить отзыв о книге](#)

[Все книги автора](#)