**Ю.Б. Гриценко**

# СИСТЕМЫ РЕАЛЬНОГО ВРЕМЕНИ

Федеральное агентство по образованию

Томский государственный университет систем управления
и радиоэлектроники

**Ю.Б. Гриценко**

# СИСТЕМЫ РЕАЛЬНОГО ВРЕМЕНИ

**Учебное пособие**

Томский государственный университет систем управления
и радиоэлектроники
2006

УДК 681.3.066.014(075.8)
ББК 32.973.2-018я73
Г

Рецензенты:

**Гриценко Ю.Б.**

Г Системы реального времени: учеб. пособие. — Томск: Томск. гос. ун-т систем управления и радиоэлектроники, 2006. — 152 с.

ISBN

Рассмотрены вопросы организации и построения систем реального времени. Сделан обзор современных операционных систем реального времени. Основное внимание уделено вопросу построения операционной системы реального времени QNX (QNX Software Systems Limited).

Предназначено для студентов специальности 230102 — «Автоматизированные системы обработки информации и управление».

УДК 681.3.066.014(075.8)
ББК 32.973.2-018я73

ISBN

## Оглавление


Предисловие ..... 5

**1. Введение в системы реального времени**

1.1. Определения и классификации ОСРВ ..... 7

1.2. Области применения и вычислительные платформы ОСРВ ..... 9

1.3. Архитектуры построения ОСРВ ..... 12

**2. Функциональные требования к операционным системам реального времени**

2.1. Основные понятия ..... 19

2.2. Диспетчеризация потоков ..... 21

2.3. Уровни приоритетов и механизмы синхронизации ..... 23

2.4. Временные характеристики ОС ..... 27

**3. Описание операционных систем реального времени**

3.1. Стандарты на ОСРВ ..... 29

3.2. Категории ОСРВ ..... 31

3.3. ОСРВ на основе обычных операционных систем 33

3.4. Виды ОСРВ на базе собственных разработок ..... 40

**4. Системы реального времени: общее представление**

4.1. Основные характеристики ОС QNX ..... 57

4.2. Основные характеристики микроядра ОС QNX 59

4.3. Связь между процессами ..... 63

4.3.1. Связь между процессами посредством сообщений ..... 63

4.3.2. Связь между процессами посредством Proxy 70

4.3.3. Связь между процессами посредством сигналов ..... 71

4.3.4. Примеры связи между процессами посредством обмена сообщениями ..... 74

4.4. Сетевое взаимодействие ..... 79

4.5. Планирование процессов ..... 82

4.6. Первичная обработка прерываний ..... 84

**5. Администратор процессов QNX**
5.1. Создание процессов ……. 88
5.2. Состояния процессов ……... 92
5.3. Управление потоками …….. 95
5.4. Управление таймером …….. 98
5.5. Обработчики прерываний …… 105
**6. Управление ресурсами в ОС QNX**
6.1. Администраторы ресурсов …….. 107
6.2. Сетевые файлы в QNX ……. 108
6.3. Инсталляционные пакеты и их репозитарии ……. 111
6.4. Символьные устройства ввода/вывода …….. 113
6.5. Сетевая подсистема QNX …… 115
**7. Работа в QNX**
7.1. Начальная загрузка в QNX Neutrino ……... 131
7.2. Графический интерфейс пользователя Photon microGUI ……... 134
7.3. Печать в QNX …… 137
7.4. Средства анализа …….. 139
7.5. Средства визуальной разработки программ …….. 141
Литература ……... 151

## Предисловие

Автоматизация промышленных процессов предъявляет жесткие временные требования к разрабатываемому программному обеспечению. Данные требования обусловили возникновение нового класса операционных систем (ОС) — систем реального времени (СРВ). Эти системы должны обеспечить программное обеспечение всеми необходимыми информационными и аппаратными ресурсами для удовлетворения временных требований.

Дисциплина «Системы реального времени» изучает структуры, методы и алгоритмы построения современных СРВ, возможности функционирования операционной системы реального времени (ОСРВ) QNX. Базовыми понятиями для данной дисциплины являются следующие: построение систем реального времени, диспетчеризация потоков, установка уровней приоритетов, механизмы синхронизации и защиты от инверсии приоритетов.

Для изучения предлагаемого материала необходимы навыки программирования на языке высокого уровня Си, знания дисциплин «Операционные системы», «Архитектура и организация ЭВМ».

Учебное пособие по курсу «Операционные системы реального времени» представлено в семи разделах.

В первом разделе приведены различные трактовки определений систем реального времени, рассматриваются области применения и вычислительные платформы данных систем. Также в разделе уделено внимание обзору архитектур операционных систем реального времени и проанализировано типичное строение системы реального времени.

Второй раздел содержит функциональные требования к операционным системам реального времени. Наиболее важной функциональной характеристикой данных систем является диспетчеризация потоков.

Третий раздел включает: описание стандартов и категорий операционных систем реального времени, обзор систем реального времени на базе обычных операционных систем и собственных оригинальных разработок, краткие сведения о специализированных операционных системах.

Введение в операционную систему QNX и описание ее микроядра приведено в четвертом разделе, который содержит следующие сведения: общее представление об ОС QNX, основные характеристики микроядра ОС QNX, описание связей между процессами, вопросы сетевого взаимодействия, механизм планирования процессов и первичной обработки прерываний.

Пятый раздел содержит описание работы администратора процессов ОС QNX: создание процессов, изменение состояния процессов, управление таймером и обработкой прерываний.

В шестом разделе приведено описание управления ресурсами ОС QNX. Вводится понятие администраторов ресурсов, описываются виды ресурсов — файловые системы, инсталляционные пакеты, символьные устройства ввода/вывода, сетевая подсистема QNX.

В седьмом разделе рассматриваются особенности работы QNX: начальная загрузка ядра QNX, графический интерфейс пользователя, печать, средства анализа и описание визуальной среды разработки Photon Application Builder.

# 1. ВВЕДЕНИЕ В СИСТЕМЫ РЕАЛЬНОГО ВРЕМЕНИ

## 1.1. Определения и классификации ОСРВ

Программные системы, которые по своему назначению должны обрабатывать одновременные события или управлять одновременно выполняемыми операциями, инициируемыми внешними по отношению к ним программами или пользователями, являются параллельными по своей природе [1]. **Параллельные системы** делятся на следующие классы:

1) системы реального времени и встроенные системы (специального назначения);

2) обычные и распределенные операционные системы, компоненты которых функционируют на нескольких компьютерах;

3) системы управления базами данных и системы обработки транзакций;

4) распределенные сервисы прикладного уровня.

Первый класс параллельных систем представлен системами реального времени. Английский термин «real-time» (в русском языке ему соответствует понятие «реальное время») является наиболее спорным и сложным термином. Он применяется в различных научно-технических областях и подразумевает некие действия, продолжительность которых определяется внешними процессами. **Специфическая особенность систем реального времени** заключается в том, что к ним предъявляются строгие временные требования, диктуемые окружением или определяемые назначением данных систем [1]. Для понимания смысла понятия «система реального времени» приведем два определения:

1) система называется системой реального времени, если правильность ее функционирования зависит не только от логической корректности вычислений, но и от времени, за которое эти вычисления производятся, т. е. для событий, происходящих в такой системе время осуществления этих событий (КОГДА), так же важно, как и логическая корректность самих событий [2].

2) системы реального времени — это системы, которые предсказуемо (в смысле времени реакции) реагируют на непредсказуемые (по времени появления) внешние события [3].

Одной из функций таких систем может быть выполнение определенных действий в ответ на сигналы тревоги, и очень важно, чтобы системы отвечали на сигналы с определенной скоростью. В связи с этим существует разделение систем реального времени на два типа:

1) системы с жесткими временными характеристиками — системы жесткого реального времени;

2) системы с нежесткими временными характеристиками — системы мягкого реального времени.

***Системой жесткого реального времени*** называется система, в которой неспособность обеспечить реакцию на какие-либо события в заданное время является отказом и ведет к невозможности решения поставленной задачи. Многие теоретики ставят здесь точку, из чего следует, что время реакции в жестких системах может составлять и секунды, и часы, и недели. Однако большинство практиков считают, что время реакции в системах жесткого реального времени должно быть все-таки минимальным. Большинство систем жесткого реального времени являются системами контроля и управления. Такие СРВ сложны в реализации, так как для них предъявляются особые требования в вопросах безопасности.

Точного определения для ***систем мягкого реального времени*** не существует, поэтому отнесем сюда все СРВ, не попадающие в категорию жестких. Так как система мягкого реального времени может не успевать выполнять все задания (ВСЕ) постоянно (ВСЕГДА) в заданное время, возникает проблема определения критериев успешности (нормальности) ее функционирования. Вопрос этот совсем не простой, так как в зависимости от функций системы этими критериями могут быть максимальная задержка в выполнении каких-либо операций, средняя своевременность обработки событий и т. п. Более того, эти критерии влияют на выбор оптимального алгоритма планирования задач.

СРВ можно разделить на системы специализированные и универсальные**.**

***Специализированной СРВ*** называется система, где конкретные временные требования определены. Такая система должна быть специально спроектирована для удовлетворения этих требований.

***Универсальная СРВ*** предназначена для выполнения произвольных (заранее не определенных) временных задач без применения специальной техники. Разработка таких систем, безусловно, является самой сложной задачей, хотя обычно требования, предъявляемые к таким системам, мягче, чем требования для специализированных систем.

Если СРВ строится как программный комплекс, то в общем виде она может быть представлена как комбинация трех компонентов:

1) прикладное программное обеспечение;
2) операционная система реального времени;
3) аппаратное обеспечение.

В целом ряде задач автоматизации программные комплексы должны работать как составная часть более крупных автоматических систем без непосредственного участия человека. В таких случаях СРВ называют встраиваемыми. ***Встраиваемые системы*** (Embedded system) можно определить как программное и аппаратное обеспечение, составляющее компоненты другой, большей системы и работающее без вмешательства человека [3].

## 1.2. Области применения и вычислительные платформы ОСРВ

В течение длительного времени основными потребителями СРВ были военная и космическая отрасли. Сейчас ситуация изменилась, и СРВ применяются даже при производстве товаров народного потребления.

Основными областями применения СРВ являются:

- отрасли ВПК:
  - бортовое и встраиваемое оборудование;
  - радары, системы измерения и управления;
  - цифровые видеосистемы, симуляторы;

– ракеты, системы определения местоположения и привязки к местности;

• промышленность:

– автоматические системы управления производством; автоматические системы управления технологическими процессами;

– симуляторы, системы управления мотором, автоматическое сцепление и другое в автомобилестроении;

– системы сбора информации, системы управления данными и оборудованием и другое в энергетике;

– коммуникационное оборудование, сетевые коммутаторы, телефонные станции в телекоммуникациях;

– банковское оборудование: банкоматы …;

• отрасли, производящие товары народного потребления:

– мобильные телефоны;

– цифровое телевидение: мультимедиа, видеосервисы, цифровые телевизионные декодеры и т. д.;

– компьютерное и офисное оборудование;

– банкоматы.

Часто ОСРВ существуют в нескольких вариантах, например в полном и сокращенном, когда объем системы составляет несколько килобайтов.

Системы реального времени могут функционировать на следующих платформах: «обычные» компьютеры; промышленные компьютеры, встраиваемые системы.

**«Обычные» компьютеры** по логическому устройству совпадают с настольными компьютерами, но аппаратное устройство несколько отлично. Для обеспечения минимального времени простоя в случае технической неполадки процессор, память и другие элементы размещаются на съемной плате, вставляемой в специальный разъем так называемой «пассивной» платы. В другие разъемы этой платы вставляются платы периферийных контроллеров и другое оборудование. Сам компьютер помещается в специальный корпус, обеспечивающий защиту от пыли и механических повреждений. В качестве мониторов используются жидкокристаллические дисплеи, иногда с сенсорочувствительным покрытием. Доминирующее положение на этих компьютерах занимают процессоры Intel 80x86.

Подобные вычислительные системы обычно не используются для непосредственного управления промышленным или иным оборудованием. Они служат как терминалы для взаимодействия с промышленными компьютерами и встроенными контроллерами, для визуализации состояния оборудования и технологического процесса. На таких компьютерах, как правило, в качестве операционной системы используют классические ОС (с разделением времени) с дополнительными программными комплексами, адаптирующими эти системы к требованиям реального времени.

**Промышленные компьютеры с**остоят из одной платы, на которой размещены процессор, контроллер памяти и память различных видов (ОЗУ, ПЗУ, статическое ОЗУ, флэш-память). Несмотря на наличие контроллеров SCSI (Small Computer System Interface) очень часто СРВ работает без дисковых накопителей. Это связано с тем, что дисковые накопители не отвечают предъявляемым к системам реального времени требованиям, таким как надежность, устойчивость к вибрациям, габаритам и времени готовности после включения питания.

Плата помещается в специальный корпус, в котором установлены разъемы шины и источник питания. Корпус обеспечивает специальный температурный режим, защиту от пыли и механических повреждений. В корпус вставляются цифро-аналоговые и аналогово-цифровые преобразователи, через которых осуществляется ввод/вывод управляющей информации, управление электромоторами и т.п.

Среди промышленных процессоров доминируют процессоры семейств PowerPC (Motorola — IBM), Motorola 68xxx (Motorola). Также широкую нишу занимают процессоры семейств SPARC (SUN), Intel (Intel), ARM (ARM). При выборе процессора определяющими факторами являются получение требуемой производительности при наименьшей тактовой частоте, а также время между переключением задач и реакции на прерывания.

Промышленные компьютеры используются для непосредственного управления промышленным или иным оборудованием. Они часто не имеют монитора и клавиатуры. Для взаимодействия с ними используются обычные компьютеры, соединенные с ними через порты или Ethernet.

Отметим основные особенности ОСРВ, диктуемые необходимостью работы на промышленном компьютере:

- система часто должна работать на бездисковом компьютере и осуществлять начальную загрузку из ПЗУ. В силу этого должны учитываться следующие факторы:
  - критически важным является размер системы;
  - для экономии места в ПЗУ часть системы хранится в сжатом виде и загружается в ОЗУ по мере необходимости;
  - система позволяет исполнять код и в ОЗУ, и в ПЗУ;
  - при наличии свободного места в ОЗУ система часто копирует код из более медленного ПЗУ в ОЗУ;
  - сама система, как правило, создается на другом, «обычном», компьютере;
- система должна использовать как можно большее число типов процессоров, что дает возможность потребителю выбрать процессор необходимой мощности;
- система должна по возможности поддерживать более широкий ряд специального оборудования (периферийные контроллеры, таймеры и т. д.);
- критически важным параметром является возможность предсказания времени реакции на прерывания.

**Встраиваемые системы** устанавливаются внутрь оборудования, которым они управляют. Для крупного оборудования совпадают с промышленными компьютерами. Для меньшего оборудования представляют собой процессор с сопутствующими элементами, размещенными на одной плате с другими электронными компонентами этого оборудования.

## 1.3. Архитектуры построения ОСРВ

Операционные системы реального времени могут быть построены на основе следующих архитектур:

1) монолитных;
2) уровневых;
3) клиент-серверных;
4) объектных.

**Монолитные ОС**

За свою историю архитектура операционных систем претерпела значительные изменения. Один из первых принципов построения так называемых монолитных ОС (рис. 1.1) заключался в представлении ОС как набора модулей, взаимодействующих между собой различным образом внутри ядра системы и предоставляющих прикладным программам входные интерфейсы для обращений к аппаратуре.

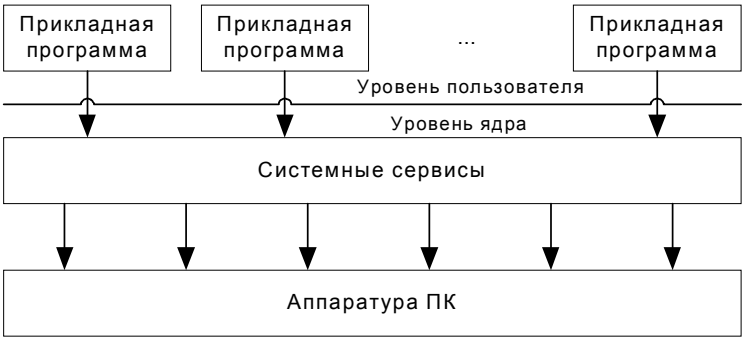


Рис. 1.1. Архитектура монолитной ОС

Системный уровень (уровень ядра) монолитной ОС состоит из трех частей:

1) интерфейс между приложениями и ядром (API — application program interface);

2) собственно ядро системы;

3) интерфейс между ядром и оборудованием.

API в таких системах выполняет двойную функцию:

1) управление взаимодействием прикладных процессов и систем;

2) обеспечение непрерывности выполнения кода системы (отсутствие переключений задач во время исполнения кода программы).

Основным ***преимуществом архитектуры монолитной ОС*** является ее относительная быстрота работы по сравнению с другими архитектурами. Однако это достигается написанием значительной части кода системы на ассемблере.

***Главный недостаток*** такой архитектуры — плохая предсказуемость ее поведения, вызванная сложным взаимодействием модулей системы между собой [4]. Однако большинство современных ОС, как реального времени, так и общего назначения, строятся именно по этому принципу. ***К другим недостаткам*** можно отнести следующие:

- системные вызовы, требующие переключения уровней привилегий (от пользовательской задачи к ядру), должны быть реализованы как прерывания или ловушки (специальный тип исключений), что существенно увеличивает время работы;
- работа ядра не может быть прервана пользовательской задачей (non-preemptable), т. е. высокоприоритетная задача не всегда получает управление в момент работы низкоприоритетной;
- сложность переноса на новые архитектуры процессоров, из-за ассемблерных вставок;
- негибкость и сложность развития. Изменение части ядра требует его полной перекомпиляции.

**Уровневые ОС**

В задачах автоматизации широкое распространение в качестве ОСРВ получили уровневые ОС (рис. 1.2). Примером такой ОС является хорошо известная система MS-DOS. В системах этого класса прикладные приложения могли получить доступ к аппаратуре не только посредством ядра системы или ее резидентных сервисов, но и непосредственно. По такому принципу строились ОСРВ в течение многих лет. По сравнению с архитектурой монолитных ОС архитектура уровневых обеспечивает следующее ***преимущество***: значительно большую степень предсказуемости реакций системы, а также возможность осуществлять быстрый доступ прикладных приложений к аппаратуре. ***Недостатком*** архитектуры этих систем является отсутствие механизма многозадачности. В рамках такой архитектуры проблема обработки асинхронных событий сводилась к буферизации сообщений, а затем последовательному опросу буферов и обработке. При этом соблюдение критических сроков обслуживания обеспечивалось высоким быстродействием вычислительного комплекса по сравнению со скоростью протекания внешних процессов.

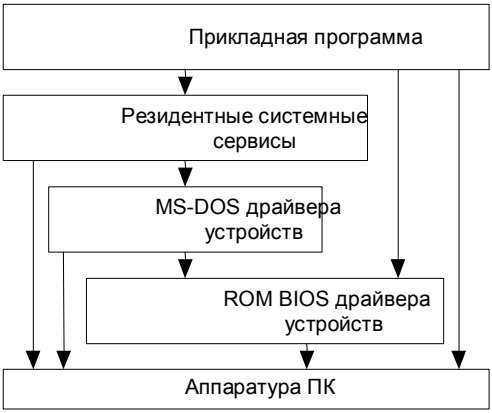


Рис. 1.2. Архитектура уровневой ОС

**Клиент-серверные ОС (модульная архитектура)**

Одной из наиболее эффективных архитектур для построения операционных систем реального времени считается архитектура клиент-сервер. Общая схема ОС работающей по этой технологии представлена на рис. 1.3.

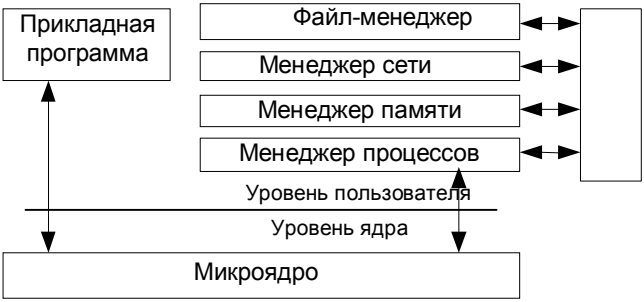


Рис. 1.3. Архитектура клиент-серверных ОС

Основным принципом модульной архитектуры является вынесение сервисов ОС в виде серверов на уровень пользователя, а микроядро выполняет функции диспетчера сообщений между клиентскими пользовательскими программами и серверами — системными сервисами. Данная архитектура дает ***преимущества*** с точки зрения требований к ОСРВ и встраиваемым системам:

• повышается надежность ОС, так как каждый сервис является, по сути, самостоятельным приложением, и его легче отладить и отследить ошибки;

• система лучше масштабируется, поскольку ненужные сервисы могут быть исключены из системы без ущерба для ее работоспособности;

• повышается отказоустойчивость системы, поскольку «зависший» сервис перезапускается без перезагрузки системы.

Среди известных ОСРВ, реализующих архитектуру микроядра, можно отметить OS9 и QNX.

***Недостатки*** данной архитектуры те же, что и у архитектуры монолитной системы. Проблемы перешли с уровня API на уровень микроядра. Системный интерфейс по-прежнему не допускает переключения задач во время работы микроядра, только сократилось время пребывания в этом состоянии. Часть API может также быть реализована на ассемблере. Проблемы переносимости сократились, но остались.

**Объектная архитектура на основе объектов-микроядер**

В данной архитектуре API отсутствует вообще. Взаимодействие между компонентами системы (микроядрами) и пользовательскими процессами осуществляется посредством обычного вызова функций, поскольку система и приложения написаны на одном языке (C++). Это обеспечивает максимальную скорость системных вызовов. Фактическое равноправие всех компонентов системы обеспечивает возможность переключения задач в любое время, т. е. система полностью preemptable[1]. Объектно-ориентированный подход полностью обеспечивает модульность, безопасность, легкость модернизации и повторного использования кода.

Роль API выполняет компилятор и динамический редактор объектных связей (linker). При старте приложения динамический редактор linker загружает нужные ему микроядра. Таким образом, в отличие от всех предыдущих типов систем, не все компоненты ОС должны быть загружены в оперативную память. Если микроядро уже загружено другим приложением, то оно повторно не загружается, а используется код и данные уже имеющегося мик-

[1] Наличие возможности прерывания выполнения задачи в режиме ядра.

роядра. Все это позволяет сократить объем используемой оперативной памяти. Поскольку разные приложения используют одни микроядра, то данные приложения должны работать в одном адресном пространстве. Следовательно, система не может использовать виртуальную память и, таким образом, работает быстрее. У нее исключаются задержки на трансляцию виртуального адреса в физический. Основным представителем такой архитектуры является архитектура ОСРВ Soft Kernel.

***Типичное строение операционных систем реального времени*** можно разделить на три слоя (рис. 1.4):

1) ядро, содержащее только строгий минимум функций необходимый для работы системы:

- управление, синхронизация и взаимодействие задач;
- управление памятью;
- управление устройствами ввода-вывода;

2) система управления, содержащая ядро и ряд дополнительных сервисов, расширяющих возможности ядра:

- расширенное управление памятью;
- расширенное управление устройствами ввода-вывода;
- расширенное управление задачами;
- управление файлами;
- обеспечение взаимодействия системы и внешнего оборудования и т. д.;

3) система реального времени, содержащая систему управления и набор утилит: средства разработки и визуализации и т. д.


Рис. 1.4. Типичное строение ОСРВ

## Вопросы для самопроверки

1. Дайте определение системы реального времени.
2. Что понимают под параллельными системами, и какие типы параллельных систем существуют?
3. Приведите классификацию систем реального времени.
4. Что означают термины «система жесткого реального времени» и «система мягкого реального времени»?
5. В каких областях применяются системы реального времени?
6. На каких платформах существуют системы реального времени?
7. Какие процессоры доминируют среди промышленных компьютеров?
8. Какие типы архитектур операционных систем Вы знаете? Назовите их преимущества и недостатки.
9. Приведите типичную структуру построения операционной системы реального времени.

# 2. ФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ К ОПЕРАЦИОННЫМ СИСТЕМАМ РЕАЛЬНОГО ВРЕМЕНИ

## 2.1. Основные понятия

Расширение области применения СРВ привело к повышению требований к этим системам. В настоящее время обязательным условием, предъявляемым к ОС, претендующей на применение в задачах реального времени, является реализация в ней **механизмов многозадачности**. Та же тенденция присутствует и в ОС общего назначения. Но для СРВ к реализации механизмов многозадачности предъявляется ряд дополнительных, по сравнению с системами общего назначения, требований. Определяются эти требования обязательным свойством СРВ — предсказуемостью.

Многозадачность предполагает параллельное выполнение нескольких действий, однако практическая реализация такой работы упирается в проблему совместного использования ресурсов вычислительной системы. Главным ресурсом, распределение которого между несколькими задачами называется **диспетчеризацией** (scheduling), является процессор. Поэтому в однопроцессор-ной системе по-настоящему параллельное выполнение нескольких задач невозможно. Существует достаточно большое количество различных методов диспетчеризации, и основные из которых будут рассмотрены далее.

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

**Задача** в терминах ОС и программных комплексов предполагает наличие двух разных понятий: процессов и потоков. ***Процесс*** является более масштабным представлением задачи, поскольку обозначает независимый модуль программы или весь исполняемый файл с его адресным пространством, состоянием регистров процессора, счетчиком команд, кодом процедур и функций. ***Поток*** — составная часть процесса — обозначает последовательность исполняемого кода. Каждый процесс содержит как минимум один поток, при этом максимальное количество потоков в пределах одного процесса в большинстве ОС ограничено только объемом оперативной памяти вычислительного комплекса. Потоки, принадлежащие одному процессу, разделяют его адресное пространство, поэтому они могут легко обмениваться данными, а время переключения между такими потоками (то есть время, за которое процессор переходит от выполнения команд одного потока к выполнению команд другого) оказывается значительно меньшим, чем время переключения между процессами. В связи с этим в приложениях реального времени параллельно выполняемые задачи стараются максимально компоновать в виде потоков, исполняющихся в пределах одного процесса.

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

Поток может находиться в одном из следующих состояний:

- активный поток, выполняемый системой в данный момент;
- поток в состоянии готовности, выполняемый в настоящий момент либо ожидающий своей очереди;
- блокированный поток, который не может выполняться по некоторым причинам (например, ожидание события или освобождения нужного ресурса).

Далее в подразд. 2.2 и 2.3 рассматриваются функциональные требования, предъявляемые ОСРВ.

## 2.2. Диспетчеризация потоков

Методы диспетчеризации, т. е. предоставления разным потокам доступа к процессору, в общем случае могут быть разделены на две группы.

**Первая группа методов диспетчеризации** применяется к потокам, которые разделяют процессор и имеют одинаковый приоритет, т. е. их важность с точки зрения системы одинакова:

- ***FIFO (First In First Out)***. Сначала выполняется задача, первой вошедшая в очередь, при этом она выполняется до тех пор, пока не закончит свою работу или не будет заблокирована в ожидании освобождения некоторого ресурса или события. После этого управление передается следующей в очереди задаче;
- ***карусельная многозадачность*** (round robin). При этом методе диспетчеризации в системе задается специализированная константа, определяющая продолжительность непрерывного выполнения потока, так называемый квант времени выполнения (time slice). Таким образом, выполнение потока может быть прервано либо окончанием его работы, либо блокированием в ожидании ресурса или события, либо завершением кванта времени (того самого time slice). После этого управление передается следующему в очередности потоку. По окончании времени выполнения последнего потока управление передается первому потоку, находящемуся в состоянии готовности. Таким образом, выполнение каждого потока разбито на последовательность временных циклов.

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

- в наиболее простом случае, если в состояние готовности переходят два потока с разными приоритетами, процессорное время передается тому, у которого более высокий приоритет. Данный метод называется ***приоритетной многозадачностью***,

но его использование в таком виде связано с рядом сложностей. При наличии в системе одной группы потоков с одним приоритетом и другой группы с другим, более низким приоритетом, при карусельной диспетчеризации каждой группы в системе с приоритетной многозадачностью потоки низкоприоритетной группы могут вообще не получить доступа к процессору;

- одним из решений проблем приоритетной многозадачности стала так называемая ***адаптивная многозадачность***, широко применяющаяся в интерфейсных системах. Суть метода заключается в том, что приоритет потока, не выполняющегося какой-то период времени, повышается на единицу. Восстановление исходного приоритета происходит после выполнения потока в течение одного кванта времени или при блокировке потока. Таким образом, при карусельной многозадачности очередь (или «карусель») более приоритетных потоков не может полностью заблокировать выполнение очереди менее приоритетных потоков;
- в задачах реального времени предъявляются специфические требования к методам диспетчеризации, поскольку передача управления потоку должна определяться критическим сроком его обслуживания (deadline-driven scheduling). В наибольшей степени этому требованию соответствует ***вытесняющая приоритетная многозадачность***. Суть этого метода заключается в том, что, как только поток с более высоким, чем у активного потока, приоритетом переходит в состояние готовности, активный поток вытесняется (т. е. из активного состояния принудительно переходит в состояние готовности) и управление передается более приоритетному потоку.

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

Для СРВ, применяющихся в обработке периодических событий, в 1970 г. Лиу и Лейленд [5] предложили математический аппарат, позволяющий определить, является ли система диспетчируемой. Этот аппарат называется «Частотно монотонный анализ» (ЧМА) (Rate Monotonic Analyzing) [2]. Эффективность данного математического аппарата привела к тому, что ЧМА был принят в качестве стандарта такими организациями, как USA Department of Defense, Boeing, General Dynamics, Magnavox, Mitre, NASA, Panamax, и др. Среди организаций, установивших ЧМА в качестве стандартного средства анализа и разработки систем жесткого реального времени, можно также отметить IBM Federal Sector Division, US Navy и European Space Agency [6].

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

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

## 2.3. Уровни приоритетов и механизмы синхронизации

Для организации параллельного выполнения нескольких потоков, как было отмечено выше, зачастую необходимо разделение этих потоков по степени важности (или критическому сроку обслуживания). Среди совокупности параллельно выполняющихся задач выделяются потоки жесткого реального времени, потоки мягкого реального времени и потоки, не критичные ко времени обслуживания. Каждая из указанных групп должна иметь свой уровень приоритетов, к тому же потоки жесткого реального времени в ряде случаев должны иметь индивидуальные значения приоритетов. Практический опыт разработки систем реального времени свидетельствует о том, что увеличение

количества разноприоритетных потоков приводит к непрозрачности и непредсказуемости системы. Однако не всегда это выглядит именно так.

На сегодняшний день существует ряд инструментов математического анализа, позволяющих распределить приоритеты между несколькими потоками таким образом, чтобы они гарантированно выполняли свои критические сроки обслуживания. Если же для данного набора потоков реализовать это невозможно, то результаты математического анализа покажут, какие именно потоки имеют критическое отношение срока обслуживания ко времени выполнения.

Упомянутый ранее аппарат ЧМА позволяет провести такое исследование для случая периодических критических времен обслуживания [7]. Однако для его применения анализируемые потоки должны иметь уникальные значения приоритетов, определяемые периодом каждого потока. В связи с этим требованием разработчики ОСРВ закладывают в своих системах достаточно большое количество приоритетов. Для примера в QNX 6.x их 64, а в Windows CE и VxWorks — 256.

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

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

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

Проблема инверсии приоритетов оказалась настолько важной для ОСРВ, что реализация механизмов защиты для таких случаев в системе вынесена в отдельное функциональное требование к ОСРВ [3]. Инверсия приоритетов возникает, когда два потока, высокоприоритетный (**В**) и низкоприоритетный (**Н**), разделяют некий общий ресурс (**Р**). Предположим также, что в системе присутствует третий поток, приоритет которого находится между приоритетами **В** и **Н**. Назовем его средним (**С**). Если поток **В** переходит в состояние готовности, когда активен поток **Н**, и **Н** заблокировал ресурс **Р**, то поток **В** вытеснит поток **Н**, и **Р** останется заблокирован. Когда **В** понадобится ресурс **Р**, то он сам перейдет в заблокированное состояние. Если в состоянии готовности находится только поток **Н**, то ничего страшного не произойдет, **Н** освободит заблокированный ресурс и будет вытеснен потоком **В**. Но если на момент блокирования потока **В** в состоянии готовности находится поток **С**, приоритет которого выше, чем у **Н**, то активным станет именно он, а **Н** опять будет вытеснен и получит управление только после того, как **С** закончит свою работу. Подобная задержка вполне может привести к тому, что критическое время обслуживания потока **В** будет пропущено. Если **В** — это поток жесткого реального времени, то подобная ситуация недопустима.

Широко распространенным и проверенным механизмом защиты от инверсии приоритетов является механизм ***наследова-***

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

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

Другой, несколько менее распространенный механизм, называется Протокол Предельного Приоритета (Priority Ceiling Protocol) [8]. Механизм заключается в добавлении к стандартным свойствам объектов синхронизации параметра, определяемого максимальным приоритетом потока, которые к объекту обращаются. Если такой параметр установлен, то приоритет любого потока, обращающегося к данному объекту синхронизации, будет увеличен до указанного уровня и, таким образом, не сможет быть вытеснен никаким потоком, который может нуждаться в заблокированном им ресурсе. После разблокирования ресурса приоритет потока понижается до начального уровня. Таким образом, получается нечто вроде предварительного наследования приоритетов. Однако этот механизм имеет ряд серьезных недостатков. Во-первых, на разработчика ложится работа по «обучению» объектов синхронизации их уровню приоритетов. Во-вто-

рых, возможны задержки в запуске высокоприоритетных потоков на время отработки низкоприоритетных потоков. В целом, максимально эффективно этот механизм может быть использован в случае, когда имеется один поток жесткого реального времени и несколько менее приоритетных потоков, разделяющих с ним ресурсы.

## 2.4. Временные характеристики ОС

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

Еще одна важная отличительная особенность ОСРВ от систем общего назначения заключается в ***независимости поведения системы и времени ее реакций на различные события от количества текущих задач***. В большинстве систем общего назначения такие параметры, как время переключения контекста

потока прямо зависят от количества потоков в системе, в системах же реального времени этой зависимости быть не должно.

## Вопросы для самопроверки

1. Дайте определение понятию механизма диспетчеризации.
2. Расскажите о проблеме разделения ресурсов.
3. В чем отличие процессов и потоков?
4. Какое свойство потока отвечает за предоставление ему процессорного времени?
5. В каких состояниях может находиться поток?
6. Какие методы диспетчеризации Вы знаете?
7. В чем отличие вытесняющей многозадачности от адаптивной и приоритетной?
8. Сколько уровней приоритетов должна иметь операционная система реального времени?
9. Какая серьезная проблема возникает при блокировании ресурсов?
10. Какие механизмы существуют для решения проблемы инверсии приоритетов?
11. Какие требования по временным характеристикам предъявляются к системам реального времени?

# 3. ОПИСАНИЕ ОПЕРАЦИОННЫХ СИСТЕМ РЕАЛЬНОГО ВРЕМЕНИ

## 3.1. Стандарты на ОСРВ

Стандарты на операционные системы реального времени стали появляться лишь после того, как уже был создан ряд ОСРВ. Основной целью введения стандартов является облегчение переноса программного обеспечения из одной системы в другую.

Основной целью разработчиков ОСРВ является обеспечение максимальной скорости ее работы и компактности. С одной стороны, среди ОСРВ преобладают системы с уникальным устройством, а с другой стороны, при создании ОСРВ необходимо соблюдение требований существующих стандартов. При этом даже системы, декларирующие свою совместимость с некоторым стандартом, обычно содержат ряд расширений, выходящих за его рамки. Тем не менее важность стандартов состоит в том, что они фактически выступают в качестве аксиоматической базы, задающей определения рассматриваемых объектов и понятий.

**Стандарт SCEPTRE** (Standardisation du Coeur des Exécutifs des Produits Temps Réel Européens) — европейский стандарт на основы систем реального времени («sceptre» по-французски означает скипетр) — разрабатывался в 1980-1990-е годы. За время его создания появились новые концепции в ОСРВ, не все из которых успели найти отражение в стандарте. В стандарте объединены усилия инженеров и исследователей в разработке групп спецификаций для промышленных приложений, даны определения и описания набора методов и подходов, используемых в ОСРВ.

Стандарт SCEPTRE определяет семь основных целей, которые должны достигаться при создании ОСРВ:

1) адекватность поставленной задаче;

2) безопасность (система должна быть максимально устойчивой к аппаратным и программным сбоям);

3) минимальная стоимость;

4) максимальная производительность;

5) переносимость (возможность реализовывать систему на процессоре другого типа, адекватном поставленной задаче);

6) адаптивность (способность системы приспосабливаться к новому управляемому ей оборудованию и задачам);

7) модульность (система, достаточная для решения поставленной задачи на имеющемся оборудовании, должна состоять из независимых компонент, из которых можно построить систему).

Сервис, предоставляемый операционной системой, разделен в стандарте на следующие группы:

- коммуникации (межпроцессорное взаимодействие);
- синхронизация процессов;
- контроль и планирование задач;
- управление памятью;
- управление прерываниями и оборудованием ввода/вывода;
- высокоуровневый интерфейс ввода/вывода и управление периферийными устройствами;
- управление файлами;
- управление транзакциями (сообщениями и передачами данных);
- обработка ошибок и исключений;
- управление временем.

**Стандарт POSIX** (Portable Operating System Interface) **1003.1b**, ранее существовавший под рабочим именем 1003.4 и разработанный IEEE (Institute of Electrical and Electronics Engineers), представляет собой расширение стандарта POSIX 1001 на операционные системы класса Unix, позволяющего использовать последние в качестве ОСРВ. Большинство приложений Unix могут быть перенесены в такие системы, поскольку стандарт POSIX 1003.1b обеспечивает единый с системами Unix программный интерфейс (API).

Стандарт POSIX 1003. 1b состоит из следующих частей:

1) POSIX 1003.1 — стандарт на основные компоненты операционной системы, API для процессов, файловой системы, устройств и т. д.;

2) POSIX 1003.1b — стандарт на основные расширения для систем реального времени;

3) POSIX 1003.1c — стандарт на задачи (threads);

4) POSIX 1003.1d — стандарт на дополнительные расширения для систем реального времени, такие как, например, поддержка обработчиков прерываний;

5) POSIX 1003.2 — стандарт на основные утилиты.

Стандарту POSIX 1003 с расширением 1003.1b удовлетворяют такие системы, как Lynx, VxWorks, QNX. Некоторые системы, например CHORUS, обеспечивают поддержку стандарта POSIX 1003.1b при загрузке поставляемого программного обеспечения, т.е. имеют два типа API (оригинальный собственный и стандартный).

## 3.2. Категории ОСРВ

Операционные системы реального времени по способу разработки программного обеспечения подразделяются категории.

1. **Self-Hosted ОСРВ** — системы, в которых пользователи могут разрабатывать приложения, работая в самой ОСРВ. Обычно это предполагает, что ОСРВ поддерживает файловую систему, средства ввода-вывода, пользовательский интерфейс, имеет компиляторы, отладчик, средства анализа программ, текстовые редакторы, работающие под управлением ОСРВ.

Достоинством таких систем является более простой и наглядный механизм создания и запуска приложений, работающих на том же классе машин, на котором они и были разработаны. Недостатком этих систем является тот факт, что во время реальной эксплуатации промышленному компьютеру часто вообще не требуется наличие пользовательского интерфейса и запуск тяжеловесных программ, например компилятора. Следовательно, большинство возможностей ОСРВ не используются и только занимают память и другие ресурсы компьютера.

Как правило, Self-Hosted ОСРВ применяются на «обычных» компьютерах промышленного исполнения.

2. **Host/Target ОСРВ** — это системы такой категории, когда системы, на которых разрабатываются приложения (host), и системы, на которых запускаются приложения (target), различны. Связь между компьютерами осуществляется с помощью последовательного соединения (COM-порта), Ethernet, общей шины

VME или compact PCI. В качестве host системы обычно выступает компьютер под управлением Unix или Windows NT, в качестве target системы выступает промышленный или встраиваемый компьютер под управлением ОСРВ.

Достоинством таких систем является использование всех ресурсов «обычной» системы для создания приложений и уменьшение размеров ОСРВ за счет включения только необходимых приложению компонентов. Недостатком является относительная сложность программных компонентов: кросс-компилятора, удаленного загрузчика и отладчика и т.д.

В настоящее время рост мощности промышленных компьютеров позволяет использовать Self-Hosted ОСРВ на большем количестве вычислительных систем. С другой стороны, увеличивается распространение встраиваемых систем (в разнообразном промышленном и бытовом оборудовании), что расширяет сферу применения Host/Target-систем, поскольку при больших объемах выпуска цена системы является определяющим фактором.

В зависимости от происхождения ОСРВ разделяют на следующие группы:

1) **обычные ОС**, используемые в качестве ОСРВ. Часто к обычным ОС добавляют дополнительные модули, реализующие поддержку специфического оборудования, а также планирование задач и обработку прерываний в соответствии с требованиями к ОСРВ, и сглаживающие невозможность прервать ядро системы. Все системы относятся к категории Self-Hosted;

2) **собственно ОСРВ**, которые могут относиться к категориям Self-Hosted и Host/Target. Некоторые ОСРВ поддерживают обе модели;

3) **специализированные ОСРВ**, разрабатываемые для конкретного микроконтроллера его производителем. Они часто не являются полноценными ОС, а представляют единый модуль с приложением и обеспечивают только необходимый минимум функций. Все такие системы относятся к категории Host/Target.

## 3.3. ОСРВ на основе обычных операционных систем

Группа обычных операционных систем, используемых в качестве систем реального времени, включает следующие виды систем:

1) системы на основе Linux;

2) системы на основе Windows NT.

**Системы на основе Linux** (свободно распространяемой версии Unix) получили значительное распространение в настольных компьютерах благодаря свободному распространению и качеству. В настоящий момент система Linux может работать на процессорах Intel 80x86, а также и на процессорах Alpha, SPARC, PowerPC, ARM, Motorola 68xxx, MIPS. Открытость ее исходных текстов позволяет реализовывать на ее основе специализированные системы и обеспечивать поддержку нового оборудования.

Адаптация системы Linux к требованиям для систем реального времени происходит по следующим направлениям:

***поддержка стандартов POSIX***, касающихся систем реального времени. Стандарт POSIX 1003.1c (работа с задачами) уже поддерживается, стандарт POSIX 1003.1b (работа с расширением систем реального времени) поддерживается лишь частично: реализованы механизмы управления памятью и механизмы планирования задач;

***поддержка специального оборудования***, важнейшим из которого является шина VME. Уже существует поддержка моста VME-PCI. Также для системы реального времени важным является повышение разрешения таймера системы;

***реализация механизма preemption*** (приоритетное прерывание обслуживания) для ядра системы. Этот механизм является необходимым для того, чтобы систему можно было назвать системой реального времени, но он также является очень сложным для реализации. Linux надолго запрещает прерывания при входе в ядро системы и не является preemptive.

Существует несколько проектов реализации ***preemption*** для ядра Linux. По способу решения этой задачи их можно разделить на две группы:

1) реализация механизма preemption путем переписывания ядра системы. На этом пути можно достичь самых качественных результатов, но на данный момент значительных успехов в этом плане нет по следующим причинам:

- значительный объем работы, связанный с большим объемом ядра;
- слишком высокая скорость изменения ядра, причем изменения вносятся, без учета требований систем реального времени;

2) реализация механизма preemption путем разработки микроядра, отвечающего за диспетчеризацию прерываний и задач. Ядро Linux работает как задача с низким приоритетом. Само ядро лишь незначительно изменено для предотвращения блокирования им аппаратных прерываний. Задачи в такой системе разделены на две группы:

- процессы, работающие под управлением только микроядра (не использующие функции ядра Linux);
- процессы, работающие под управлением Linux (обычные приложения), а также задачи, работающие под управлением микроядра, но использующие функции Linux. Это процессы неудовлетворяющие требованиям реального времени, поскольку могут быть блокированы ядром Linux.

Недостатком такого подхода является необходимость реализации микроядра, обеспечивающего функционирование процессов реального времени. Например, если процесс реального времени затребовал такой ресурс, как коммуникационный порт, то драйвер этого порта надо перенести из ядра Linux в микроядро. Наиболее законченной реализацией этого подхода является проект RT-Linux.

***Система RT-Linux*** создана в New Mexico Institute of Mining and Technology (USA) и является свободно распространяемой. Эта система продолжает разрабатываться энтузиастами в ряде университетов мира. Представляет собой простейшее микроядро, отвечающее за создание и планирование задач, обеспечение их взаимодействия и диспетчеризацию прерываний. Реализован простейший приоритетный механизм планирования и единственный механизм взаимодействия — очередь сообщений FIFO.

Ядро Linux работает как самая низкоприоритетная задача. В само ядро внесены исправления — макроопределения, отвечающие за запрещения/разрешения прерываний, заменены на соответствующие функции микроядра. Задачи Linux не могут прервать ядро Linux, задачи же микроядра могут.

Такая структура накладывает некоторые ограничения на задачи реального времени, которые не могут легко использовать различные драйвера Linux, не имеют доступа к сети и т. д. Зато они могут обмениваться данными со стандартами задачами Linux.

Для обмена данными между процессами реального времени и процессами Linux реализованы простые очереди FIFO.

Типичное приложение состоит из двух частей: задачи реального времени, непосредственно работающей с аппаратурой, и обыкновенной задачи Linux, выполняющей остальные операции, такие как сохранение данных на диске, пересылка их по сети, работа с пользователем (GUI) и т. д.

Самый короткий период для периодически вызываемых задач реального времени в RT-Linux на Pentium 120 — менее 150 мкс. Задачи, вызываемые по прерыванию, могут иметь намного меньший период.

Ядро реального времени не защищает от перегрузок. Если одна из задач реального времени полностью утилизирует процессор, ядро Linux, имея самый низкий приоритет, не получит управления и система «повиснет».

Задачи реального времени запускаются в адресном пространстве ядра и с привилегиями ядра и могут быть реализованы, например, при помощи модулей Linux.

Минимальный размер системы для записи в ПЗУ (без X-Windows) — 2,7 мб.

**Системы на основе Windows NT,** по поводу использования которых в качестве ОСРВ ведется множество дискуссий. Имеются аргументы и «за», и «против». С точки зрения специалистов фирмы Microsoft наличие нижеприведенного набора возможностей Windows NT позволяет считать данную систему операционной системой реального времени, а именно:

- многопроцессность и многозадачность системы;
- поддержка многопроцессорности;

- preemption задач;
- preemption прерываний и возможность их маскирования;
- асинхронный ввод/вывод.

Прямой доступ к оборудованию посредством интерфейса HAL (Hardware Abstraction Level). HAL обеспечивает изоляцию приложения от деталей реализации оборудования, обеспечивая платформенно-независимый прямой доступ к оборудованию.

Windows NT используется специальная схема приоритетов. Все приоритеты разделены на две класса:

1) класс динамических приоритетов (0..15). Приоритеты динамически меняются планировщиком по алгоритму, близкому к принятому в Unix;

2) класс приоритетов реального времени (16, 22-26, 31). Данные приоритеты фиксированы, задачи из этого класса планируются на основе приоритетной очереди и получают управление раньше задач с динамическими приоритетами.

Пространство ввода/вывода для задач из класса реального времени не участвует в страничном обмене механизма виртуальной памяти.

Для закрепления страниц задачи в памяти существует специальный системный вызов (VirtualLock()).

Представляются объекты синхронизации: критические секции, таймеры, события, mutex и др.

Существуют также мнения других специалистов по поводу ряда имеющихся в Windows NT недостатков, которые не позволяют считать ее операционной системой реального времени:

- ядро системы не является preemptive;
- механизм DPC (Differed Procedure Call), вызываемый обработчиком прерываний, имеет недостатки:

– все процедуры DPC получают один и тот же приоритет и обрабатываются планировщиком в порядке поступления (FIFO). Тем самым низкоприоритетные прерывания будут обрабатываться ранее высокоприоритетных, но поступивших позднее;

– система не дает возможности узнать, количество DPC в очереди, поэтому невозможно определить, когда начнет обрабатываться прерывание. Таким образом, существует случайная задержка между приходом прерывания и началом его обработки;

• для каждого прерывания только один экземпляр DPC может быть в очереди. Следовательно, процедура DPC должна обладать способностью обрабатывать повторяющиеся прерывания, а оборудование должно обеспечивать буферизацию прерываний во избежание потери данных. Это удорожает как драйверы устройств, так и оборудование;

• малое количество приоритетов в классе реального времени (7 приоритетов) приводит к тому, что множество задач будут иметь одинаковый приоритет и планироваться алгоритмом типа round robin. Следовательно, время до начала исполнения задачи будет случайной величиной. Оно зависит от текущей загруженности системы;

• не решена проблема инверсии приоритетов. Вместо традиционного для ОСРВ механизма наследования приоритетов Windows NT назначает задаче случайный уровень приоритета, позволяющий ей начать работу, что непредсказуемо и неприемлемо для ОСРВ. Поскольку приоритет задач класса реального времени не меняется, то этот механизм действует только на задачи из динамического класса. Тем самым ситуация только ухудшается: приоритет низкоприоритетных задач реального времени не меняется, а высокоприоритетные задачи могут быть вытеснены совсем низкоприоритетными задачами из динамического класса;

• высокоприоритетные задачи могут блокироваться низкоприоритетными. Некоторые компоненты ядра работают на уровне приоритета динамического класса. Следовательно, некоторые системные вызовы приводят к понижению приоритета и вывода задачи из класса реального времени. При этом она может быть блокирована другой задачей, имевшей до этого более низкий приоритет;

• все страницы неактивного процесса, например, ожидающего данных, могут быть перенесены на диск, несмотря на закрепление их вызовом VirtualLock(). Это приводит к случайным задержкам при активации процесса (например, при поступлении ожидавшихся данных);

• для Windows NT официально не приводятся времена системных вызовов и блокирования прерываний;

• систему нельзя использовать без дисплея и клавиатуры;

• система предъявляет слишком большие запросы для ОСРВ на память.

Для устранения этих недостатков ряд компаний предлагает программные и аппаратные средства. Основные идеи их построения те же, что и для Linux. Поскольку исходный код системы недоступен, то, в отличие от Linux, существует только один способ приспособить систему к требованиям реального времени: разработать микроядро, обеспечивающее надлежащее планирование задач и диспетчеризацию прерываний, а Windows NT выполнять как процесс.

Приведем несколько известных разработок такого рода;

***система Hyperkernel***, созданная фирмой «Nematron», представляет собой ядро, обеспечивающее детерминированное планирование и работающее на уровне привилегий 0 процессора Intel 80x86 вместе с Windows NT. Задачи Hyperkernel не доступны Windows NT. Для них определены 8 уровней приоритетов с preemptive планированием. В качестве средства разработки используются стандартные для Windows NT компиляторы Visual C/C++ и специальные библиотеки. Используются API Win32 и стандартный HAL. Разрешение таймера: 1 мкс, минимальный квант времени 20 мкс. Время задержки на прерывание — 5 мкс, переключение контекста — 4 мкс на Intel Pentium 133МГц;

***система LP RT-Technology,*** выпускаемая фирмой LP Electronic GmbH, включает три компонента:

1) плату для шины ISA, обеспечивающую 7 дополнительных уровней прерываний. Для взаимодействия с остальной системой плата использует NMI — немаскируемое прерывание процессора Intel 80x86;

2) LP-RTWin Toolkit — комплект разработчика ISR, используемый совместно с Visual C/C++ и отладчиком SoftICE фирмы «NuMega»;

3) LP-VxWin RTAcc — программный комплекс, обеспечивающий сосуществование Windows NT и VxWorks на одном PC. Две ОС взаимодействуют посредством протокола TCP/IP через разделяемую память. В качестве средств разработки используется Tornado — комплект разработчика ОСРВ VxWorks;

***система Realtime ETS Kernel,*** разработанная специалистами фирмы «Phar Lap SoftWare» в двух вариантах:

1) TNT Embedded ToolSuite, Realtime Edition, включающий: Realtime ETS Kernel, ETS TCP/IP, отладчик CodeView с поддержкой Borland Turbo Debugger, ассемблер 386ASM, linker, поддержку компиляторов Visual C/C++, Borland C/C++, Watcom C/C++ и API Win32;

2) Realtime ETS Kernel — полная замена Windows NT — включает компактное ядро (28Кб), поддерживающее Win32 API и использующее стандартные библиотеки компиляторов. Ядро имеет 32 уровня приоритетов и может быть записано в ПЗУ;

***система Component Integrator,*** выпускаемая фирмой «VenturCom, Inc.» В отличие от предыдущих систем, здесь не вводится новое ядро, а предлагаются пакеты, расширяющие ряд узких мест Windows NT:

1) Embedded Component Kit (ECK), включающий поддержку работы без дисплея и клавиатуры, уменьшение потребности в памяти, отключение страничного обмена, поддержку флэш-памяти и шины VME;

2) Real-time Extension (RTX), включающий поддержку таймe-ров (разрешение — 1 мкс, минимальный квант времени — 100 мкс), отключение виртуальной памяти, работа с физической памятью, 128 новых уровней привилегий, время задержки реакции на прерывание — 5–20 мкс;

***система Willows RT,*** выпускаемая фирмой «Willows SoftWare, Inc.» В отличие от предыдущих систем, здесь приложение вообще не будет запускаться в Windows NT. Windows NT используется только как платформа разработки, приложение же будет работать под управлением настоящей ОСРВ (QNX, VxWorks, …). Система состоит из трех частей:

1) библиотек TWIN32 и TWIN16, заменяющих библиотеки Microsoft и реализующих API;

2) TWIN Platform Abstraction Layer — ядра, обеспечивающего системные вызовы Microsoft);

3) TWIN Binary Interface, обеспечивающей трансляцию вызовов Platform Abstraction Layer в API используемой ОСРВ.

## 3.4. Виды ОСРВ на базе собственных разработок

**Система LynxOS** выпускается фирмой «Lynx Real Time Systems (Los Gatos, USA)».

Основные характеристики системы:

1) категория — Self-Hosted;

2) архитектура — на основе микроядра;

3) стандарт — собственный и POSIX 1003;

4) процессоры (target) — Intel 80x86, Motorola 68xxx, SPARC, PowerPC;

5) размер ядра — 33 кб;

6) средства синхронизации и взаимодействия — POSIX 1003;

7) планирование: приоритетное, FIFO, Round Robin, preemptible;

8) средства разработки: комплект разработки C/C++, включающий компиляторы, отладчик, анализатор, X-Windows/Motif для Lynx, Total View — многопроцессный отладчик.

***Система LynxOS 4.x*** фирмы «LynuxWorks» является результатом более чем 15-летнего опыта и предназначена для создания ПО встроенных систем, работающих в режиме жесткого реального времени. Ее могут использовать производители комплектного и телекоммуникационного оборудования, в частности изготовители бортовых систем военного применения. Разработка может осуществляться как на самой целевой системе (Self-Hosted), так и на инструментальном компьютере (host), готовое ПО работает на целевой системе (target).

Система сертифицирована по стандарту POSIX. Это означает, что LynxOS проверена аккредитованными независимыми экспертами на полное соответствие этому стандарту. В частности, LynxOS была сертифицирована на соответствие POSIX 1003.1-1996 фирмой «Mindcraft» (www.mindcraft.com). Это уникальное свойство отличает ее от большинства других ОСРВ, которые являются POSIX-совместимыми (compatible) и в лучшем случае соответствуют стандарту POSIX на 90–95%.

LynxOS поддерживает многозадачные и многопотоковые приложения. Она может использоваться для приложений с вы-

сокими требованиями ко времени реакции и надежности. Программы, написанные и скомпилированные в ОС Linux, могут запускаться и работать в среде LynxOS без каких-либо изменений в исходных текстах и без перекомпилирования. Это свойство LynxOS является уникальным для ОСРВ и очень удобным для пользователей (например, если отсутствуют исходные тексты). LynxOS обеспечивает совместимость с Linux на уровне ABI (Application Binary Interface), форматов объектных файлов, вызовов API, динамически подключаемых библиотек (DLL), компоновки и загрузки на этапе выполнения. Как уже отмечалось, система полностью поддерживает стандарт POSIX.1003.la, а также подразделы POSIX. 1003.lb и POSIX. 1003.1c. Она может работать на разных аппаратных платформах (IA-32, PowerPC, MIPS, ARM, xScale) и поддерживает самые современные сетевые средства и Интернет-технологии. В ней предусмотрены необходимые средства для создания систем с возможностями «горячей замены» и «высокой доступности» (Hot Swap, High Availability), а также устройств с высоким коэффициентом резервирования. Существует версия LynxOS-178, сертифицированная в соответствии со стандартом DO-178.

***Систему LynxOS-178 2.0*** рассмотрим более подробно. Эта система в значительной степени удовлетворяет самым высоким современным требованиям к ОСРВ, разработанным для авиации [9]. Ключевое свойство LynxOS-178 2.0 — это поддержка нескольких полностью разделенных по времени, памяти и ресурсам разделов в соответствии с требованиями ARINC 653[1].

LynxOS-178 2.0 поддерживает:

[1] ARINC 653 (Avionics Application Software Standard Interface) разработан компанией «ARINC» («Aeronautical Radio, Inc.») в 1997 г. Стандарт определяет универсальный программный интерфейс APEX (Application/Executive) между операционной системой авиационного компьютера и прикладным программным обеспечением (www.arinc.com/cf/store/documentlist.cfm). Требования к интерфейсу между прикладным ПО и сервисами операционной системы определяются таким образом, чтобы разрешить прикладному ПО контролировать диспетчеризацию, связь и состояние внутренних обрабатываемых элементов. В 2003г. принята новая редакция этого стандарта. ARINC 653 в качестве одного из основных требований для ОСРВ в авиации вводит архитектуру изолированных (partitioning) виртуальных машин.

- до 16 разделов (виртуальных машин), включая корневой;
- до 64 процессов внутри каждого раздела;
- до 51 потока (нити) внутри каждого процесса;
- диспетчеризацию реального времени потоков внутри раздела;
- POSIX-функции межпроцессного взаимодействия внутри раздела.

Каждый раздел полностью изолирован, так что распространение сбоев между разделами невозможно. Это разделение относится к процессорному времени, адресному пространству и ресурсам в каждом разделе как к пути для изоляции сбоев и гарантии доступности всех ресурсов.

С помощью LynxOS-178 фиксированные разделы обслуживаются как виртуальные машины LynxOS. Каждый прикладной процесс оперирует внутри его собственной среды операционной системы, как если бы он работал на своем собственном процессоре. Это применимо ко всем ресурсам процессора и именованным пространствам. Такая абстракция избавляет разработчика прикладной ОС от дополнительных усилий при программировании сложной системы. Управление разделами контролируется с помощью таблицы конфигурации виртуальных машин (Virtual-Machine Configuration Table — VCT) и является обязательным в среде LynxOS-178, что позволяет разработчику сконцентрироваться на создании приложений, а не на разделении системы.

Кроме того, LynxOS-178 поддерживает разделение систем, совместимых с RTCA DO-255 (Radio Transport Corporation of America), которое разрешает системе выполнять программное обеспечение с различными уровнями безопасности в соответствии со стандартом DO-178B[1] в разных разделах (виртуальных машинах). Это означает, что ОС может выполнять приложение с уровнем A (по DO-178B) на одной виртуальной машине и при-

[1] Стандарт DO-178B «Software Consideration in Airborne Systems and Equipment Certification» разработан и поддерживается ассоциацией «RTCA» («Radio Technical Commission for Aeronautics» (www.rtca.org). Первая его версия была принята в 1982 г., вторая (DO-178A) — в 1985, текущая (DO-178B) — в 1992 г., а принятие новой версии ( DO-178C) намечено на сентябрь 2005 г.

ложение с уровнем C на другой, причем оба уровня работают на одном и том же процессоре в рамках одной и той же системы.

Стандартом DO-178B предусмотрено пять уровней серьезности отказа, и для каждого из них определен набор требований к программному обеспечению, которые должны гарантировать работоспособность всей системы в целом при возникновении отказов данного уровня серьезности отказа:

- уровень A — обеспечение защиты от сбоев, приводящих к катастрофическим (catastrophic) последствиям, и соблюдение 66 требований;
- уровень B — ПО должно обеспечивать защиту от сбоев, приводящих к опасным (hazardous) последствиям, и удовлетворять 65 требованиям;
- уровень C — обеспечение защиты от сбоев, приводящих к серьезным (major) последствиям, и соблюдение 57 требований;
- уровень D — обеспечение защиты от сбоев, приводящих к незначительным (minor) последствиям, и соблюдение 28 требований;
- уровень E — обеспечение защиты от сбоев, не приводящих ни к каким последствиям.

Компания «LynuxWorks» разработала и сертифицировала на соответствие уровню A стандарта DO-178 стек TCP/IP — LCS (Lynx Certifiable Stack). Основные особенности LCS:

- выполняется на выделенном коммуникационном процессоре Motorola 8260 и может взаимодействовать с другими системами;
- является расширением стандартного стека TCP/IP, обеспечивающим детерминизм и повышенную надежность;
- обеспечивает полную реализацию IPv4 (Internet Protocol Version 4);
- верифицирован по 100%-му покрытию кода по критерию MCDC (Modified Condition Decision Coverage);
- поддерживает статически преконфигурированные IP-адреса и порты, что повышает надежность и безопасность системы;
- имеет прикладной интерфейс с LynxOS-178. Взаимодействие выделенной виртуальной машины LCS с виртуальными машинами LynxOS-178 («хостами») осуществляется по шине PCI.

Фирма «LynuxWorks» развивает LynxOS-178, чтобы обеспечить соответствие этой ОС максимальным требованиям Common Criteria —общим критериям для оценки секретности информационных технологий (Common Criteria for Information Technology Security Evaluation, CCITSE) — это набор требований и условий секретности, одобренный Агентством национальной безопасности и Национальным институтом стандартов и технологий США, а также соответствующими органами других стран. В 1999 г. CCITSE получил статус международного стандарта ISO 15408. Дополнительную информацию по сертификации CCITSE можно найти на сайте www.commoncriteria.org.

CCITSE определяет уровни гарантии секретности — EAL (Evaluation Assurance Level). Common Criteria оценивает не только безопасность и надежность продуктов, но и процессы их разработки и поддержки, что гарантирует быстрое решение проблем. Выделены семь EAL-уровней гарантии секретности, которые применяются в соответствующих случаях:

1) EAL1 («Functionally tested») при необходимости минимальной конфиденциальности, обеспечение секретности не рассматривается как важное требование;

2) EAL2 («Structurally tested»), когда от системы требуется средний уровень гарантированной секретности в отсутствие полной информации обо всех процедурах разработки;

3) EAL3 («Methodically tested and checked»), когда разработчики или пользователи требуют среднего уровня гарантированной секретности и исчерпывающего исследования операционной системы и этапов ее разработки, не прибегая к существенной переработке ОС;

4) EAL4 («Methodically designed, tested and reviewed»), когда разработчики или пользователи требуют высокого уровня гарантированной секретности операционной системы и специальной доработки уже существующей ОС для обеспечения этих требований;

5) EAL5 («Semi formally designed and tested»), когда разработчики или пользователи требуют высокого уровня гарантированной секретности операционной системы и строгого подхода к проектированию, так чтобы эти свойства были заложены уже на

этапе проектирования при использовании специальных средств обеспечения секретности;

6) EAL6 («Semi formally verified, designed and tested»), когда существует высокий уровень опасных ситуаций и оправданы высокие затраты на защиту от несанкционированного доступа;

7) EAL7 («Formally verified, designed and tested») в приложениях с очень высокой ценой несанкционированного доступа.

В 2006 г. должен выйти прототип операционной системы LynxOS, отвечающий EAL-7. LynuxWorks разрабатывает новое ядро ОСРВ — LynxSecure, которое будет содержать всего около 8000 строк исходного кода и будет также соответствовать требованиям EAL-7.

**Система OS-9** выпускается фирмой Microware (USA).

Основные характеристики системы:

1) категория — Host/Target;

2) архитектура — на основе микроядра;

3) стандарт — собственный;

4) ОС разработки (host) — Unix/Windows;

5) процессоры (target) — Intel 80x86, Motorola 68xxx, ARM, PowerPC, MIPS;

6) линии связи host/target — последовательный канал, Ethernet;

7) размер ядра — 16 кб;

8) средства синхронизации и взаимодействия — разделяемая память, сигналы, семафоры, события и т. д.;

9) планирование: приоритетное, FIFO, специальный механизм планирования, preemptible;

10) средства разработки: Hawk — интегрированная среда разработки на C/C++, PersonalJava — виртуальная машина Java.

В 1979 г. совместными усилиями фирм «Microware» и «Motorola» была разработана операционная система реального времени для микропроцессора 6809. Версия Level I OS-9/6809 была способна адресовать 64 Кб памяти. Версия Level II OS-9/6809, используя прогрессивные в то время аппаратные средства динамической трансляции адресов, была способна адресовать уже до 2 Мб памяти и стала применяться в ряде популярных компьютерных систем, например, таких как Tandy Color Computer III.

В 1982 г. «Microware» (уже независимо от «Motorola») портировала OS-9 для семейства микропроцессоров 68000, создав систему OS-9/680X0 для 16 и 32-разрядных микропроцессоров и микроконтроллеров. Код системы лишь на 20 % был написан на языке высокого уровня (фрагмент распределения памяти и настраиваемая часть загрузочного кода), остальная часть с целью достижения максимальной производительности написана на Ассемблере. За пять лет с момента появления (1987 г.) OS-9/680X0 стала признанным промышленным стандартом де-факто для операционных систем реального времени (и абсолютным лидером по применимости в промышленных приложениях на базе технологии VME).

В списке поддерживаемых микропроцессоров OS-9/68K наиболее полно представлено семейство 68K — от младшего в серии MC68000 до 32-разрядного, суперскалярного MC68060. Наибольшую популярность и распространение получили версии 2.4/2.5 системы, и в настоящее время пользователями по достоинству оценены мощь и надежность новейшей версии системы 3.0.1.

OS-9000 — переносимая версия OS-9, написанная, главным образом, (95 %) на языке C. Оставшиеся, критичные с точки зрения производительности, участки кода написаны на Ассемблере. Как результат только 5 % OS-9000 необходимо переписать, чтобы перенести ее на новый процессор. Теоретически OS-9000 может быть перенесена на любую современную микропроцессорную архитектуру.

Для пользователя OS-9 и OS-9000 UNIX-подобная среда (модель процессов, межпроцессные коммуникации, многопользовательская файловая система и большое количество стандартных для UNIX утилит) позволяет программисту, знакомому с UNIX, в минимально короткий срок почувствовать себя уверенно в среде реального времени OS-9. Единство архитектурных решений, совместимость приложений на уровне исполняемого кода, технология реализации интегральных средств разработки, а также дизайна и состава системных продуктов — эти факторы объединяют операционные системы реального времени OS9 и OS9000 в семейство операционных систем OS9 (рис. 3.1).

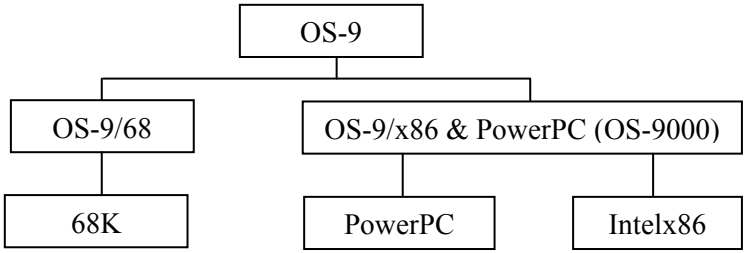


Рис. 3.1. Семейство операционных систем OS9

Система OS-9 является модульной, чрезвычайно гибко конфигурируемой, высокопроизводительной, встраиваемой системой реального времени. Именно такой она задумывалась с самого начала, и именно таким видится путь ее дальнейшего развития.

Совокупность требований, исходя из которых проектировалась OS-9, в самом общем случае сводится к обеспечению следующих потребительских свойств создаваемой коммерческой ОСРВ:

1) минимизация объема (компактность);

2) гибкость (модифицируемость);

3) возможность наращивания функциональности;

4) доступность по цене (и в первую очередь для выпускаемой массово продукции);

5) охват наиболее широкого спектра задач/приложений реального времени.

Если первые четыре пункта вполне понятны, то последний требует более полной характеристики. OS-9 предоставляет профессиональный набор программных системных средств реального времени, позволяющий в наиболее широком спектре приложений найти оптимальное сочетание производительности и функциональности средств уровня операционной системы для поддержания высокой надежности и производительности целевой системы РВ. Именно в этом предлагается рассмотреть некоторые архитектурные особенности OS-9. Тем более что критерий производительности ОСРВ хоть и важен для небольшого класса приложений РВ, но появление высокопроизводительного

и недорогого аппаратного обеспечения снижает степень исключительности требования по этому критерию.

Ядро OS-9 обеспечивает: диспетчеризацию процессов на основе приоритетов с разделением (необязательным) времени, обслуживание прерываний, обмен информацией между процессами, обработку ошибок, распределение и защиту всех разделяемых системных ресурсов. При реализации каждой из указанных функций предпринимались все меры обеспечения максимальной степени надежности. К примеру, ядро управляет всеми разделяемыми ресурсами (получая и отвечая на запросы исполняющихся процессов), вместо того чтобы просто позволить различным процессам конкурировать за разделяемый ресурс. При таком подходе гарантируется, что допущенная программистом ошибка при создании прикладной программы не приведет к краху всей системы. Кроме того, защита выделенной ядром исполняющемуся процессу памяти выполняется диспетчером памяти.

В системе имеются следующие требующие защиты ресурсы:

- содержимое памяти;
- регистры портов ввода/вывода;
- регистры системных часов;
- процедуры перехвата сигналов в процессах;
- процедуры обработки ошибок;
- интервалы приостановки исполнения;
- приоритеты диспетчеризации;
- пользовательские и групповые привилегии.

Изменение любых из этих величин, выполняемое в процессе работы системы, должно осуществляться очень осторожно и тщательно во избежание возможного краха системы. Единственный способ гарантии того, что все изменения будут производиться с соблюдением необходимых мер предосторожности, — это поручить ядру задачи распределения и защиты этих ресурсов. При прекращении исполнения процесса (нормальным образом или аварийно) ядро определяет, какие ресурсы были связаны с этим процессом в момент прекращения выполнения, и освобождает эти ресурсы.

Система OS-9 эффективна при обслуживании прерываний. Прерывания в OS-9 обрабатываются соответствующими подпро-

граммами, которые могут быть установлены в операционной системе либо драйвером устройства, либо прикладным процессом.

Базовые файловые менеджеры OS-9 предназначены для обеспечения обмена информацией между процессами и предоставляют приложениям OS-9 доступ к различным последовательным устройствам типа принтеров и терминалов, а также к устройствам внешней памяти типа дисков (жестких, гибких, электронных и оптических) и лент.

Модульная структура OS-9 позволяет системному разработчику выбирать именно те функциональные блоки, которые требуются данному приложению. Для обеспечения локального хранения и регистрации данных в систему может быть включена поддержка дисковых и ленточных устройств. Сетевые файловые менеджеры обеспечивают доступ к самым разным сетевым устройствам по протоколу TCP/IP и «прозрачно» связывать между собой различные системы. Любая из опций OS-9 легко может быть добавлена в систему для поддержки изменившихся требований к системе.

**Система VxWorks** выпускается фирмой Wind River Systems (Alameda, CA, USA).

Основные характеристики системы:

1) категория — Host/Target;

2) архитектура — монолитная;

3) стандарт — собственный и POSIX 1003;

4) ОС разработки (host) — Unix/Windows;

5) процессоры (target) — Intel 80x86, Motorola 68xxx, Intel 80960, ARM, PowerPC, MIPS, Alpha, SPARC:

6) линии связи host/target — последовательный канал, Ethernet, шина VME;

7) размер ядра — 5–8 кб.

8) средства синхронизации и взаимодействия — POSIX 1003.

9) планирование: приоритетное, preemptible.

10) средства разработки: Tornado — интегрированная среда разработки на C/C++, VxSim — эмулятор для Unix, WindView — графический визуализатор состояния задач.

Операционная система VxWorks является системой с кросс-средствами разработки прикладного программного обеспечения,

разработка ведется на инструментальном компьютере (host) в среде Tornado для последующего исполнения на целевой машине (target) под управлением VxWorks. Данная ОС имеет большое количество версий для различных сфер деятельности.

Операционная система VxWorks построена, как и положено ОС жесткого реального времени, по технологии микроядра, т. е. на нижнем непрерываемом уровне ядра выполняются только базовые функции планирования задач и управления коммуникацией/синхронизацией. Все остальные функции операционной системы более высокого уровня (управление памятью, вводом/выводом, сетевые средства и т. д.) базируются на простых функциях нижнего уровня, что позволяет обеспечить быстродействие и детерминированность ядра, а также легко строить необходимую конфигурацию операционной системы.

В многозадачном ядре wind применен алгоритм планирования задач, учитывающий приоритеты и включающийся по прерываниям. В качестве основного средства синхронизации задач и взаимоисключающего доступа к общим ресурсам в ядре wind применены семафоры. Имеется несколько видов семафоров, ориентированных на различные прикладные задачи: двоичные, целочисленные, взаимного исключения и POSIX.

Все аппаратно-зависимые части VxWorks вынесены в отдельные модули, для того чтобы разработчик встроенной компьютерной системы мог самостоятельно переставить VxWorks на свою нестандартную целевую машину. Этот комплект конфигурационных и инициализационных модулей называется Board Support Package (BSP) и поставляется для стандартных компьютеров (VME-процессор, PC или Sparcstation) в исходных текстах. Разработчик нестандартной машины может взять за образец BSP наиболее близкий по архитектуре стандартный компьютер и перенести VxWorks на свою машину путем разработки собственного BSP с помощью BSP Porting Kit.

Базовые сетевые средства VxWorks: UNIX-networking, SNMP и STREAMS. VxWorks была первой ОСРВ, в которой реализован протокол TCP/IP с учетом требований к этим системам. С тех пор VxWorks поддерживает все сетевые средства, стандартные для UNIX: TCP/UDP/ICMP/IP/ARP, Sockets, SLIP/CSLIP/PPP, telnet/rlogin/rpc/rsh, ftp/tftp/bootp, NFS (клиент и сервер).

Реализация SNMP-агента с поддержкой как MIB-I, так и MIB-II предназначена для применения VxWorks в интеллектуальном сетевом оборудовании (хабы, мосты, маршрутизаторы, повторители) и других устройствах, работающих в сети.

STREAMS — стандартный интерфейс для подключения переносимых сетевых протоколов к операционным системам, реализован в VxWorks как в версии SVR3, так и в версии SVR4. Таким образом, в VxWorks можно инсталлировать любой протокол, имеющий STREAMS-реализацию, как стандартный (Novell IPX/SPX, DECNET, Apple-Talk и пр.), так и специализированный.

Wind River Systems анонсировала (1994 г.) программу WindNet, по которой ведущие фирмы-производители программных средств в области коммуникаций интегрировали свои программные продукты с VxWorks. На сегодняшний день — это сетевые протоколы X.25, ISDN, ATM, SS7, Frame Relay и OSI; CASE-средства разработки распределенных систем на базе стандартов ROOM (Real-Time Object Oriented Modelling) и CORBA (Common Object Request Broker Architecture); менеджмент сетей по технологиям MBD (Management By Delegation) и CMIP/GDMO (Common Management Information Protocol/Guidelines for Definition of Managed Objects).

**Система Soft Kernel** выпускается фирмой «Microprocess» (Coubevoie, France).

Основные характеристики:

1) категория — Host/Target;

2) архитектура — на основе объектов-микроядер;

3) стандарт — собственный;

4) ОС разработки (host) — Unix/Windows;

5) процессоры (target) — Motorola 68xxx, Intel 80960, ARM, PowerPC, ARM, SPARC;

6) линии связи host/target — последовательный канал, Ethernet.

7) размер ядра — 40 кб;

8) средства синхронизации и взаимодействия — семафоры, сигналы, события, почтовые ящики.

9) планирование: приоритетное, preemptible;

10) средства разработки: SoftWorks — интегрированная среда разработки на C/C++.

По принципу своей работы инструментальные средства Soft Kernel и Soft Environment, созданные компанией «OSE», напоминают пакет, служащий буфером между операционной системой и тем или иным аппаратным модулем (BSP — Board Support Package), только в данном случае в роли такого модуля выступает интерфейс 32-разрядных Windows-приложений (Win32 Api). Запустив одновременно несколько инструментов Soft Kernel на одной хост-машине, можно создать модель сетевой или многопроцессорной среды и вести разработку соответствующего проекта.

OSE Soft Kernel предоставляет пользователям поддержку приложений симуляции операционных систем реального времени (RTOS), запущенных на OSE TM. Она включает расширенные продукты разработки и поддержки компанией «OSE» встроенных систем.

OSE Soft Kernel позволяет OSE real-time-процессам быть запущенными на компьютере. При помощи механизмов симуляции ускоряется разработка и тестирование приложений без предварительного переноса приложения на аппаратное обеспечение. Это симулятор однопроцессорной системы (Single CPU System). Приложения могут быть расположены на машине так, как будто они оба запущены на OSE системе.

Традиционно имеется жёсткая граница между симуляцией и операцией в СРВ.

Графическое представление OSE Soft Kernel интегрировано с OSE Illuminator, инструментом разработки и отладки. Illuminator имеет дружеский графический интерфейс, который предоставляет лёгкий доступ к его инструментам и плагинам. Дополнительно OSE Soft Kernel позволяет пользователям отлаживать приложения в debug-режиме, непосредственно сгенерировав код. Это намного эффективней, чем если бы отладка производилась с помощью удаленных аппаратных средств, когда пользователю необходимо произвести загрузку, зафиксировать ошибку и исправить ее, а затем вновь осуществить загрузку и проверку на наличие каких-либо других ошибок.

Приложения, запущенные под OSE Soft Kernel, могут использовать все системы, совместимые с real-time kernel.

В крупных телекоммуникационных проектах число пользователей среды Soft Kernel Environment может достигать нескольких сот человек, что свидетельствует о несомненных достоинствах инструмента Soft Kernel в условиях сегодняшней рыночной гонки.

**Система RTC** (Real Time Craft) выпускается фирмой «GSI-TECSI» (Paris, France).

Основные характеристики:

1) категория — Host/Target (RTC) и Self-Hosted (RTC/PC);

2) архитектура — монолитная;

3) стандарт — собственный и SCEPTRE;

4) ОС разработки (host) — Unix/Windows;

5) процессоры (target) — Intel 80x86, Motorola 68xxx, Intel 80C166, Am39K;

6) линии связи host/target — Ethernet;

7) размер ядра — 1,5 кб для RTC, 4 кб для RTC/PC;

8) средства синхронизации и взаимодействия — SCEPTRE;

9) планирование: приоритетное, FIFO, preemptible;

10) средства разработки — компилятор C, отладчик Watcom RTC.

Существует журнал RTC, в котором публикуется большое число статей, посвященных ОСРВ. Множество статей этого журнала помещено в Интернете. дополнительную информацию можно получить на сайте: http://www.rtcgroup.com

**Система CHORUS** выпускается фирмой « Chorus Système» (Saint Quentin Yvelines, France). В ноябре 1997 г. фирма приобретена компанией «Sun Microsystems» (Menlo Park, CA, USA).

Основные характеристики:

1) категория — Host/Target (CHORUS/Micro и CHO-RUS/ClassiX) и Self-Hosted (CHORUS/MiX);

2) архитектура — на основе микроядра;

3) стандарт — собственный и POSIX 1003;

4) ОС разработки (host) — CHORUS/Unix/Windows;

5) процессоры (target) — Intel 80x86, Motorola 68xxx, Motorola 88xxx, SPARC, PowerPC, T805, MIPS, PA-RISC, YMP (Cray), DEC Alpha;

6) линии связи host/target — последовательный канал, Ethernet;

*7)* размер ядра — 10 кб для CHORUS/Micro, 50 кб для CHORUS/ClassiX.

8) средства синхронизации и взаимодействия — POSIX 1003;

9) планирование: приоритетное, FIFO, preemptible (вытесняющая);

10) средства разработки: CHORUS/Harmony — интегрированная среда разработки C/C++, включающая компиляторы, отладчик, анализатор; CHORUS/JaZZ — виртуальная машина Java.

**Система QNX** выпускается фирмой «QNX SoftWare Systems» (Canada & USA).

Основные характеристики:

1) категория — Self-Hosted;

2) архитектура — на основе микроядра;

3) стандарт — POSIX 1003;

4) процессоры (target) — Intel 80x86;

5) линии связи host/target — последовательный канал, Ethernet;

6) размер ядра — 60 кб;

7) средства синхронизации и взаимодействия — POSIX 1003;

8) планирование: приоритетное, FIFO, адаптивное, Roun Robin, preemptable;

9) средства разработки: комплект разработки C/C++, включающий компиляторы, отладчик, анализатор от QNX и независимых поставщиков (Watcom, SyBase); X Windows/Motif для QNX; комплект разработки QNX Momentics.

**Система pSOS** выпускается фирмой «Integrated Systems» (Santa Clara, USA). В феврале 2000 г. фирма приобретена компанией «Wind River Systems» (Alameda, CA, USA).

Основные характеристики:

1) категория — Host/Target;

2) архитектура — на основе микроядра;

3) стандарт — собственный;

4) ОС разработки (host) — Unix/Windows;

5) процессоры (target) — Intel 80x86, Motorola 68xxx, Intel 80960, ARM, PowerPC, MIPS;

6) линии связи host/target — последовательный канал, Ethernet;

7) размер ядра — 15 кб;

8) средства синхронизации и взаимодействия — семафоры, события, mutex и др.

9) планирование: приоритетное, preemptible;

10) средства разработки — интегрированная среда разработки C/C++/Ada.

**Система VRTX** выпускается фирмой «Ready Systems» (Sunnyvale, USA). Основные характеристики системы:

1) категория — host/target;

2) стандарт — собственный;

3) ОС разработки (host) — Unix/Windows;

4) процессоры (target) — Intel 80x86, Motorola 68xxx, Intel 80960, PowerPC;

5) линии связи host/target — последовательный канал, Ethernet, шина VME;

6) размер ядра — 16 кб;

7) средства синхронизации и взаимодействия: семафоры, события, mutex и др.;

8) планирование: приоритетное, preemptible;

9) средства разработки — MasterWorks — интегрированная среда разработки; Xray — специализированный отладчик; Simulator Xray — эмулятор ядра.

Существует также множество систем, проектируемых под конкретные модели микроконтроллеров или конкретных задач, которые обладаютт определенными преимуществами:

- наивысшей производительностью;
- наилучшим учетом особенностей оборудования;
- наибольшей компактностью.

Недостатками специализированных ОСРВ являются:

- большое время разработки;
- высокая стоимость;
- непереносимость на другие платформы.

Примерами таких систем являются ОСРВ, разработанные многими производителями электронной техники (компаниями «Sony», «Sagem» и др.), а также системы, разработанные под конкретную большую задачу, например, управление железными дорогами (TGV, France).

## Вопросы для самопроверки

1. Какова цель разработчиков ОСРВ? Какую роль выполняют стандарты на ОСРВ?
2. Какие стандарты на ОСРВ вы знаете? Опишите их.
3. На какие категории делятся ОСРВ?
4. На каких процессорах может работать ОС Linux?
5. Какие способы существуют для реализации механизма preemption?
6. В чем особенность системы RT-Linux?
7. Приведите основные аргументы «за» и «против» использования Windows NT в качестве ОСРВ.
8. Какие расширения Windows NT, поддерживающие реальное время, вы знаете?
9. Приведите классификацию ОСРВ по категориям.
10. Приведите преимущества и недостатки специализированных ОСРВ.

# 4. СИСТЕМЫ РЕАЛЬНОГО ВРЕМЕНИ: ОБЩЕЕ ПРЕДСТАВЛЕНИЕ

## 4.1. Основные характеристики ОС QNX

Операционная система QNX является мощной операционной системой, позволяющей проектировать сложные программные системы, работающие в реальном времени, как на одном-единственном компьютере, так и в локальной вычислительной сети. Встроенные средства операционной системы QNX обеспечивают поддержку многозадачного режима на одном компьютере и взаимодействие параллельно выполняемых задач на разных компьютерах, работающих в среде локальной вычислительной сети. Основным языком программирования в системе является язык С. Основная операционная среда соответствует стандартам POSIX-интерфейса. Это позволяет с небольшими доработками перенести необходимое накопленное программное обеспечение в среду операционной системы QNX для организации работы в среде распределенной обработки.

ОС QNX является сетевой, мультизадачной, многопользовательской (многотерминальной) и масштабируемой системой. С точки зрения пользовательского интерфейса и API она очень похожа на UNIX. Однако QNX — это не версия UNIX, хотя многие так считают. QNX была разработана, что называется, «с нуля» канадской фирмой «QNX Software Systems Limited» в 1989 г. по заказу Министерства обороны США [10]. Причем эта система построена на совершенно других архитектурных принципах, отличных от принципов, использованных при создании ОС UNIX.

QNX — первая коммерческая ОС, основанная на принципах построения архитектуры микроядра и обмена сообщениями. Система реализована в виде совокупности независимых, но взаимодействующих через обмен сообщениями процессов различного уровня (менеджеры и драйверы), каждый из которых реализует определенный вид сервиса. Система QNX имеет несколько важнейших преимуществ:

- ***предсказуемость,*** означающая ее применимость к задачам

жесткого реального времени. QNX является операционной системой, которая дает полную гарантию в том, что процесс с наивысшим приоритетом начнет выполняться практически немедленно и что критическое событие (например, сигнал тревоги) никогда не будет потеряно. Ни одна версия UNIX не может достичь подобного качества, поскольку нереентерабельный код ядра слишком велик. Любой системный вызов из обработчика прерывания в UNIX может привести к непредсказуемой задержке (то же самое касается Windows NT);

- ***масштабируемость*** и ***эффективность***, достигаемые оптимальным использованием ресурсов и означающие ее применимость для встроенных (embedded) систем; вы не увидите в каталоге /dev множества файлов, соответствующих ненужным драйверам. Драйверы и менеджеры можно запускать и удалять (кроме файловой системы, что очевидно) динамически, просто из командной строки. Вы можете иметь только тот сервис, который вам реально нужен, причем это не требует серьезных усилий и не создает проблем;
- ***расширяемость*** и ***надежность*** одновременно, поскольку написанный вами драйвер не нужно компилировать в ядро, рискуя вызвать нестабильность системы. Менеджеры ресурсов (сервис логического уровня) работают в кольце защиты 3, и вы можете добавлять свои, не опасаясь за систему. Драйверы работают в кольце с уровнем привилегий 1 и могут вызвать проблемы, но не фатального характера. Кроме того, их достаточно просто писать и отлаживать;
- ***быстрый сетевой протокол FLEET***, прозрачный для обмена сообщениями, автоматически обеспечивающий отказоустойчивость, балансирование нагрузки и маршрутизацию между альтернативными путями доступа;
- ***компактная графическая подсистема Photon***, построенная на тех же принципах модульности, что и сама ОС, позволяет получить полнофункциональный GUI (расширенный Motif), работающий вместе с POSIX-совместимой ОС всего в 4 Мб памяти, начиная с i80386 процессора.

Приведем основные требования, обязательное выполнение которых необходимо при создании ОСРВ. Первым обязатель-

ным требованием к архитектуре ОСРВ является ***многозадачность*** в истинном смысле этого слова. Очевидно, что варианты с псевдомногозадачностью (а точнее невытесняющей многозадачностью) типа MS Windows 3.x или Novell NetWare неприемлемы, поскольку они допускают возможность блокировки или даже полного развала системы одним неправильно работающим процессом. Для предотвращения блокировок ОСРВ должна использовать квантование времени (то есть вытесняющую многозадачность), что является вполне выполнимым требованием. Второе требование (***организация надежных вычислений)*** может быть эффективно выполнено при полном использовании возможностей процессоров Intel 80386 и старше, что предполагает работу ОС в 32-разрядном режиме процессора. Для эффективного обслуживания прерываний ОС должна использовать алгоритм диспетчеризации, обеспечивающий ***вытесняющее планирование, основанное на приоритетах***. Крайне желательны эффективная ***поддержка сетевых коммуникаций*** и наличие развитых механизмов ***взаимодействия между процессами,*** поскольку реальные технологические системы обычно управляются целым комплексом компьютеров и/или контроллеров. Весьма важно также, чтобы ОС поддерживала ***множественные потоки управления*** (не только ***мультипрограммный,*** но и ***мультизадачный режимы***), а также симметричную мультипроцессорность. И, наконец, при выполнении всех перечисленных требований ОС должна быть способна ***работать на ограниченных аппаратных ресурсах,*** поскольку одна из ее основных областей применения — это встроенные системы. К сожалению, данное требование обычно выполняется путем урезания стандартных сервисных средств.

## 4.2. Основные характеристики микроядра ОС QNX

**Основным назначением любой ОС является управление ресурсами компьютера**. Все процессы в системе (планирование выполнения прикладных программ, запись файлов на диск,

пересылка данных по сети и т. д.) должны выполняться единообразно и бесконфликтно.

Некоторые прикладные системы могут предъявлять повышенные требования к управлению ресурсами и планированию процессов. Например, работа приложений реального времени зависит от того, как операционная система управляет большим количеством событий, возникающих за конечные интервалы времени. Чем больше функций берет на себя ОС, тем более свободно «чувствуют» себя эти приложения при возникновении конфликтных ситуаций.

Для приложений, работающих в режиме реального времени, QNX является идеальной операционной системой. Она удовлетворяет всем основным требованиям, предъявляемым к системам реального времени: в ней реализован многозадачный режим, приоритетноуправляемое планирование и быстрое переключение контекста.

Кроме того, система QNX обладает большой гибкостью. Разработчики могут легко адаптировать ее под требования своих приложений. Настройка системы QNX может быть выполнена от минимальной (ядро и несколько небольших модулей) до полной сетевой конфигурации (обслуживание сотен пользователей), позволяя использовать в каждом конкретном случае только те ресурсы, которые необходимы.

*Уникальная эффективность, модульность и простота системы QNX определяются архитектурой ядра и взаимодействием между процессами посредством сообщений.*

**Архитектура ядра системы QNX** представлена небольшим микроядром и набором взаимодействующих процессов. Система не имеет иерархической структуры, ее организация скорее напоминает спортивную команду, в которой игроки (процессы), имеющие равную значимость, взаимодействуют друг с другом и со своим ведущим игроком (ядром) [11].

Ядро является «сердцем» любой операционной системы. В некоторых системах на ядро возложено такое количество функций, что, по сути дела, оно само является полной операционной системой. В системе QNX ядро является действительно ядром. Прежде всего, как и подобает ядру ОС реального времени, оно имеет небольшой размер — менее 8 кб.

***Ядро операционной системы QNX*** выполняет следующие функции:

- связь между процессами (IPC — Inter-Process Communication);
- сетевое взаимодействие нижнего уровня. Ядро передает все сообщения, предназначенные процессам на другом узле;
- планирование процессов. Планировщик ядра определяет, какой процесс будет выполняться следующим;
- первичную обработку прерываний. Все прерывания и сбои аппаратного обеспечения сначала обрабатываются в ядре, а затем передаются соответствующему драйверу или системному администратору.

В отличие от процессов, непосредственно ядро никогда не планируется к выполнению. Управление передается ядру только в результате прямого вызова ядра либо из процесса, либо по аппаратному прерыванию.

Все функции, выполняемые операционной системой QNX, за исключением функций ядра, реализуются стандартными процессами. В типичной конфигурации системы QNX имеются следующие системные процессы [11]:

- администратор процессов (Proc);
- администратор файловой системы (Fsys);
- администратор устройств (Dev);
- сетевой администратор (Net).

**Администратор процессов** практически ничем не отличаются от любого процесса пользователя: у системных процессов нет специального или скрытого интерфейса, недоступного процессу пользователя.

Именно такая архитектура обеспечивает системе QNX неограниченную расширяемость. Поскольку большинство функций QNX выполняется стандартными системными процессами, то расширить операционную систему совсем не сложно: достаточно написать и включить в систему программу, реализующую новую функцию ОС.

Действительно, грань между операционной системой и прикладными программами весьма условна. Единственным принципиальным отличием системных процессов от прикладных

является то, что системные процессы управляют ресурсами системы, предоставляя их прикладным процессам.

Работу по **администрированию файловой системы** рассмотрим на примере сервера базы данных. Сервер базы данных должен выполнять функции, аналогичные функциям администратора файловой системы, который получает запросы (сообщения) на открытие файлов и чтение или запись данных. Несмотря на то что запросы к серверу базы данных могут быть достаточно сложными, и в том и в другом случае формируется набор примитивов (посредством сообщений), в результате чего обеспечивается доступ к системному ресурсу. В обоих случаях речь идет о процессах, которые могут быть описаны конечным пользователем, и выполняться по необходимости. Таким образом, сервер базы данных можно рассматривать как системный процесс в одном случае и как прикладной — в другом. Фактически нет никакой разницы. Важно отметить, что в системе QNX подобные процессы, включаются без модификаций других компонентов операционной системы.

**Администратор устройств** организует работу драйверов устройств. ***Драйверы устройств*** — это процессы, которые избавляют операционную систему от необходимости иметь дело со всеми особенностями работы аппаратного обеспечения. Поскольку драйверы выполняются как стандартные процессы, то добавление нового драйвера в систему QNX не влияет на работу других компонентов операционной системы. Единственное изменение, которое происходит в среде QNX, — это запуск нового драйвера. Драйвер может быть оформлен либо как дополнение к системному процессу, либо, для сохранения его «индивидуальности», в качестве стандартного процесса.

Сетевой администратор отвечает за связь драйверов сетевых адаптеров с администраторами протоколов. Благодаря его использованию операционная система выполняет передачу сообщений между ядрами.

## 4.3. Связь между процессами

### 4.3.1. Связь между процессами посредством сообщений

***Механизм передачи межпроцессных сообщений*** (пересылка сообщений между процессами) является одной из важнейших частей операционной системы, так как все общение между процессами, в том числе и системными, происходит через сообщения. Сообщение в QNX — это последовательность байтов произвольной длины (0–65535 байтов) произвольного формата. Протокол обмена сообщениями выглядит таким образом: задача блокируется для ожидания сообщения. Другая же задача посылает первой сообщение и при этом блокируется сама, ожидая ответа. Первая задача разблокируется, обрабатывает сообщение и отвечает, разблокируя при этом вторую задачу.

Сообщения и ответы, пересылаемые между процессами при их взаимодействии, находятся в теле отправляющего их процесса до того момента, когда они могут быть приняты. Это значит, что, с одной стороны, уменьшается вероятность повреждения сообщения в процессе передачи, а с другой — уменьшается объем оперативной памяти, необходимый для работы ядра. Кроме того, уменьшается число пересылок из памяти в память, что разгружает процессор. Особенностью процесса передачи сообщений является то, что в сети, состоящей из нескольких компьютеров под управлением QNX, сообщения могут прозрачно передаваться процессам, выполняющимся на любом из узлов. Определены в QNX еще и два дополнительных метода передачи сообщений — метод представителей (Proxy) и метод сигналов (Signal) (см. подробно пп. 4.3.2 и 4.3.3).

Для прямой связи друг с другом взаимодействующие процессы используют следующие функции языка Си:

`Send()` — посылка сообщений;

`Receive()` — прием сообщений;

`Reply()` — ответа процессу, пославшему сообщение.

Эти функции могут использоваться локально или по всей сети.

***Обратите внимание!*** Для прямой связи процессов друг с другом необязательно использование функций `Send()`, `Receive()` и `Reply()`. Система библиотечных функций QNX надстроена над системой обмена сообщениями, поэтому процессы могут использовать передачу сообщений косвенно, применяя стандартные сервисные средства, например программные каналы (pipe).

Приведем пример (рис. 4.1), который иллюстрирует использование функций `Send()`, `Receive()` и `Reply()` при взаимодействии двух процессов — **A** и **B**:

- процесс **A** посылает сообщение процессу **B**, выдав ядру запрос `Send()`. С этого момента процесс **A** становится SEND-блокированным до тех пор, пока процесс **B** не выдаст `Receive()`, подтверждая получение сообщения;
- процесс **B** выдает `Receive()` процессу **A**, ожидающему сообщения. Процесс **A** изменяет свое состояние на REPLY-блокированное. Поскольку от процесса **B** ожидается сообщение, он не блокируется.

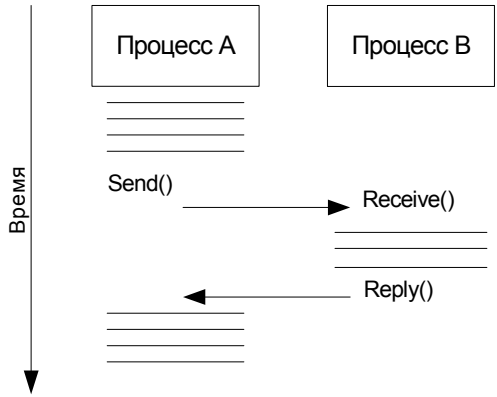


Рис. 4.1. Обмен сообщениями между процессами

***Обратите внимание!*** Если бы процесс **B** выполнил функцию `Receive()` до отправления ему сообщения, он оставался бы RECEIVE-блокированным до момента приема сообщения.

В этом случае процесс **A** (отправитель) перешел бы сразу в REPLY-блокированное состояние после отправления сообщения процессу **B**;

• процесс **B** выполняет необходимую обработку, определяемую полученным от процесса **A** сообщением, и выдает `Reply()`. Процесс **A** получает ответное сообщение и разблокируется. Процесс **B** также разблокируется. Последовательность выполнения процессов зависит от их относительных приоритетов.

Передача сообщений служит не только для обмена данными между процессами, но, кроме того, является средством синхронизации выполнения нескольких взаимодействующих процессов.

Обратимся снова к приведенному выше рис. 4.1. После того как процесс **A** выдаст запрос `Send()`, он не сможет выполняться до тех пор, пока не получит ответа на переданное им сообщение. Это служит гарантией того, что обработка данных, выполняемая процессом **B** для процесса **A,** завершится до того, как процесс **A** сможет продолжить свою работу. В свою очередь процесс **B** после выдачи запроса `Receive()` может продолжать свою работу до поступления другого сообщения.

Рассмотрим более подробно функции `Send()`, `Receive()` и `Reply()`.

**Использование функции `Send()`.** Предположим, что процесс **A** выдает запрос на передачу сообщения процессу **B**. Запрос оформляется вызовом функции `Send()`:

```
Send (pid, smsg, rmsg, smsg_bn, rmsg_len);
```

Функция `Send()` имеет следующие аргументы:

`pid` — идентификатор процесса-получателя сообщения (т. е. процесса **B**), посредством которого процесс опознается операционной системой и другими процессами;

`smsg` — буфер сообщения (т. е. посылаемого сообщения);

`rmsg` — буфер ответа, в который помещается ответ процесса B;

`smsg_len` — длина посылаемого сообщения;

`rmsg_len` — максимальная длина ответа, который должен получить процесс **A**.

***Обратите внимание!*** В сообщении будет передано не более чем smsg_len байтов и принято в ответе не более чем rmsg_len байтов — это служит гарантией того, что буферы никогда не будут переполнены.

**Использование функции `Receive()`.** Процесс **B** может принять запрос `Send()`, выданный процессом **A**, с помощью функции `Receive()`:

```
pid = Receive (0, msg, msg_len);
```

Функция `Receive()` имеет следующие аргументы:

`pid` — идентификатор процесса, пославшего сообщение (т. е. процесса **A**);

`0` (ноль) — указание на то, что процесс **B** готов принять сообщение от любого процесса;

`msg` — буфер, в который будет принято сообщение;

`msg_len` — максимальное количество байтов данных, которое может поместиться в приемном буфере.

В случае, если значения smsg_len в функции `Send()` и msg_len в функции `Receive()` различаются, количество передаваемых данных будет определяться наименьшим из них.

**Использование функции `Reply()`.** После успешного приема сообщения от процесса **A** процесс **B** должен ответить ему, используя функцию `Reply()`:

```
Reply (pid, reply, reply_len)
```

Функция `Reply()` имеет следующие аргументы:

`pid` — идентификатор процесса, которому направляется ответ (т.е. процесса A);

`reply` — буфер ответа;

`reply_len` — длина сообщения, передаваемого в ответе.

Если значения `reply_len` в функции `Reply()` и `rmsg_len` в функции `Send()` различаются, то количество передаваемых данных определяется наименьшим из них.

Пример передачи сообщений, который мы только что рассмотрели, иллюстрирует наиболее типичный способ передачи, при котором обслуживающий процесс находится в RECEIVE-блокированом состоянии, ожидая запроса от другого процесса

на выполнение какой-либо работы. Этот способ передачи сообщений называется ***Send-управляемым***, при котором процесс, требующий обслуживания, инициирует работу, посылая сообщение, а обслуживающий процесс завершает работу, выдавая ответ на принятое сообщение.

Существует и другой, менее распространенный, чем Send-управляемый, но в отдельных случаях более предпочтительный способ передачи сообщений, а именно ***Reply-управляемый***, при котором работа инициируется функцией `Reply()`. В соответствии с этим способом «рабочий» процесс посылает сообщение обслуживающему процессу, указывая на то, что он готов к работе. Обслуживающий процесс фиксирует, что «рабочий» процесс послал ему сообщение, но не отвечает ему немедленно. Через некоторое время обслуживающий процесс может ответить «рабочему» процессу. «Рабочий» процесс выполняет свою работу, а затем, завершив ее, посылает обслуживающему процессу сообщение, содержащее результаты.

Данные, передаваемые в сообщении, находятся в процессе-отправителе до момента, когда получатель будет готов к обработке сообщения. Сообщение не копируется в ядро. Это обеспечивает сохранность данных, так как процесс-отправитель остается SEND-блокированным и не может случайным образом модифицировать данные сообщения.

При выдаче запроса `Reply()` данные, содержащиеся в ответном сообщении, передаются от отвечающего процесса REPLY-блокированному процессу за одну операцию. Функция `Reply()` не блокирует отвечающий процесс, так как REPLY-блокированный процесс разблокируется сразу после того, как данные скопируются в его адресное пространство.

Процессу-отправителю нет необходимости «знать» что-либо о состоянии процесса-получателя, которому он посылает сообщение. В том случае, если процесс-получатель будет не готов к приему сообщения, то процесс-отправитель после отправления сообщения просто перейдет в SEND-блокированное состояние. При необходимости любой процесс может посылать сообщение нулевой длины, ответ нулевой длины либо то и другое.

С точки зрения разработчика, выдача запроса `Send()` обслуживающему процессу — это практически то же самое, что и обращение к библиотеке подпрограмм. В обоих случаях разработчик формирует некоторые наборы данных, а затем выдает `Send()` или обращается к библиотечной функции. После этого между двумя определенными точками программы `Receive()` и `Reply()` в одном случае, либо между входом функции и оператором завершения функции return в другом, управление передается сервисным программам, при этом ваша программа ожидает завершения их выполнения. После того как сервисные программы отработали, ваша программа «знает», где находятся результаты их работы, и может затем анализировать коды ошибок, обрабатывать результаты и т. д.

Несмотря на это кажущееся сходство, процесс передачи сообщения намного сложнее обычного вызова библиотечной функции. Например, `Send()` может по сети обратиться к другой машине, где действительно будет выполняться сервисная программа. Кроме того, может быть организована параллельная обработка данных без создания нового процесса. Обслуживающий процесс выполняет функцию `Reply()`, позволяя вызывающему процессу продолжать работу, и затем осуществляет дальнейшие действия.

Несколько процессов могут посылать сообщения одному процессу. Обычно процесс-получатель принимает сообщения в порядке их поступления, однако может быть установлен режим приема сообщений в порядке приоритетов процессов-отправителей.

В системе QNX имеются функции, предоставляющие **дополнительные возможности передачи сообщений**, а именно:

- условный прием сообщений;
- чтение и запись части сообщения;
- передача составных сообщений.

Обычно для приема сообщения используется функция `Receive()`. Этот способ приема сообщений в большинстве случаев является наиболее предпочтительным.

Однако иногда процессу требуется предварительно «знать», было ли ему послано сообщение, чтобы не ожидать поступления сообщения в RECEIVE-блокированном состоянии. Например,

процессу требуется обслуживать несколько высокоскоростных устройств, не способных генерировать прерывания, и, кроме того, процесс должен отвечать на сообщения, поступающие от других процессов. В этом случае используется функция `Creceive()`, которая считывает сообщение, если оно становится доступным, или немедленно возвращает управление процессу, если нет ни одного отправленного сообщения.

***Внимание!*** По возможности следует избегать использования функции `Creceive()`, так как она позволяет процессу непрерывно загружать процессор на соответствующем приоритетном уровне.

В некоторых случаях предпочтительнее считывать или записывать только часть сообщения, для того чтобы использовать буфер, уже выделенный для сообщения, и не заводить рабочий буфер. Например, администратор ввода/вывода может принимать для записи сообщения, состоящие из заголовка фиксированной длины и переменного количества данных. Заголовок содержит значение количества байтов данных (от 0 до 64 кб). Администратор ввода/вывода может принимать сначала только заголовок, а затем, используя функцию `Readmsg()`, считывать данные переменной длины в соответствующий буфер. Если количество посылаемых данных превышает размер буфера, администратор ввода/вывода может вызывать функцию `Readmsg()` несколько раз, передавая данные по мере освобождения буфера. Аналогично, функцию `Writemsg()` можно использовать для сбора и копирования данных в буфер отправителя по мере его освобождения, снижая таким образом требования к размеру внутреннего буфера администратора ввода/вывода.

До сих пор мы рассматривали сообщения как единый пакет байтов. Однако, как правило, сообщения состоят из нескольких дискретных частей. Например, сообщение может иметь заголовок фиксированной длины, за которым следуют данные переменной длины. Для того чтобы части сообщения эффективно передавались и принимались без копирования во временный рабочий буфер, составное сообщение может формироваться в нескольких раздельных буферах. Этот метод позволяет администрат-

раторам ввода/вывода системы QNX (Dev и Fsys) обеспечивать высокую производительность.

Для работы с составными сообщениями используются следующие функции:

```
Creceivemx()
Readmsgmx()
Receivemx()
Replymx()
Sendmx()
Writemsgmx()
```

Все сообщения в системе QNX начинаются с 16-битового слова, которое называется кодом сообщения. Системные процессы QNX используют следующие коды сообщений:

0X0000–0X00FF — сообщения Администратора процессов;

0X0100–0X01FF — сообщения ввода/вывода (для всех обслуживающих программ);

0X0200–0X02FF — сообщения Администратора файловой системы;

0X0300–0X03FF — сообщения Администратора устройств;

0X0400–0X04FF — сообщения Сетевого администратора;

0X0500–0X0FFF — сообщения, зарезервированные для системных процессов, которые могут появиться в будущем.

### 4.3.2.Связь между процессами посредством Proxy

Proxy представляет собой форму неблокирующей передачи сообщений, специально предназначенную для оповещения о событиях, при которых процесс-отправитель не нуждается во взаимодействии с процессом-получателем. Единственной функцией proxy является посылка фиксированного сообщения процессу, создавшему proxy. Так же как и сообщения, proxy работают по всей сети. Благодаря использованию proxy процесс или обработчик прерываний могут послать сообщение другому процессу, не блокируясь и не ожидая ответа. Ниже приведены некоторые примеры использования proxy:

• процесс оповещает другой процесс о наступлении некоторого события, не желая при этом оставаться SEND-блокиро-

ванным до тех пор, пока получатель не выдаст `Receive()` и `Reply()`;

• процесс посылает данные другому процессу, но не требует ни ответа, ни другого подтверждения о том, что получатель принял сообщение;

• обработчик прерываний оповещает процесс о том, что некоторые данные доступны для обработки.

Proxy-процессы создаются с помощью функции

`qnx_proxy_at-tach()`.

Любой процесс или обработчик прерываний, которому известен идентификатор proxy, может воспользоваться функцией `Trigger()`, для того чтобы выдать заранее определенное сообщение. Запросами `Trigger()` управляет ядро.

Процесс Proxy может быть запущен несколько раз: выдача сообщения происходит каждый раз при его запуске. Процесс Proxy может накопить в очереди для выдачи до 65535 сообщений.

### 4.3.3. Связь между процессами посредством сигналов

Связь посредством сигналов представляет собой традиционную форму асинхронного взаимодействия, используемую в различных операционных системах. В системе QNX поддерживается большой набор POSIX-совместимых сигналов, специальные QNX-сигналы, а также исторически сложившиеся сигналы, используемые в некоторых версиях системы UNIX.

Сигнал выдается процессу при наступлении некоторого заранее определенного для данного сигнала события. Процесс может выдать сигнал самому себе. Если вы хотите сгенерировать сигнал из интерпретатора Shell, используйте утилиты kill() или slay(). Если вы хотите сгенерировать сигнал из процесса, используйте утилиты kill() или raise().

В зависимости от того, каким образом был определен способ обработки сигнала, возможны три варианта его приема:

1) если процессу не предписано выполнять каких-либо специальных действий по обработке сигнала, то по умолчанию поступление сигнала прекращает выполнение процесса;

2) процесс может проигнорировать сигнал. В этом случае выдача сигнала не влияет на работу процесса (***обратите внимание*** на то, что сигналы `SIGCONT`, `SIGKILL` и `SIGSTOP` не могут быть проигнорированы при обычных условиях);

3) процесс может иметь обработчик сигнала, которому передается управление при поступлении сигнала. В этом случае говорят, что процесс может «ловить» сигнал. Фактически такой процесс выполняет обработку программного прерывания. Данные с сигналом не передаются.

Интервал времени между генерацией и выдачей сигнала называется задержкой. В этот момент времени для одного процесса могут быть задержаны несколько разных сигналов. Сигналы выдаются процессу тогда, когда планировщик ядра переводит процесс в состояние готовности к выполнению. Порядок поступления задержанных сигналов не определен.

Для задания способа обработки сигнала следует воспользоваться функцией ANSI C `signal()` или функцией POSIX `sigaction()`. Функция `sigaction()` предоставляет больше возможностей по управлению средой обработки сигнала.

Способ обработки сигнала можно изменить в любое время. Если установить обработчику режим игнорирования сигналов, то все задержанные сигналы будут немедленно отменены.

Отметим некоторые особенности работы процессов, которые «ловят» сигналы с помощью обработчика сигналов.

Обработчик сигналов аналогичен программному прерыванию. Он выполняется асинхронно с другими программами процесса. Следовательно, обработчик сигналов может быть запущен во время выполнения любой функции в программе (включая библиотечные функции).

Если процессу не требуется возврата управления от обработчика сигналов в прерванную точку, то в этом случае в обработчике сигналов могут быть использованы функции `siglongjmp()` или `longjmp()`. Причем `siglongjmp()` предпочтительнее, так как в случае использования `longjmp()` сигнал остается блокированным.

Иногда может потребоваться временная задержка выдачи сигнала без изменения при этом способа его обработки. В сис-

теме QNX имеется набор функций, позволяющих блокировать выдачу сигналов. После разблокировки сигнал выдается программе.

Во время работы обработчика сигналов QNX автоматически блокирует обрабатываемый сигнал. Это означает, что не требуется организовывать вложенные вызовы обработчика сигналов. Каждый вызов обработчика сигналов не прерывается остальными сигналами данного типа. При нормальном возврате управления от обработчика сигнал автоматически разблокируется.

***Внимание!*** В некоторых версиях системы UNIX работа с обработчиком сигналов организована некорректно, так как в них не предусмотрена блокировка сигналов. В результате в некоторых приложениях, работающих под управлением UNIX, используется функция `signal()` внутри обработчика прерываний с целью «перевзвода» обработчика. В этом случае может возникнуть одна из двух аварийных ситуаций. Во-первых, если другой сигнал поступает во время работы обработчика, но вызова функции `signal()` еще не было, то программа будет снята с обработки. Во-вторых, если сигнал поступает сразу же после вызова обработчиком функции `signal()`, то обработчик будет запускаться рекурсивно. В QNX выполняется блокировка сигналов, поэтому указанные выше проблемы не могут возникнуть. Нет необходимости вызывать `signal()` из обработчика. Если требуется выйти из любой точки обработчика, то следует воспользоваться функцией `siglongjmp()`.

Существует важная взаимосвязь между сигналами и сообщениями. Если при генерации сигнала ваш процесс окажется SEND-блокированным или RECEIVE-блокированным (причем имеется обработчик сигналов), то будут выполняться следующие действия:

1) разблокировка процесса;
2) обработка сигнала;
3) возврат управление с кодом ошибки функциями `Send()` или `Receive()`.

Если процесс был SEND-блокированным, то проблемы не возникает, так как получатель не получит сообщение. Но если

процесс был REPLY-блокированным, то неизвестно, было обработано отправленное сообщение или нет, а следовательно, неизвестно, нужно ли еще раз выдавать `Send()`.

Процесс, выполняющий функции сервера (т. е. принимающий сообщения), может запрашивать уведомления о том, когда обслуживаемый процесс выдаст сигнал, находясь в REPLY-блокированном состоянии. В этом случае обслуживаемый процесс становится SIGNAL-блокированным с задержанным сигналом и обслуживающий процесс принимает специальное сообщение, описывающее тип сигнала. Обслуживающий процесс может выбрать одно из следующих действий:

- нормально завершить первоначальный запрос: отправитель будет уведомлен о том, что сообщение было обработано надлежащим образом;
- освободить все закрепленные ресурсы и возвратить управление с кодом ошибки, указывающим на то, что процесс был разблокирован сигналом: отправитель получит чистый код ошибки.

Когда обслуживающий процесс сообщает другому процессу, что он SIGNAL-блокирован, сигнал выдается немедленно после возврата управления функцией `Send()`.

### 4.3.4. Примеры связи между процессами посредствам обмена сообщениями

**Программа КЛИЕНТ**

***Передача сообщения со стороны клиента*** осуществляется применением какой-либо функции из семейства `Send()`. Рассмотрим это на примере простейшей из них — `MsgSend()`:

```
include <sys/neutrino.h>
int MsgSend(int cold, const void *smsg, int
sbytes,
            void *rmsg, int rbytes);
```

***Для создания соединения*** между процессом и каналом используется функция `ConnectAttach()`, в параметрах которой задаются идентификатор процесса и номер канала:

```
#include <sys/neutrino.h>
int ConnectAttach( uint32_t nd,
                   pid_t pid,
                   int chid,
                   unsigned index,
                   int flags );
```

**Пример программы КЛИЕНТ**

Передадим сообщение процессу с идентификатором 77 по каналу 1:

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

char *smsg = "Это буфер вывода";
char rmsg[200];
int coid;
int main(void);
{
// 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 MsgSendNn");
perror(NULL);
exit(EXIT FAILURE);
```

```
}
if (strlen(rmsg) > 0)
{
printf("Процесс с ID 77 возвратил \"%s\"\n",
            rmsg);
}
exit(0);
}
```

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 возвратить 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).

**Программа СЕРВЕР**

***Создание канала*** необходимо для присоединения клиента к серверу посредством функции `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);
```

На данном этапе будем использовать для параметра flags 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. Сервер отрабатывает схему сообщений обмена в два этапа — этап «прием» (receive) и этап «ответ» (reply).

```
    #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);
```

Для каждой буферной передачи указываются два размера (в случае запроса от 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 примере программы КЛИЕНТ размер буфера функции `MsgSend()` совпадал с длиной строки сообщения.

**Пример программы СЕРВЕР**

```
#include <sys/neutrino.h>
#include <sys/iomsg.h>
#include <errno.h>
#include <stdio.h>
#include <process.h>
void main(void)
{
int  rcvid;
int   chid;
char  message [512];

// 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 байтов).

Наш клиент (представленный выше) передал только 28 байт (длина строки).

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

Если один процесс создает другой процесс, тогда это просто ситуация, при которой 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 идентификато-

ра родительского процесса и использовать некоторый «известный» идентификатор канала:

```
#include <process.h>
pid_t getpid( void );
```

Выясним вопрос, как сервер объявляет о своем местонахождении. Существует множество способов сделать это. Рассмотрим только три из них в порядке возрастания «элегантности»:

1) открыть файл с известным именем и сохранить в нем ND/PID/CHID. Такой метод является традиционным для серверов UNIX. В этом случае сервер открывает файл (например, /etc/httpd.pid), записывает туда свой идентификатор процесса в виде строки ASCII и предполагается, что клиенты откроют этот файл и 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 своей природе является очень редким;

3) занять часть пространства имен путей и стать администратором ресурсов.

## 4.4. Сетевое взаимодействие

В системе QNX приложение может взаимодействовать с процессом, выполняющимся на другом компьютере сети, так же как с процессом, выполняющимся на своем компьютере. В самом деле, с точки зрения приложения нет никакой разницы между локальными и удаленными ресурсами. Такая высокая степень прозрачности обеспечивается благодаря использованию виртуальных каналов, которые являются путями, по которым *Сетевой администратор* передает сообщения и сигналы по всей сети. **Виртуальные каналы (ВК)** способствуют эффективному использованию ресурсов во всей сети QNX по нескольким причинам:

- при создании виртуального канала имеется возможность задать работу с сообщениями определенной длины: это означа-

ет, что вы можете распределить ресурсы для обработки сообщения. Тем не менее, если потребуется послать сообщение, длина которого превышает максимально заданную, виртуальный канал автоматически изменит установленный максимальный размер буфера в соответствии с длиной передаваемого сообщения;

- если два процесса, находящиеся на разных узлах, взаимодействуют между собой более чем через один виртуальный канал, виртуальные каналы разделяются во времени, так как между процессами существует только один реальный виртуальный канал. Эта ситуация часто возникает, когда процесс обращается к нескольким файлам удаленной файловой системы;
- если процесс подключается к существующему разделенному виртуальному каналу и запрашивает размер буфера, больший, чем тот, который используется в данное время, размер буфера автоматически увеличивается;
- когда процесс завершается, все связанные с ним виртуальные каналы освобождаются.

Процесс-отправитель отвечает за установку виртуального канала между собой и процессом, с которым устанавливается связь. Для этого процесс-отправитель обычно вызывает функцию `qnx_vc_attach()`. При этом на каждом конце канала создается **виртуальный процесс** с идентификатором VID. Для каждого процесса на обоих концах виртуального канала VID представляет собой идентификатор удаленного процесса, с которым устанавливается связь. Процессы связываются друг с другом посредством VID.

Каждый VID обеспечивает соединение, которое содержит следующую информацию:

- локальный pid;
- удаленный pid;
- удаленный nid (идентификатор узла);
- удаленный vid.

Вряд ли вам придется работать с виртуальным каналом напрямую. Если приложению требуется, например, получить доступ к удаленному ресурсу ввода/вывода, то виртуальный канал создается вызываемой библиотечной функцией `open()`. Приложения непосредственно не участвуют в создании или исполь-

зовании виртуального канала. Если приложение определяет нахождение обслуживающего его процесса с помощью функции `qnx_name_locate()`, то виртуальный канал создается автоматически при вызове функции. Для приложения виртуальный канал просто отождествляется с PID.

Существует несколько причин, по которым процесс не может осуществлять связь по установленным виртуальным каналам, а именно:

- произошло отключение питания компьютера, на котором выполняется процесс;
- был отсоединен кабель сети от компьютера;
- был завершен удаленный процесс, с которым установлена связь.

Любая из этих причин может препятствовать передаче сообщений по виртуальному каналу. Необходимо фиксировать эти ситуации для выполнения приложением необходимых действий с целью корректного завершения работы, в противном случае отдельные ресурсы могут оказаться постоянно занятыми.

На каждом узле Администратор процессов проверяет целостность виртуального канала. Это делается следующим образом:

1) каждый раз при успешной передаче сообщения по виртуальному каналу обновляется временная метка, связанная с данным виртуальным каналом, для фиксации времени последней активности;

2) через интервалы времени, устанавливаемые при инсталляции, Администратор процессов просматривает каждый виртуальный канал. В том случае, если в виртуальном канале нет активности, Администратор процессов посылает сетевой пакет проверки целостности канала Администратору процессов другого узла;

3) в том случае, если ответ не получен или зафиксирован сбой, виртуальный канал помечается как сбойный. Далее предпринимается ряд действий, определенных при инсталляции для восстановления связи;

4) если попытки восстановления закончились безуспешно, виртуальный канал «отключается». Все процессы, блокированные на данном канале, переходят в состояние ГОТОВ (READY).

Процессы анализируют возвращаемый код сбоя виртуального канала.

Для управления параметрами, связанными с проверкой целостности виртуального канала, используется утилита `netpoll`.

## 4.5. Планирование процессов

Планирование процессов осуществляется планировщиком ядра, который запускается в следующих случаях:

- после разблокировки процесса;
- по истечении временного кванта для выполняющегося процесса;
- после выгрузки выполняющегося процесса.

В системе QNX каждому процессу присваивается приоритет. Планировщик выбирает для выполнения процессы, находящиеся в состоянии ГОТОВ, в соответствии с их приоритетами. Центральный процессор может использовать только процесс, находящийся в состоянии ГОТОВ.

Процессам присваиваются приоритеты в диапазоне от 0 (низший) до 31 (высший). По умолчанию процесс наследует приоритет от породившего его процесса, обычно он равен 10 для приложений, запускаемых из интерпретатора Shell. Для выполнения выбирается процесс, имеющий наивысший приоритет.

Если вы хотите определить приоритет процесса, используйте функцию `getprio()`.

Если вы хотите задать приоритет процессу, используйте функцию `setprio()`.

Для удовлетворения потребностей разных приложений в системе QNX реализованы три метода планирования:

1) планирование по принципу простой очереди (первым пришел —первым обслужен);

2) круговой метод планирования;

3) адаптивное планирование.

Каждый процесс в системе может выполняться, используя любой из этих методов. Они эффективны применительно к одному процессу, а не ко всем процессам на узле.

Запомните, что данные методы планирования используются только тогда, когда два или более процессов, разделяющих один и тот же приоритет, находятся в состоянии ГОТОВ (т. е. процессы непосредственно конкурируют друг с другом). Если в состояние ГОТОВ переходит процесс, имеющий более высокий приоритет, он немедленно выгружает все процессы с меньшим приоритетом.

Если вы хотите определить метод планирования для процесса, используйте функцию `getscheduler()`.

Если вы хотите установить метод планирования для процесса, используйте функцию `setscheduler()`.

В системе QNX в большинстве случаев взаимодействия между процессами используется модель «клиент-сервер». Серверы (обслуживающие процессы) выполняют некоторые сервисные функции, а клиенты (обслуживаемые процессы) посылают сообщения к этим серверам, запрашивая обслуживание. В общем случае серверы более «надежны и долговечны», чем клиенты.

Обычно количество клиентов превосходит количество серверов. В результате сервер, как правило, выполняется с приоритетом, превосходящим приоритеты всех своих клиентов. При этом может использоваться любой из описанных выше методов планирования, однако круговой метод предпочтительнее.

Если низкоприоритетный клиент посылает сообщение серверу, то по умолчанию его запрос будет обрабатываться сервером с более высоким приоритетом. Это косвенно повышает приоритет клиента, так как именно запрос клиента заставил работать сервер. Если сервер работает на коротком отрезке времени, то этим можно пренебречь. Если же сервер работает более длительное время, то низкоприоритетный клиент может негативно влиять на другие процессы, имеющие приоритеты выше, чем у клиента, но ниже, чем у сервера.

Для разрешения этой проблемы серверу можно устанавливать приоритет, соответствующий приоритету клиента, который послал ему сообщение. Когда сервер получает сообщение, его приоритет становится равным приоритету клиента. Обратите внимание на то, что меняется только приоритет, а метод планирования остается тем же. Если при работе сервера поступает

другое сообщение, то приоритет сервера увеличивается в том случае, если приоритет нового клиента окажется выше приоритета сервера. В результате новый клиент «выравнивает» приоритет сервера под свой, позволяя ему закончить выполнение текущего запроса и перейти к выполнению вновь поступившего. В противном случае приоритет нового клиента понизился бы, так как он заблокировался бы на низкоприоритетном сервере.

Если вы выбираете для вашего сервера приоритеты, управляемые клиентом, то вам следует также позаботиться и о том, чтобы сообщения доставлялись в порядке приоритетов (а не в порядке времени поступления).

Для установки приоритета, управляемого клиентом, воспользуйтесь функцией `qnx_pflags()`.

## 4.6. Первичная обработка прерываний

Как бы нам этого не хотелось, но компьютер не может иметь бесконечное быстродействие. В системе реального времени крайне важно, использовать все циклы работы центрального процессора. Также важно минимизировать интервал времени между возникновением внешнего события и фактическим началом выполнения программы, реализующей ответную реакцию на это событие. Это время называется задержкой или временем ожидания (latency). В системе QNX можно определить несколько типов задержки: задержки прерывания, задержки планирования

**Задержка прерывания** — это интервал времени между приемом аппаратного прерывания и началом выполнения первой команды обработчика данного прерывания. В системе QNX все прерывания открыты всегда, поэтому задержка прерывания обычно незначительна. Но некоторые критические программы требуют, чтобы на время их выполнения прерывания были закрыты. Максимальное время закрытия прерывания обычно определяет худший случай задержки прерывания; следует отметить, что в системе QNX это время очень мало.

На рис. 4.2 представлена диаграмма обработки аппаратного прерывания соответствующим обработчиком прерываний. Об-

работчик прерываний либо просто возвращает управление процессу, либо возвращает управление и вызывает «срабатывание» proxy. Времена обработки для разных процессоров различны.

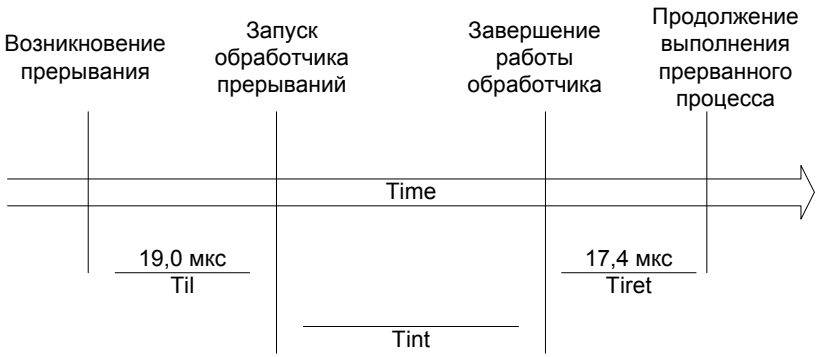


Рис. 4.2. Диаграмма обработки аппаратного прерывания: Til — время задержки прерывания; Tint — время обработки прерывания; Tiret — время завершения прерывания

На диаграмме (см. рис. 4.2), задержка прерывания Til представляет собой минимальную задержку для случая, когда во время возникновения прерывания все прерывания были открыты. В худшем случае время задержки составляет сумму минимального времени и наибольшего времени работы процесса QNX, когда прерывания закрыты.

В некоторых случаях низкоприоритетный обработчик аппаратных прерываний должен планировать выполнение высокоприоритетных процессов. В этом случае обработчик прерываний возвращает управление и вызывает срабатывание proxy. Это и есть вторая форма задержки — задержка планирования.

**Задержка планирования** — это время между завершением работы обработчика прерываний и началом выполнения первой команды управляющего процесса. Обычно это интервал времени, который требуется для сохранения контекста процесса, выполняющегося в данный момент времени, и восстановления контекста управляющего процесса. Несмотря на то что это время больше задержки прерывания, оно также остается небольшим в системе QNX.

На рис. 4.3 представлена диаграмма задержки планирования. Обработчик прерываний завершает работу и инициирует срабатывание proxy. Времена обработки для разных процессоров различны.

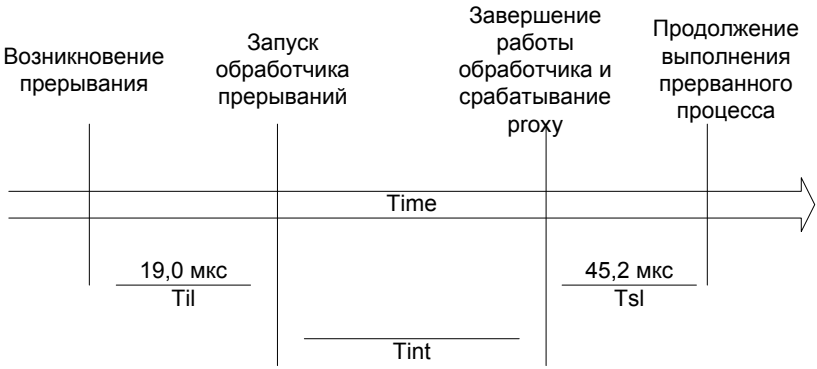


Рис. 4.3. Диаграмма задержки планирования: Til — время задержки прерывания; Tint — время обработки прерывания; Tsl — время задержки планирования

Важно заметить, что большинство обработчиков прерываний завершают работу без инициирования «срабатывания» proxy. В большинстве случаев обработчик прерываний сам справляется со всеми аппаратными событиями. Выдача proxy для подключения управляющего процесса более высокого уровня происходит только при возникновении особых событий. Например, обработчик прерываний драйвера устройства с последовательным интерфейсом, передающий один байт данных аппаратуре, должен на каждое принятое прерывание для передачи запустить высокоуровневый процесс Dev только в том случае, если выходной буфер в итоге окажется пустым.

**Вложенные прерывания.** Поскольку архитектура микрокомпьютера позволяет присваивать аппаратным прерываниям приоритеты, то высокоуровневые прерывания могут вытеснять низкоуровневые. Этот механизм полностью поддерживается в системе QNX.

В предыдущих примерах описаны простейшие и наиболее распространенные ситуации, когда поступает только одно прерывание. Практически такие же временные характеристики имеют прерывания с высшим приоритетом. При расчете наихудших временных показателей для низкоприоритетных прерываний следует учитывать время обработки всех высокоприоритетных прерываний, так как в системе QNX высокоприоритетные прерывания вытесняют низкоприоритетные.

## Вопросы для самопроверки

1. Какие типичные процессы вкючены в систему QNX?
2. Назовите основные функции ядра ОС QNX.
3. Какие типы связи между процессами поддерживает ядро ОС QNX?
4. Какие функции используются для связи между процессами посредством сообщений?
5. Какие виды блокирования существуют при связи между процессами посредством сообщений?
6. Как осуществляется связь между процессами посредством proxy?
7. Какие варианты приемки сигнала существуют в зависимости от видов обработки?
8. Для чего используются виртуальные каналы и виртуальные процессы?
9. Какие методы планирования реализованы в ОС QNX?
10. Что понимают под задержкой планирования?

# 5. АДМИНИСТРАТОР ПРОЦЕССОВ ОС QNX

## 5.1. Создание процессов

Администратор процессов тесно связан с ядром операционной системы. Несмотря на то что он разделяет с ядром одно и то же адресное пространство (единственный из всех системных процессов), он выполняется как обычный процесс. Это означает, что администратор процессов планируется к выполнению ядром и использует те же примитивы передачи сообщений для взаимодействия с другими процессами. Администратор процессов отвечает за создание в системе новых процессов и управление ресурсами, связанными с процессом. Все эти функции реализуются посредством передачи сообщений. Например, создание выполняющимся процессом нового процесса осуществляется посредством посылки сообщения, содержащего подробную информацию о вновь создаваемом процессе. Поскольку передача сообщений распространяется на всю сеть, то можно легко создать новый процесс на другом узле, послав соответствующее сообщение администратору процессов удаленного узла.

Функциями, используемыми ОС QNX для запуска других процессов, являются:

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

Вид применяемой функции зависит от двух требований: переносимости и функциональности.

**`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 функция `system()` запускает копию командного интерпретатора.

**`Fork()`** порождает процесс, являющийся точной копией порождающего процесса. Новый процесс выполняется в том же

адресном пространстве и наследует все данные порождающего процесса.

Между тем родительский и дочерний процессы имеют различные идентификаторы процессов, поскольку в системе не может быть двух процессов с одинаковыми идентификаторами. Есть и еще одно отличие, это значение, возвращаемое функцией `fork()`. В дочернем процессе функция возвращает ноль, а в родительском процессе — идентификатор дочернего процесса.

Пример использования функции `fork()`:

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

**`Vfork()`** также порождает процесс. В отличие от функции `fork()`, позволяет существенно сэкономить на ресурсах, поскольку она формирует разделяемое адресное пространство родителя. Функция `vfork()` создает дочерний процесс, а затем приостанавливает родительский до тех пор, пока дочерний процесс не вызовет функцию `exec()` или не завершится.

**`Exec()`** заменяет образ порождающего процесса образом нового процесса. Возврата управления из нормально отработавшего `exec()` не существует, так как образ нового процесса накладывается на образ порождающего процесса. В системах стандарта POSIX новые процессы обычно создаются без возврата управления порождающему процессу — сначала вызывается `fork()`, а затем из порожденного процесса — `exec()`.

**`Spawn()`**создает новый процесс по принципу «отец–сын». Это позволяет избежать использования примитивов `fork()`и `exec()`, что ускоряет обработку и является более эффективным средством создания новых процессов. В отличие от `fork()` и `exec()`, которые по определению создают процесс на том же узле, что и порождающий процесс, примитив `spawn()`может создавать процессы на любом узле сети.

Функции `fork()` и `exec()` определены стандартом POSIX, а функция `spawn()` реализована только в QNX.

При создании процесса с помощью одного из трех описанных выше примитивов он наследует многое от той программной среды, в которой выполнялся его «родитель». Конкретная информация представлена в табл. 5.1.

Таблица 5.1

Возможности наследования при создании процесса

| Наследуемый параметры | `fork()` | `exec()` | spawn() |
|---|---|---|---|
| Идентификатор процесса | Нет | Да | Нет |
| Открытые файлы | Да | На выбор | На выбор |
| Блокировка файлов | Нет | Да | Нет |
| Задержанные сигналы | Нет | Да | Нет |
| Маска сигнала | Да | На выбор | На выбор |
| Игнорируемые сигналы | Да | На выбор | На выбор |
| Обработчик сигналов | Да | Нет | Нет |
| Переменные среды | Да | На выбор | На выбор |
| Идентификатор сеанса | Да | Да | На выбор |
| Группа процесса | Да | Да | На выбор |
| Реальные идентификаторы группы и пользователя ( UID, GID ) | Да | Да | Да |
| Эффективные UID, GID | Да | На выбор | На выбор |
| Текущий рабочий каталог | Да | На выбор | На выбор |
| Маска создания файлов | Да | Да | Да |
| Приоритет | Да | На выбор | На выбор |
| Метод планирования | Да | На выбор | На выбор |
| Виртуальные каналы | Нет | Нет | Нет |
| Символические имена | Нет | Нет | Нет |
| Таймеры реального времени | Нет | Нет | Нет |

Каждый процесс проходит четыре фазы:

1) ***создание процесса,*** заключающееся в присвоении идентификатора процесса ID новому процессу и задании информации, определяющей его программную среду. Большая часть этой информации наследуется от «родителя» нового процесса;

2) ***загрузка образов процессов,*** выполняемая загрузчиком «по цепочке». Загрузчик входит в состав администратора процессов и выполняется под идентификатором нового процесса.

Это позволяет администратору процессов выполнять другие запросы при загрузке программ;

3) ***выполнение процесса,*** начинающееся после загрузки программного кода. Процесс начинает конкурировать с другими процессами, стремясь получить ресурсы центрального процессора. ***Обратите внимание***: процессы выполняются конкурентно вместе со своими «родителями». Кроме того, гибель «родителя» не вызывает автоматически гибель порожденных им процессов;

4) ***завершение процесса,*** осуществляемое одним из способов:

- по сигналу, определяющему процесс завершения;
- при возврате управления по функции `exit()` — явно, либо по функции `main()` — по умолчанию.

Завершение включает в себя две стадии.

1. Администратор процессов инициирует выполнение программы завершения «по цепочке». Программа завершения входит в состав администратора процессов и выполняется под идентификатором завершающегося процесса. При этом осуществляется закрытие всех описателей открытых файлов и освобождаются следующие ресурсы:

- все виртуальные каналы, которые имел процесс;
- вся память, выделенная процессу;
- все символические имена;
- все номера основных устройств (только для администраторов ввода/вывода);
- все обработчики прерываний;
- все proxy;
- все таймеры.

2. После запуска программы завершения к породившему ее процессу посылается уведомление о завершении порожденного процесса. Эта стадия выполняется внутри администратора процессов.

Если породивший процесс не выдал `wait()` или `waitpid()`, то порожденный процесс становится так называемым «зомби-процессом» и не завершается до тех пор, пока породивший процесс не выдаст `wait()` или не завершится сам.

Для того чтобы не ждать завершения порожденного процесса, следует либо установить признак `_SPAWN_NOZOMBIE` в функ-

циях `qnx_spawn()` или `qnx_spawn_option()`, либо в функции `signal()` задать для `SIGCHLD` признак `SIG_IGN`. В этом случае порожденные процессы при завершении не становятся «зомби-процессами».

Породивший процесс может ожидать завершения порожденного процесса на удаленном узле. Если процесс, породивший «зомби-процесс», завершается, то освобождаются все ресурсы, связанные с «зомби». Если процесс завершается по сигналу завершения и при этом выполняется утилита dumper, то формируется дамп образа памяти. Этот дамп можно просмотреть с помощью символьного отладчика.

## 5.2. Состояния процессов

Процесс всегда находится в одном из следующих состояний:

**READY** (готов) — процесс может использовать центральный процессор (т. е. процесс не ждет наступления какого-либо события);

**BLOCKED** (блокирован) — процесс находится в одном из следующих состояний блокировки:

SEND-блокирован;
RECEIVE-блокирован;
REPLY-блокирован;
SIGNAL-блокирован;

**HELD** (задержан)— процесс получил сигнал SIGSTOP. До тех пор пока он не выйдет из состояния HELD, ему не разрешается использовать центральный процессор. Вывести из состояния HELD можно либо выдачей сигнала SIGCONT, либо завершением процесса по сигналу;

**WAIT** (ожидает) — процесс выдал `wait()` или `waitpid()` и ожидает информацию о состоянии порожденных им процессов;

**DEAD** (мертв) — процесс завершен, но не может передать информацию о своем состоянии породившему его процессу, поскольку тот не выдал функцию `wait()` или `waitpid()`. За завершенным процессом сохраняется состояние, но занимаемая память освобождается. Процесс в состоянии DEAD также назы-

вают «зомби-процессом». Определены следующие переходы из одного состояния в другое:

- процесс посылает сообщение;
- процесс-получатель принимает сообщение;
- процесс-получатель отвечает на сообщение;
- процесс ожидает сообщения;
- процесс принимает сообщение;
- сигнал разблокирует процесс;
- сигнал пытается разблокировать процесс; получатель запрашивает сообщение о захвате сигнала;
- процесс-получатель принимает сигнал;
- процесс ожидает завершения порожденного процесса;
- порожденный процесс завершается, либо сигнал разблокирует процесс;
- процессу выдан SIGSTOP;
- процессу выдан SIGCONT;
- процесс завершается;
- порождающий процесс ожидает завершения, завершается сам или уже завершен.

Определить состояние конкретного процесса возможно из интерпретатора (Shell) с помощью утилит `ps` или `sin`; из программы с помощью функции `qnx_psinfo()`.

Определить состояние операционной системы в целом возможно из интерпретатора с помощью утилиты `sin`; из программы с помощью функции `qnx_osinfo()`.

Утилита `ps` определена стандартом POSIX, командные файлы с ее использованием являются мобильными. Утилита `sin` уникальна в QNX, она предоставляет полезную информацию о системе QNX, которую нельзя получить с помощью утилиты ps.

В QNX обеспечивается возможность разработки приложений, представляющих собой набор взаимодействующих процессов. Эти приложения отличаются более высокой возможностью параллельной обработки данных и, кроме того, могут распределяться в сети, повышая производительность системы.

Однако разбиение приложения на взаимодействующие процессы требует специальных соглашений. Чтобы такие процессы

могли надежно связываться друг с другом, они должны иметь возможность определять идентификаторы (ID) друг друга. Рассмотрим, например, сервер базы данных, который работает с произвольным количеством обслуживаемых процессов (клиентов). Клиенты могут обращаться к серверу в любое время, а сервер всегда должен быть доступен. Каким образом клиенты определяют идентификатор сервера базы данных, для того чтобы послать ему сообщение?

В QNX эта проблема решается путем предоставления возможности присваивать процессам символические имена. В случае одного узла процессы могут зарегистрировать это имя с помощью ***администратора процессов*** на том узле, где они выполняются. Другие процессы могут затем получить у администратора процессов идентификатор процесса, соответствующий этому имени.

В случае работы в сети проблема усложняется, так как сервер должен обслуживать клиентов, которые находятся на разных узлах сети. В QNX имеется возможность поддерживать работу, как с глобальными, так и с локальными именами. Глобальные имена доступны во всей сети, а локальные — только на том узле, где они зарегистрированы. Глобальные имена начинаются со знака «слэш» (/). Например:

qnx локальное имя;

company/xyz локальное имя;

/company/xyz глобальное имя.

Для того чтобы использовать глобальные имена хотя бы на одном из узлов сети, необходимо запустить определитель имен процессов (утилита nameloc), который содержит записи всех зарегистрированных глобальных имен. В одно и то же время в сети могут работать до десяти определителей имен процессов. Каждый имеет идентичную копию всех активных глобальных имен. Эта избыточность обеспечивает надежность работы сети, гарантируя работоспособность при одновременном аварийном завершении нескольких определителей имен процессов. Для регистрации имени процесс-сервер использует функцию Си `qnx_name_attach()`. Для определения имени процесса процесс-клиент использует функцию Си `qnx_name_locate()`.

## 5.3. Управление потоками

Процесс может содержать один или несколько потоков. Число потоков варьируется. Один разработчик программного обеспечения, используя только единственный поток, может реализовать те же самые функциональные возможности, что и другой, используя 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.

Любой поток может создать другой поток в том же самом процессе. На это не налагается никаких ограничений (за исключением объема памяти). Как правило, для этого применяется функция POSIX `pthread_create()`:

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

Параметрами функции `pthread_create()` являются:

`thread` — указатель на pthread_t, где хранится идентификатор потока;

`attr` — атрибутивная запись;

`start_routine` — подпрограмма с которой начинается поток;

`arg` — параметр, который передается подпрограмме `start_routine`.

Параметры `thread` и `attr` необязательные, вместо них можно передавать NULL. Параметр thread можно использовать для хранения идентификатора вновь создаваемого потока.

**Пример однопоточной программы**

Предположим, что мы имеет программу, выполняющую алгоритм трассировки луча. Каждая строка растра не зависит от остальных. Это обстоятельство (независимость строк растра)

автоматически приводит к программированию данной задачи как многопоточной.

```
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 видно, что программа независимо по всем значениям x1 рассчитывает необходимые растровые строки.

**Пример многопоточной программы для параллельного выполнения функции `do_one_line (x1)`**

Однопоточная программа изменяется следующим образом:

```
int main ( int argc, char **argv)
{
int x1;
… // Выполнить инициализации
for (x1 = 0; x < num_x_lines; x1++)
{
pthread_create (NULL, NULL, do_one_line,
                (void *) x1);
}
… // Вывести результат
}
```

В вышеприведенном примере непонятно, когда нужно выполнять вывод результатов, так как приложение запустило массу потоков, но неизвестно, когда они завершатся. Можно поставить задержку выполнения программы (`sleep 1`), но это будет неправильно. Лучше использовать функцию `pthread_join()`.

Есть еще один минус у приведенной выше программы, если у нас много строк в изображении, не факт, что все созданные потоки будут функционировать параллельно, как правило, про-

цессоров в системе, гораздо меньше. В этом случае лучше модифицировать программу так, чтобы запускалось столько потоков, сколько у нас процессоров в системе.

**Пример многопоточной программы, учитывающей количество процессоров в системе**

```
int num_lines_per_cpu;
int num_cpus;

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

… // Выполнить инициализации

// Получить число процессоров
num_cpus = _syspage_ptr->num_cpu;

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);
}
// Синхронизировать с завершением всех по-
токов
for (cpu = 0; cpu < num_cpus; cpu++)
{
pthread_join (thread_ids [cpu], NULL);
}
… // Вывести результат
}
void *do_one_batch (void *c)
```

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

## 5.4. Управление таймером

В QNX управление временем основано на использовании системного таймера, который содержит текущее координатное универсальное время (UTC) относительно 0 часов 0 минут 0 секунд 1 января 1970 г. Для установки местного времени функции управления временем используют переменную среды TZ.

Программы интерпретатора Shell и процессы могут быть задержаны на заданное количество секунд с помощью простой утилиты таймирования. Программы интерпретатора используют для этого утилиту sleep; процессы — функцию Си `sleep()`. Можно также воспользоваться функцией `delay()`, в которой задается интервал времени в миллисекундах.

Процесс может также создавать таймеры, задавать им временной интервал и удалять их. Эти более сложные средства таймирования соответствуют стандарту POSIX 1003.4/Draft 9.

Процесс может создать один или несколько таймеров. Таймеры могут быть любого типа, поддерживаемого системой, а их количество ограничивается максимально допустимым количеством таймеров в системе.

Для создания таймера используется функция Си `mktimer()`. Эта функция позволяет задавать следующие типы механизма ответа на события:

- перейти в режим ожидания до завершения. Процесс будет находиться в режиме ожидания, начиная с момента установки таймера до истечения заданного интервала времени;
- оповестить с помощью proxy. Proxy используется для оповещения процесса об истечении времени ожидания;

• оповестить с помощью сигнала. Сформированный пользователем сигнал выдается процессу по истечении времени ожидания.

Таймеру можно задать следующие временные интервалы:

*абсолютный.* Время относительно 0 часов, 0 минут, 0 секунд, 1 января 1970 г.;

*относительный.* Время относительно значения текущего времени.

Можно также задать повторение таймера на заданном интервале. Например, вы установили таймер на 9 утра завтрашнего дня. Его можно установить так, чтобы он срабатывал каждые пять минут после истечения этого времени. Можно также установить новый временной интервал существующему таймеру. Результат этой операции зависит от типа заданного интервала:

• для абсолютного таймера новый интервал замещает текущий интервал времени;

• для относительного таймера новый интервал добавляется к оставшемуся временному интервалу.

Для установки абсолютного временного интервала используйте функцию `abstimer()`.

Для установки относительного временного интервала используйте функцию `reltimer()`. Для удаления таймера воспользуйтесь функцией Си `rmtimer()`. Таймер может удалить сам себя по истечении временного интервала при вызове функции `rmtimer()` с включенной опцией _TNOTIFY_SLEEP при условии, что таймер неповторяемый.

Период таймера задается утилитой ticksize или функцией Си `qnx_timerperiod()`. Вы можете выбрать период в интервале от 500 мкс до 50 млс.

Для определения оставшегося времени таймирования или для того, чтобы узнать, был ли таймер удален, используйте функцию Си `gettimer()`.

**Пример программы работы с таймером**

Данная программа запускает в цикле ожидание на 10 секунд и прерывает это ожидание с помощью сигнала:

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

```
   #include <stdio.h>
   #include <sys/siginfo.h>
   #include <sys/neutrino.h>
   #include <signal.h>
   #include <time.h>
   #include <errno.h>
   #include <unistd.h>
   #include <stdlib.h>
   int main(void)
   {
   struct itimerspec timer; // структура с
описанием
                     //таймера
   timer_t timerid;  // ID таймера
   extern void handler(); //Обработчик таймера
   struct sigaction act;   //Структура описы-
вающая //действие //по сигналу
   sigset_t set; //Набор сигналов нам необхо-
димый //для таймера
   sigemptyset( &set ); //Обнуление набора
   sigaddset( &set, SIGALRM); //Включение в
набор //сигнала //от таймера
   act.sa_flags = 0;
   act.sa_mask = set;
   act.sa_handler = &handler; // Вешаем обра-
ботчик // на действие
   sigaction( SIGALRM, &act, NULL); //Зарядить
сигнал, // присваивание структуры // для кон-
кретного сигнала // (имя сигнала, // структу-
ра-действий)//Создать таймер
   if (timer_create (CLOCK_REALTIME, NULL,
&timerid)
    == -1)
   {
     fprintf (stderr, "%s: ne udalos timer
%d\n", "TM", errno);
   }
```

```
    // Данный макрос для сигнала SIGALRM не ну-
жен
    // Его необходимо вызывать
    // для пользовательских сигналов
    // SIGUSR1 или SIGUSR2. Функция
timer_create()
    // в качестве второго параметра должна
    // использовать &event.
    // SIGEV_SIGNAL_INIT(&event, SIGALRM);
    timer.it_value.tv_sec=  3; //Взвести таймер
                        //на 3 секунды
    timer.it_value.tv_nsec= 0;
    timer.it_interval.tv_sec=  3;
      //Перезагружать //таймер
    timer.it_interval.tv_nsec= 0; // через 3
секунды
    timer_settime (timerid, 0, &timer, NULL);
     //Включить таймер
      for (;;)
    {
    sleep(10); // Спать десять секунд.
              // Использование таймера задержки
    printf("More time!\n");
    }
    exit(0);
    }
    void handler( signo )
    {
    //Вывести сообщение в обработчике.
    printf( "Alarm clock ringing!!!.\n");
    // Таймер заставляет процесс проснуться.
    }
```

Здесь `it_value` и `it_interval` принимают одинаковые значения. Такой таймер сработает один раз (с задержкой `it_value`), а затем будет циклически перезагружаться с задержкой `it_interval`.

Оба параметра `it_value` и `itinterval` фактически являются структурами типа ***struct timespec*** — еще одного POSIX-объекта. Эти структуры позволяют вам обеспечить разрешающую способность на уровне долей секунд. Первый 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 млрд — это будет подразумевать смещение на более чем 1 с).

**Задание относительного однократного таймера:**

```
t_value.tv_sec = 5;
it_value.tv_nsec = 500000000;
it_interval.tv_sec = 0;
it_interval.tv_nsec = 0;
```

Таким образом сформируется однократный таймер, который сработает через 5,5 с. (5,5 с складывается из 5 с и 500,000,000 нс). Мы предполагаем здесь, что этот таймер используется как относительный, потому что если бы это было не так, то его время срабатывания уже давно бы истекло (5,5 с с момента 00:00, по Гринвичу, 1 января 1970).

**Задание абсолютного однократного таймера:**

```
it_value.tv sec = 987654321;
it_value.tv_nsec = 0;
it_interval.tv_sec = 0;
it_iriterval. tv_nsec = 0;
```

Данная комбинация параметров сформирует однократный таймер, который сработает в четверг, 19 апреля 2001 года

В программе могут быть использованы следующие функции:

**`TimerCreate(), TimerCreate_r()`** — функции создания таймера:

```
#include <sys/neutrino.h>
int TimerCreate( clockid_t id,
           const struct sigevent *event );
```

```
int TimerCreate_r( clockid_t id,
           const struct sigevent *event );
```

Обе функции идентичны, исключение составляют лишь способы обнаружения ими ошибок:

1) при использовании функции `TimerCreate()`, если происходит ошибка, то возвращается значение –1;

2) при использовании функции `TimerCreate_r()` при возникновении ошибки ее конкретное значение возвращается из секции Errors.

В приведенных выше функциях может быть использована структура ***Struct sigevent***, описывающая событие таймера:

```
#include <sys/siginfo.h>
union sigval {
    int          sival_int;
    void        *sival_ptr;
};
```

Файл <sys/siginfo.h> определяет также некоторые макросы для более облегченной инициализации структуры `sigevent`. Все макросы ссылаются на первый аргумент структуры `sigevent` и устанавливают подходящее значение `sigev_notyfy` (уведомление о событии):

**SIGEV_INTR** — увеличить прерывание. В этой структуре не используются никакие поля. Инициализация макроса имеет следующий вид: `SIGEV_INTR_INIT(event);`

**SIGEV_NONE —** не посылать никаких сообщений. Используется без полей. Инициализация: `SIGEV_NONE_INIT (event);`

**SIGEV_PULSE —** посылать периодические сигналы. Имеет следующие поля:

`int sigev_coid` — ID подключения. По нему происходит связь с каналом, откуда будет получен сигнал;

`short sigev_priority` — установка приоритета сигналу;

`short sigev_code` — интерпретация кода в качестве манипулятора сигнала. `Sigev_code` может быть любым 8-битным значением, чего нужно избегать в программе. Значение `sigev_code` меньше нуля приводит к конфликту в ядре.

Инициализация макроса имеет вид: `SIGEV_PULSE_INIT (event, coid, priority, code, value);`

**SIGEV_SIGNAL** — послать сигнал процессу. В качестве поля используется `int sigev_signo` (повышение сигнала). Может принимать значение от 1 до –1. Инициализация макроса имеет следующий вид: `SIGEV_SIGNAL_INIT(event, signal)`.

В программе может быть использована также и функция **SignalAction(), SignalAction_r()** // , определяющая действия для сигналов:

```
#include <sys/neutrino.h>
int SignalAction(pid_t pid,
                 void (*sigstub)(),
                 int signo,
                 const struct sigaction* act,
                 struct sigaction* oact );
int SignalAction_r( pid_t pid,
                 void* (sigstub)(),
                 int signo,
                 const struct sigaction* act,
                 struct sigaction* oact );
```

Все значения ряда сигналов идут от_SIGMIN (1) до _SIGMAX (64).

Если параметр вышеприведенных функций act не NULL, тогда модифицируется указанный сигнал; если oact не NULL, то предыдущее действие сохраняется в структуре, на которую он указывает. Использование комбинации act и oact позволяет запрашивать или устанавливать (либо и то и другое) действия, котрые будут произведены по сигналу.

Структура ***sigaction*** содержит следующие параметры:

`void (*sa_handler)()` — возврат адрес манипулятора сигнала или действия для неполученного сигнала, действие-обработчик;

`void (*sa_sigaction) (int signo, siginfo_t *info, void *other)` — возврат адрес манипулятора сигнала или действия для полученного сигнала;

`sigset_t sa_mask` — дополнительная установка сигналов для изолирования (блокирования) функций, улавливающих сигнал в течение исполнения;

`int sa_flags` — специальные флаги, для того чтобы влиять на действие сигнала (SA_NOCLDSTOP и SA_SIGINFO):

– `SA_NOCLDSTOP` используется только в случае, когда сигнал является дочерним (SIGCHLD). Система не создает дочерний сигнал внутри родительского, он останавливается через SIGSTOP;

– SA_SIGINFO сообщает Neutrino о постановке в очередь текущего сигнала. Если установлен флаг SA_SIGINFO, сигналы ставятся в очередь и все передаются в порядке очередности.

Добавление сигнала на установку:

```
#include <signal.h>
int sigaddset( sigset_t *set,
               int signo );
```

Функция ***sigaddset()*** добавляет `signo` в `set` по указателю, осуществляет присвоение сигнала набору. Функция ***sigaddset()*** возвращает 0 при удачном исполнении и –1 в случае ошибки;

Функция ***sigemptyset()*** осуществляет обнуление набора сигналов:

```
#include <signal.h>
int sigemptyset( sigset_t *set );
```

Эта функция возвращает 0 при удачном исполнении и –1 в случае ошибки.

## 5.5. Обработчики прерываний

Обработчики прерываний обслуживают прерывания аппаратной части компьютерной системы, реагируют на аппаратные прерывания и управляют на нижнем уровне передачей данных между компьютером и внешними устройствами.

Физически обработчики прерываний формируются как часть стандартного процесса QNX (например, драйвера), но они всегда выполняются асинхронно с процессом, в котором содержатся.

Обработчик прерываний обладает следующими свойствами:

• запускается удаленным вызовом, а не прямо прерыванием (лучше писать его на языке Си, а не на Ассемблере);

• выполняется внутри процесса, в который встроен, поэтому имеет доступ ко всем глобальным переменным процесса;

• выполняется только для разрешенных прерываний и приоритетно обслуживает прерывания более высокого уровня;

• не взаимодействует непосредственно с контроллером прерываний (микросхемой 8259). Это делает операционная система.

По одному прерыванию (если это поддерживается аппаратно) могут запускаться несколько процессов. При возникновении физического прерывания каждому обработчику прерываний передается управление. В каком порядке обработчики прерываний разделяют обработку этого прерывания — не определено.

Если вы хотите установить аппаратное прерывание, используйте функцию `qnx_hint_attach()`.

Если вы хотите удалить аппаратное прерывание, используйте функцию `qnx_hint_detach()`.

Можно подключить обработчик прерываний напрямую к системному таймеру таким образом, чтобы обработчик запускался по каждому прерыванию от таймера. Для установки периода используйте утилиту ticksize.

Можно также подключиться к масштабируемому прерыванию от таймера, которое выдается каждые 100 млс в зависимости от значения ticksize. Эти таймеры являются альтернативой таймерам стандарта POSIX 1003.4 при обработке прерываний нижнего уровня.

## Вопросы для самопроверки

1. Какие механизмы существуют в ОС QNX для создания процессов?
2. Какие фазы проходит каждый процесс?
3. В каких состояниях могут находиться процессы?
4. Расскажите про механизм управления таймером.
5. Какими свойствами обладает обработчик прерываний?

# 6. УПРАВЛЕНИЕ РЕСУРСАМИ В ОС QNX

## 6.1. Администраторы ресурсов

Управление ресурсами ЭВМ — одна из главных функций любой операционной системы. К основным ресурсам, которыми управляет QNX, относятся файловые системы, символьные устройства ввода/вывода (последовательные и параллельные порты, сетевые карты и т. д.) и виртуальные устройства (нуль-устройство, псевдотерминалы, генератор случайных чисел и т. п.). В QNX поддержка ресурсов не встроена в микроядро и организована с помощью специальных программ и динамически присоединяемых библиотек, получивших название «администраторы ресурсов». Взаимодействие между администраторами ресурсов и другими программами реализовано через четко определенный, хорошо документированный интерфейс файлового ввода/вывода. Хотя, конечно, на самом деле весь обмен построен на базовом механизме QNX-сообщений микроядра Neutrino.

***Администратор ресурсов*** — это прикладная серверная программа, принимающая QNX-сообщения от других программ и, при необходимости, взаимодействующая с аппаратурой [11]. Для связи между программой-клиентом и администратором ресурсов используется механизм пространства имен.

Работу администратора ресурсов легко представить в виде последовательности действий:

1) инициализация интерфейса сообщений, при этом создается канал, по которому клиенты могут посылать свои сообщения администратору ресурсов;

2) регистрация путевого имени (т. е. зона ответственности) в пространстве имен администратора процессов;

3) запуск бесконечного цикла по приему сообщений от клиентов;

4) выполнение переключения на нужный обработчик для каждого типа сообщения с помощью операторов switch/case.

## 6.2. Файловые системы в QNX

**Классификация файловых систем в QNX**

ОС QNX обеспечивает поддержку различных файловых систем с помощью соответствующих администраторов ресурсов, каждый из которых при запуске регистрирует у администратора процессов зону ответственности в виде путевого имени. Такая реализация позволяет запускать и останавливать любую комбинацию файловых систем динамически.

Файловые системы, поддерживаемые в QNX, классифицируются следующим образом:

***образная файловая система*** (image filesystem) — простая файловая система «только для чтения», состоящая из модуля procnto и других файлов, включенных в загрузочный образ QNX. Этот тип файловой системы поддерживается непосредственно администратором процессов и достаточен для многих встроенных систем. Если же требуется обеспечить поддержку других файловых систем, то модули их поддержки добавляются в образ и могут запускаться по мере необходимости;

***RAM*** — плоская файловая система, которую автоматически поддерживает администратор процессов. Файловая система RAM основана на использовании ОЗУ и позволяет выполнять операции чтения/записи из каталога /dev/shmem. Этот тип файловой системы нашел применение в очень маленьких встроенных системах, в которых не требуется хранение данных на энергонезависимом носителе и для которых достаточно ограниченных функциональных возможностей (нет поддержки каталогов, жестких и мягких ссылок);

***блочные файловые системы*** — традиционные файловые системы, обеспечивающие поддержку блок-ориентированных устройств типа жестких дисков и дисководов CD-ROM. К ним относятся файловые системы QNX4, DOS, Ext2 и CD-ROM:

- файловая система QNX4 (**fs-gnx4.so**) — высокопроизводительная файловая система, сохранившая формат и структуру дисков ОС QNX4, однако усовершенствованная для повышения надежности, производительности и совместимости со стандартом POSIX;

• файловая система DOS **(fs-dos.so)** обеспечивает прозрачный доступ к локальным разделам FAT (12, 16, 32), при этом файловая система конвертирует POSIX-примитивы работы с диском в соответствующие DOS-команды. Если эквивалентную операцию выполнить нельзя (например, создать символьную ссылку), то возвращается ошибка;

• файловая система CD-ROM (**fs-cd.so**) обеспечивает прозрачный доступ к файлам на компакт-дисках формата ISO 9660 и их расширениям (Rock Ridge, Joliet, Kodak Photo CD и Audio);

• файловая система Ext2 **(fs-ext2.so)** обеспечивает прозрачный доступ из среды QNX к Linux-разделам жесткого диска как 0, так и 1 версии;

***Flash*** — не блок-ориентированные файловые системы, разрабатываемые специально для устройств флэш-памяти;

***Network*** — файловые системы, обеспечивающие доступ к файловым системам на других ЭВМ. К ним относятся файловые системы NFS и CIFS (SMB):

• файловая система NFS (Network File System) обеспечивает прозрачный доступ клиентской рабочей станции через сеть к файлам независимо от операционных систем, используемых файл-серверами. NFS использует механизм удаленного вызова процедур (RPC) и работает поверх TCP/IP;

• файловая система CIFS (Common Internet File System) обеспечивает клиентским станциям прозрачный доступ к сетям Windows, а также к UNIX-системам с запущенным сервером SMB. Работает поверх TCP/IP;

***Virtual*** — особые файловые системы, обеспечивающие специфические функциональные возможности при работе с другими файловыми системами:

• пакетная файловая система, обеспечивающая привычное представление выделенных файлов и каталогов для клиента. Эта файловая система будет подробно рассмотрена ниже;

• Inflator — администратор ресурсов, зона ответственности которого устанавливается ближе к корню файловой системы, предназначенный для динамического разжимания файлов, сжатых утилитой deflate.

**Реализация поддержки файловых систем**

Поскольку у некоторых файловых систем, работающих в ОС QNX, много общих черт, то для максимизации повторного использования программного кода файловые системы проектируют как комбинацию драйверов и разделяемых библиотек. Такое решение позволяет существенно сократить количество дополнительной памяти, требуемой при добавлении файловой системы в QNX, поскольку добавляется только код, непосредственно реализующий протокол обмена с данной файловой системой.

Например, если администратор конфигурирования аппаратуры enum-devices обнаружил интерфейс EIDE, то запускается драйвер devb-eide. Этому драйверу для работы необходимо загрузить модуль поддержки блок-ориентированного ввода-вывода io-blk.so, который создает в каталоге /dev несколько блок-ориентированных файлов устройств. По умолчанию они обозначаются hdn (для жесткого диска) и cdn (для CD-ROM), n соответствует физическому номеру устройства. Кроме того, для каждого раздела жестких дисков создается свой блок-ориентированный специальный файл с именем hd*n*t*m*, где *n* — номер устройства, а *m* — тип раздела. Например, для раздела FAT32 на первом жестком диске будет создан файл /dev/hd0t11. Если разделов одного типа несколько, то они нумеруются дополнительно с разделением номеров точкой, например /dev/hd0t11.l.

Модуль io-bik.so обеспечивает для всех блочных файловых систем буферный кэш, в который помещаются данные при выполнении записи на диск. Это позволяет значительно сократить число операций чтения/записи с физическим диском, т. е. повышает производительность работы файловых систем. Однако критичные с точки зрения надежности функционирования блоки файловой системы (например, информация о структуре диска) записываются на диск немедленно и синхронно, минуя обычный механизм записи.

Для доступа к жестким дискам, компакт-дискам и оптическим дискам драйверу необходимо подгрузить соответствующие модули поддержки общих методов доступа — соответственно

cam-disk.so, cam-cdrom. so и/или cam-optical.so.

Для поддержки собственно блочных файловых систем модуль io-blk.so загружает необходимые администраторы файло-

вых систем, также реализованные в виде динамически присоединяемых библиотек. Поддержка блочных файловых систем реализована в модулях администраторов ресурсов:

fs-qnx4.so — файловой системы QNX4;

fs-ext2.so — файловой системы Ext2;

fs-dos.so — файловой системы FAT32;

fs-cd.so — файловой системы ISO9660.

Администраторы файловых систем монтируют свои разделы в определенные точки файловой системы. Раздел QNX4, выбранный в процессе загрузки как первичный, монтируется в корень файловой системы — /. Остальные разделы по умолчанию монтируются в каталог /fs. Например, компакт-диск монтируется в точку /fs/cd, а файловая система FAT32 монтируется в точку /fs/hd0-dos.

Для того чтобы программа diskboot могла использовать раздел для поиска базового образа файловой системы QNX, необходимо наличие файла /.diskroot.

Операция по изменению информации на диске выполняется в форме транзакции, поэтому даже при катастрофических сбоях (например, при отключении питания) файловая система QNX4 остается цельной. В худшем случае некоторые блоки могут быть выделены, но не использованы, вернуть эти блоки можно, запустив утилиту chkfsys.

Дерево каталогов, которое пользователь привык видеть на экране, — это виртуальная файловая система, которая была создана и управляется администратором пакетной файловой системы fs-pkg.

## 6.3. Инсталляционные пакеты и их репозитарии

При создании ОС QNX Neutrino разработчиками учитывались вопросы обеспечения простого и эффективного механизма поставки и инсталляции программного обеспечения для QNX. В настоящее время очень распространенным средством доставки ПО от разработчика к пользователю стал Интернет. Поэтому было принято решение, что файлы, входящие в состав программного продукта для QNX, следует помещать в архив формата

TGZ, т. е. все файлы компонуются в один утилитой tar, и этот файл сжимается утилитой gzip. Такой архив и есть QNX-пакет — файл, имеющий расширение qpk.

Чтобы избавить пользователя от необходимости скачивать весь пакет для получения информации о производителе, версии, совместимости с другим программным обеспечением. Такого рода информация помещается в отдельный от пакета файл манифеста, имеющий расширение qpm. Одну или несколько пар файлов qpk и qpm можно с помощью тех же tar и gzip объединить в репозитарии qpr. В этом случае информация о содержимом репозитария помещается в отдельном файле-манифесте qrm. Репозитарий с его манифестом можно помещать на любом носителе — CD-ROM, веб-сайте и т. п. Для установки пакетов в операционную систему QNX используется уже знакомый вам инсталлятор QNX Software Installer — программа gnxinstall.

Рассмотренный нами тип репозитария — это репозитарий для хранения готовых к инсталляции пакетов программного обеспечения. В составе ОС QNX есть необходимые инструменты для построения собственных инсталляционных пакетов.

В самой ОС QNX есть два типа инсталляционных пакетов:

1) /pkgs/base для базовой системы QNX;

2) /pkgs/repository для дополнительного программного беспечения.

Эти пакеты, хранящиеся в соответствующих каталогах, содержат реальные объекты файловой системы на диске.

Содержимое репозитариев имеет определенную структуру, позволяющую избежать конфликтов между программным обеспечением разных производителей, версиями, целевыми и хост-платформами. Путевое имя каталога, содержащего тот или иной пакет, имеет стандартизованный формат:

/pkgs/repository/производитель/ продукт/каталог_пакета

Внутри каждого пакета обязательно есть XML-файл MANIFEST, который полностью описывает пакет. Остальные файлы и каталоги организованы таким образом, как они должны располагаться в файловой системе, причем корневым каталогом считается каталог пакета. Важная деталь: файлы и каталоги пакета, специфичные для какой-либо аппаратной архитектуры, располагаются в отдельном каталоге (ppcbe, x86, shle и т. д.).

Такой подход, с одной стороны, позволяет экономить дисковое пространство — платформо-независимый код не дублируется. С другой стороны, на одной ЭВМ можно без конфликтов хранить файловые системы для разных аппаратных платформ, что при сетевой прозрачности QNX обеспечивает широкие возможности для машин с ограниченными ресурсами.

Администратор пакетной файловой системы fs-pkg отображает содержимое файловой системы в привычном для пользователя виде, скрывая пакетную структуру данных. Некоторые каталоги существуют реально: /var, /tmp и другие создаются при первом старте ОС после инсталляции. Информация о конфигурации пакетов хранится в файле /etc/system/package/packages.

Идеология пакетов предполагает их неизменность, т. е. пакеты должны быть доступны только для чтения. А что же делать, если требуется модифицировать или вообще заменить какой-либо файл? Возможность таких изменений реализована с помощью каталога /var/pkg/spill. Именно туда помещаются измененные и добавленные файлы, а также записывается информация об «удалении».

Деинсталляция и деинициализация пакетов выполняются программой QNX Software Installer.

## 6.4. Символьные устройства ввода/вывода

Символьными устройствами ввода/вывода называют такие устройства, которые передают или принимают последовательность байтов один за другим, в отличие от блок-ориентированных устройств. Имена администраторов символьных устройств ввода/вывода имеют вид devc-*. Обычно в системе имеются следующие символьные устройства:

- консольные устройства (или текстовые консоли);
- последовательные порты;
- параллельные порты;
- псевдотерминалы (ptys).

Для максимального использования кода управления символьными устройствами используется статическая библиотека io-char, которая управляет потоками данных между приложением

и драйвером устройства посредством очередей разделяемой памяти. Каждая очередь работает по принципу FIFO.

Режимы ввода при работе устройств:

- поточный, или «сырого ввода» (raw) — наиболее производительный режим, однако io-char не выполняет никаких редактирований принимаемых данных;
- редактируемый (edited) — режим, при котором io-char может выполнять операции редактирования над каждым символом строки. После окончания редактирования строки она становится доступной для обработки прикладным процессом (обычно после ввода символа возврата каретки CR). Такой режим часто именуют каноническим.

**Консольные устройства**

Системные консоли управляются процессами-драйверами devc-con или devc-tcon. Совокупность клавиатуры и видеокарты с монитором называют физической консолью.

Консольный драйвер devc-con позволяет запускать на физической консоли несколько терминальных сессий посредством виртуальных консолей. При этом devc-con организует несколько очередей ввода/вывода к io-char через несколько символьных устройств с именами /dev/con1, /dev/con2 и т. д. С точки зрения приложения создается эффект наличия нескольких консолей.

Консольный драйвер devc-tcon представляет собой «облегченную» (т. е. поддерживающую только одиночную консоль с «сырым» вводом) версию драйвера devc-con для систем с ограниченным объемом памяти.

**Последовательные устройства**

Последовательные каналы ввода/вывода управляются семейством процессов-драйверов devc-ser*. Каждый из драйверов может управлять более чем одним физическим устройством и обеспечивать поддержку нескольких символьных устройств.

**Параллельные устройства**

Параллельные принтерные порты (до четырех) управляются драйвером-процессом devc-par. Этот драйвер поддерживает только вывод, чтение из устройства /dev/par*n* дает результат, аналогичный чтению из /dev/null.

**Псевдотерминалы (ptys)**

Псевдотерминалы управляются драйверным процессом devc-pty. С помощью аргумента при запуске драйвера определяется количество псевдотерминалов. Псевдотерминал состоит из двух частей: основной (master) — драйвера и подчиненной (slave) — устройства. Подчиненное устройство обеспечивает для прикладных процессов интерфейс, идентичный обычному POSIX-терминалу. Обычный терминал взаимодействует с аппаратным устройством, а подчиненное устройство псевдотерминала вместо этого взаимодействует с основным драйвером псевдотерминала. Основной драйвер может взаимодействовать с другим процессом. Таким образом, псевдотерминал может использоваться для того, чтобы процесс мог взаимодействовать с другим процессом как с символьным устройством. Псевдотерминалы, как правило, используются для создания интерфейса для таких программ, как эмуляторы терминала pterm или telnet.

## 6.5. Сетевая подсистема QNX

ОС QNX — система изначально сетевая, однако сетевые механизмы, как и все остальные, реализованы в виде дополнительных администраторов ресурсов. Хотя, некоторая поддержка сети имеется в микроядре: способ адресации QNX-сообщений обеспечивает возможность передачи их по сети. Собственно, для микроядра Neutrino безразлично, является сообщение сетевым или локальным. Это дает QNX мощный механизм поддержки сетевых кластеров.

В QNX реализованы средства поддержки двух протоколов: TCP/IP**,** являющегося промышленным стандартом; Qnet, «родного» протокола QNX, реализующего концепцию «прозрачной сети». Компьютеры, объединенные в сеть Qnet, фактически представляют собой виртуальную многопроцессорную суперЭВМ.

**Структура сетевой подсистемы QNX**

QNX является сетевой операционной системой, позволяющей организовывать эффективные распределенные вычисления. Упрощенно структуру сетевой подсистемы можно представить в виде схемы (рис. 6.1).

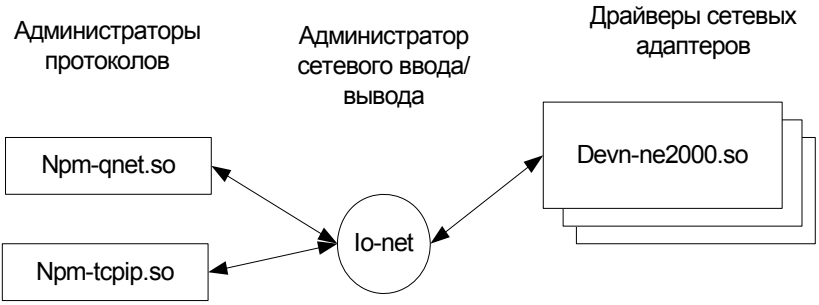


Рис. 6.1. Структура сетевой подсистемы QNX

В центре реализации сетевой подсистемы QNX Neutrino лежит администратор сетевого ввода/вывода io-net (менеджер Net). Процесс io-net при старте регистрирует префикс-каталог /dev/io-net и загружает необходимые администраторы сетевых протоколов и аппаратные драйверы. Протоколы, драйверы и другие необходимые компоненты (все они реализованы в виде DLL) загружаются либо в соответствии с аргументами командной строки, заданными при запуске io-net, либо в любое время командой монтирования mount.

Следующая команда запускает поддержку сети с драйвером сетевого адаптера NE2000 и поддержкой протоколов TCP/IP и Qnet: `io-net -dne2000 -ptcpip -pqnet`

Можно выполнить, например, такие команды:

1) запустить администратор сети: `io-net &`

2) загрузить драйвер Ethernet для адаптера NE2000:

```
mount -T io-net devn-ne2000.so
```

3) загрузить администратор протокола Qnet:

```
mount -T io-net npm-qnet.so
```

4) загрузить администратор протокола TCP/IP:

```
mount -T io-net npm-tcpip.so
```

Отключить поддержку, например, Qnet и выгрузить DLL npm-qnet.so можно следующей командой:

```
umount /dev/ io-net /qnet0
```

Для получения диагностической и статистической информации о работе сетевой карты используется утилита nicinfo (Net-

work Interface Card INFOrmation). По умолчанию nicinfo обращается к устройству /dev/io-net/en0. Для адаптеров Ethernet nicinfo сообщает наименование модели контроллера и другие характеристики:

- физический адрес (MAC-адрес) адаптера;
- MAC-адрес, присвоенный адаптеру (имеет смысл для адаптеров с программируемыми MAC-адресами);
- скорость и способ передачи данных;
- максимальный размер кадра, в байтах;
- логический номер сети (используется, если к ЭВМ подключено несколько сетевых адаптеров);
- диапазон портов;
- номер прерывания;
- признак включения режима приема всех пакетов;
- признак включения режима групповой передачи;
- общее число удачно отправленных кадров;
- общее число кадров, отправленных с ошибками;
- общее число удачно принятых кадров;
- общее число принятых пакетов, содержащих ошибки;
- количество отправленных байтов;
- количество полученных байтов;
- количество коллизий при передаче;
- количество коллизий при передаче с отменой передачи пакета;
- количество потерь при передаче;
- число попыток;
- количество пакетов, передача которых была отложена;
- число поздних коллизий;
- количество переполнений приемного буфера;
- число ошибок выравнивания;
- количество ошибок по несовпадению контрольной суммы.

Для разработки собственных драйверов сетевых карт в состав QNX Momentics входит специальный программный пакет, называемый Network Driver Development Kit (Network DDK), включающий исходные тексты нескольких драйверов и детальную инструкцию по написанию аппаратно-зависимого кода.

**Qnet**

Прозрачность сетевого взаимодействия в сети Qnet основана на способности QNX выполнять передачу сообщений между микроядрами Neutrino через сеть. Каким же образом можно отличить сетевое сообщение от локального, и, главное, как идентифицировать узлы сети? Для этой цели служит пространство имен путей администратора процессов. При загрузке и инициализации администратор npm-qnet.so регистрирует символьное устройство /dev/io-net/qnet0 и префикс-каталог /net, в котором помещаются папки с именами узлов сети. Например, если в сети есть узлы с именами alpha и beta, каталог /etc узла beta является каталогом /net/beta/etc узла alpha.

Таким образом, Qnet обеспечивает прозрачный доступ к файлам всех узлов сети с помощью обычных локальных средств: файлового менеджера, команды is и т. п. Кроме того, Qnet позволяет запускать процессы на любом узле сети. Для этого используется утилита on.

Утилита on является своего рода расширением возможностей командного интерпретатора по запуску приложений. Синтаксис утилиты таков: `on опции команда`. При этом команда будет выполнена в соответствии с предписаниями, заданными посредством опций. Укажем основные опции этой утилиты:

`-n node` указывает имя узла (node), на котором должна быть запущена команда. При этом в качестве файловой системы будет использована файловая система узла, с которого запущен процесс;

`-f node` производит те же действия, что и предыдущая опция, но в качестве файловой системы будет использована файловая система узла, на котором запущен процесс;

`-t tty` указывает, с каким терминалом целевой ЭВМ должен быть ассоциирован запускаемый процесс;

`-d` отсоединяет от родительского процесса. Если задана эта опция, то утилита on завершится, а запущенный процесс будет продолжать выполняться (т. е. он будет запущен с флагом SPAWN_NOZOMBIE);

`-p` (приоритет [дисциплина]) задает значение приоритета и, если требуется, дисциплину диспетчеризации. Например, команда:

```
on -d -n host5 -p 30f -t con2 inetd
```

запустит на консоли /dev/con2 узла с именем hosts программу inetd, находящуюся на нашем узле. Приоритет запущенного процесса будет иметь значение 30 с дисциплиной диспетчеризации FIFO. После запуска программы inetd утилита on сразу завершит свою работу, и интерпретатор выдаст приглашение для ввода очередной команды.

Здесь уместно вспомнить про то, как администратор пакетной файловой системы fs-pkg отображает платформенно-зависимые файлы. Как вы помните, такие файлы отображаются не только в корневой каталог /, но и в платформенно-зависимый каталог, например, `/ppcbe`. Это позволяет брать на удаленном узле те файлы, которые предназначены для выполнения на аппаратуре конкретного узла сети. Например, мы хотим на нашем бездисковом компьютере на базе процессора PowerPC (Big Endian) запустить утилиту `ls`, расположенную на узле с именем alpha и процессором Pentium III). Для этого выполним команду:

```
/net/alpha/ppcbe/usr/bin/ls -I /home
```

Этот механизм позволяет без конфликтов использовать дисковое пространство всех узлов сети Qnet независимо от аппаратуры, на которой работает QNX Neutrino. В сочетании с возможностями утилиты on администратор получает «виртуальную суперЭВМ». Но не забывайте про «ложку дегтя» — в сети Qnet забудьте о безопасности информации. Информацию о доступных узлах сети Qnet можно посмотреть командой sin net. В результате получим список доступных узлов Qnet-сети с информацией о каждом из них:

```
   $ sin net
   myhost  144M  1  631  Intel  686
F6M8S6
   hishost  467M  1  939  Intel  686
F6M8S10
```

Как видно из приведенного фрагмента, выдаются имя узла, размер доступной оперативной памяти, количество процессоров, тактовая частота и модель процессора.

***Обратите внимание***, что некоторые утилиты QNX имеют сетевую опцию, указывающую, к какому узлу сети применить действие утилиты (например, sin или slay).

**Протокол транспортного уровня FLEET**

Для организации обмена в сети используется надежный и эффективный протокол транспортного уровня FLEET. Каждый из узлов может принадлежать одновременно нескольким QNX-сетям. В том случае, если сетевое взаимодействие может быть реализовано несколькими путями, для передачи выбирается незагруженная и более скоростная сеть.

Сетевое взаимодействие является узким местом в большинстве операционных систем и обычно создает значительные проблемы для систем реального времени. Для того чтобы обойти это препятствие, разработчики QNX создали собственную специальную сетевую технологию FLEET и соответствующий протокол транспортного уровня FTL (FLEET transport layer). Этот протокол не базируется ни на одном из распространенных сетевых протоколов типа IPX или NetBios и обладает рядом качеств, которые делают его уникальным. Основные его качества зашифрованы в аббревиатуре FLEET, расшифровка которой приведена в табл. 6.1 [10].

Благодаря этой технологии сеть компьютеров с QNX фактически можно представлять как один виртуальный суперкомпьютер. Все ресурсы любого из узлов сети автоматически доступны другим, и для этого не нужны специальные «фокусы» с использованием технологии RPC. Это значит, что любая программа может быть запущена на любом узле, при этом ее входные и выходные потоки могут быть направлены на любое устройство на любых других узлах.

Например, утилита make в QNX автоматически распараллеливает компиляцию пакетов из нескольких модулей на все доступные узлы сети, а затем собирает исполняемый модуль по мере завершения компиляции на узлах. Специальный драйвер, входящий в комплект поставки, позволяет использовать для се-

тевого взаимодействия любое устройство, с которым может быть ассоциирован файловый дескриптор, например последовательный порт, что открывает возможности для создания глобальных сетей.

Таблица 6.1

Расшифровка аббревиатуры FLEET

| | |
|---|---|
| Fault-Tolerant Networking | QNX может одновременно использовать несколько физических сетей. При выходе из строя любой из них данные будут автоматически перенаправлены «на лету» через другую сеть |
| Load-Balancing on the Fly | При наличии нескольких физических соединений QNX автоматически распараллеливает передачу пакетов по соответствующим сетям |
| Efficient Performance | Специальные драйверы, разрабатываемые фирмой «QSSL» для широкого спектра оборудования, позволяют с максимальной эффективностью использовать сетевое оборудование |
| Extensablc Architecture | Любые новые типы сетей могут быть поддержаны путем добавления соответствующих драйверов |
| TransparentDistributed Processing | Благодаря отсутствию разницы между передачей сообщений в пределах одного узла и между узлами нет необходимости вносить какие-либо изменения в приложения, для того чтобы они могли взаимодействовать через сеть |

Достигаются все эти удобства за счет того, что поддержка сети частично обеспечивается и микроядром (специальный код в его составе позволяет QNX фактически объединять все микроядра в сети в одно ядро). Разумеется, за такие возможности приходится платить тем, что мы не можем получить драйвер для какой-либо сетевой платы от кого-либо еще, кроме фирмы «QSSL», то есть использоваться может только то оборудование, которое уже поддерживается. Однако ассортимент такого оборудования достаточно широк и периодически пополняется новейшими устройствами.

Когда ядро получает запрос на передачу данных процессу, находящемуся на удаленном узле, он переадресовывает этот запрос менеджеру Net, в подчинении которого находятся драйверы всех сетевых карт. Имея перед собой полную картину состояния всего сетевого оборудования, менеджер Net может отслеживать состояние каждой сети и динамически перераспределять нагрузку между ними. В случае когда одна из сетей выходит из строя, информационный поток автоматически перенаправляется в другую доступную сеть, что очень важно при построении высоконадежных систем. Кроме поддержки своего собственного протокола, Net обеспечивает передачу пакетов TCP/IP, SMB и многих других, используя то же сетевое оборудование. Производительность QNX в сети приближается к производительности аппаратного обеспечения.

При проектировании системы реального времени, как правило, необходимо обеспечить одновременное выполнение нескольких приложений. В QNX/Neutrino параллельность выполнения достигается за счет использования потоковой модели POSIX, в которой процессы в системе представляются в виде совокупности потоков. Поток является минимальной единицей выполнения и диспетчеризации для ядра Neutrino, процесс определяет адресное пространство для потоков. Каждый процесс состоит минимум из одного потока. QNX представляет богатый набор функций для синхронизации потоков. В отличие от потоков, само ядро не подлежит диспетчеризации. Код ядра исполняется только в том случае, когда какой-нибудь поток вызывает функцию ядра, или при обработке аппаратного прерывания.

Напомним, что QNX базируется на концепции передачи сообщений. Передачу сообщений, а также их диспетчеризацию осуществляет ядро системы. Кроме того, ядро управляет временными прерываниями. Выполнение остальных функций обеспечивается задачами-администраторами. Программа, желающая создать задачу, посылает сообщение администратору задач (модуль task) и блокируется для ожидания ответа. Если новая задача должна выполняться одновременно с порождающей ее задачей, администратор задач task создает ее и, отвечая, выдает порождающей задаче идентификатор (id) созданной задачи. В противном случае, никакого сообщения не посылается до тех пор, пока новая зада-

ча ни закончится сама по себе. В этом случае в ответе администратора задач будут содержаться конечные характеристики закончившейся задачи.

Сообщения отличаются количеством данных, которые передаются от одной задачи к другой. Данные копируются из адресного пространства первой задачи в адресное пространство второй, и выполнение первой задачи приостанавливается до тех пор, пока вторая задача не вернет ответное сообщение. В действительности обе задачи кратковременно взаимодействуют во время выполнения передачи. Ничто, кроме длины сообщения (максимальная длина 65535 байтов), не заботит QNX при передаче сообщения. Существует несколько протоколов, которые могут быть использованы для сообщений.

Основные операции над сообщениями — это *послать, получить* и *ответить,* а также несколько их вариантов для обработки специальных ситуаций. Получатель всегда идентифицируется своим идентификатором задачи, хотя существуют способы ассоциации имен с идентификатором задачи. Наиболее интересные варианты операций включают в себя возможность получать (копировать) только первую часть сообщения, а затем получать оставшуюся часть такими кусками, какие потребуются. Это может быть использовано для того, чтобы сначала узнать длину сообщения, а затем динамически распределить принимающий буфер. Если необходимо задержать ответное сообщение до тех пор, пока не будет получено и обработано другое сообщение, то чтение первых нескольких байтов дает вам компактный «обработчик», через который позже можно получить доступ ко всему сообщению. Таким образом, ваша задача освобождается от необходимости хранения большого количества буферов.

Другие функции позволяют программе получать сообщения только тогда, когда она уже ожидает их приема, а не блокироваться до тех пор, пока не прибудет сообщение, и транслировать сообщение к другой задаче без изменения идентификатора передатчика. Задача, которая транслировала сообщение, в транзакции невидима.

Кроме этого, QNX обеспечивает объединение сообщений в структуру данных, называемую очередью. Очередь – это просто область данных в отдельной задаче, которая временно принимает

передаваемое сообщение и немедленно отвечает передатчику. В отличие от стандартной передачи сообщений, передатчик немедленно освобождается, для того чтобы продолжить свою работу. Задача администратора очереди хранит в себе сообщение до тех пор, пока приемник не будет готов прочитать его, что выполняется путем запроса сообщения у администратора очереди. Любое количество сообщений (ограничено только возможностью памяти) может храниться в очереди. Они хранятся и передаются в том порядке, в котором они были приняты. Может быть создано любое количество очередей. Каждая очередь идентифицируется своим именем.

Помимо сообщений и очередей, в QNX для взаимодействия задач и организации распределенных вычислений имеются так называемые порты, которые позволяют формировать сигнал одного конкретного условия, и механизм исключений, о котором мы уже упоминали выше. Порт подобен флагу, известному всем задачам на одном и том же узле (но не на различных узлах). Он имеет два состояния, которые могут трактоваться как «присоединить» и «освободить», хотя пользователь может создать свою интерпретацию. Например, «занят» и «доступен». Порты используются для быстрой, простой синхронизации между задачей и обработчиком прерываний устройства. Они нумеруются от нуля до 32 (на некоторых типах узлов и больше). Первые 20 номеров зарезервированы для использования операционной системой.

С портом могут быть выполнены три операции:

1) присоединить порт;

2) отсоединить порт;

3) послать сигнал в порт.

Одновременно к порту может быть присоединена только одна задача. Если другая задача попытается «отсоединиться» от того же самого порта, то произойдет отказ при вызове функции и управление вернется к задаче, которая в настоящий момент присоединена к этому порту. Это самый быстрый способ обнаружить идентификатор другой задачи, подразумевая, что задачи могут договориться использовать один номер порта. Напомним, что все рассматриваемые задачи должны находиться на одном и том же узле. При работе нескольких узлов специальные функции обеспечивают большую гибкость и эффективность.

Любая задача может посылать сигнал в любой порт независимо от того, была ли она присоединена к нему или нет (предпочтительно, чтобы не была присоединена). Сигнал подобен неблокирующей передаче пустого сообщения (передатчик не приостанавливается, а приемник не получает какие-либо данные и только отмечает, что конкретный порт изменил свое состояние).

Задача, присоединенная к порту, может ожидать прибытия сигнала или может периодически читать порт. QNX хранит информацию о сигналах, передаваемых в каждый порт, и уменьшает счетчик после каждой операции приема сигнала («чтение» возвращает счетчик и устанавливает его в нуль). Сигналы всегда принимаются перед сообщениями, потому что имеют больший приоритет. В этом смысле сигналы часто используются обработчиками прерываний для того, чтобы оповестить задачу о внешних (аппаратных) событиях (действительно, обработчики прерываний не имеют возможности посылать сообщения и должны использовать сигналы).

В отличие от описанных выше методов, которые строго синхронизируются, исключения обеспечивают асинхронное взаимодействие (исключение может прервать нормальное выполнение потока задачи). Таким образом, исключения являются аварийными событиями. QNX резервирует 16 исключений, для того чтобы оповещать задачи о прерывании с клавиатуры, нарушении памяти и других необычных ситуациях. Остальные 16 исключений могут быть определены и использованы прикладными задачами.

Системная функция может быть вызвана для того, чтобы позволить задаче управлять своей собственной обработкой исключений, выполняя свою собственную внутреннюю функцию во время возникновения исключения. Заметим, что функция исключения задачи вызывается асинхронно операционной системой, а не самой задачей. Вследствие этого исключения могут иметь сильнодействующее побочное влияние на операции (например, передачу сообщений), которые выполняются в это же время. Обработчики исключений должны быть написаны очень аккуратно.

Одна задача может установить одно или несколько исключений на другой задаче. Это может быть комбинация системных исключений (определенных выше) и исключений, определяемых

приложениями, что обеспечивает другие возможности для межзадачного взаимодействия.

Благодаря такому свойству QNX, как возможность обмена посланиями между задачами и узлами сети, программы не заботятся о конкретном размещении ресурсов в сети. Это свойство придает системе необычную гибкость. Так, узлы могут произвольно добавляться и изыматься из системы, не затрагивая системные программы. QNX приобретает эту конфигурационную независимость благодаря применению концепции виртуальных задач. У виртуальных задач непосредственный код и данные, будучи на одном из удаленных узлов, возникают и ведут себя так, как если бы они были локальными задачами какого-то узла со всеми атрибутами и привилегиями. Программа, посылающая сообщение в сети, никогда не посылает его точно. Сначала она открывает «виртуальную цепочку», включающую все виртуальные задачи, связанные между собой. На обоих концах такой связи имеются буферы, позволяющие хранить самое большое послание из тех, которые цепочка может нести в данном сеансе связи. Сетевой администратор помещает в эти буферы все сообщения для соединенных задач. Виртуальная задача, таким образом, занимает всего лишь пространство, необходимое для буфера и входа в таблице задач. Чтобы открыть связь, необходимо знать идентификатор узла и задачи, с которой устанавливается связь. Для этого необходимо знать идентификатор задачи администратора, ответственного за данную функцию, или глобальное имя сервера.

**Поддержка стека протоколов TCP/IP в QNX**

Поддержка стека протоколов TCP/IP обеспечивается с помощью трех модулей, которые могут загружаться администратором сетевого ввода/вывода io-net:

`/lib/dll/npm-ttcpip.so` — облегченный стек TCP/IP для систем с ограниченными ресурсами, реализует часть функциональности полных реализаций стека TCP/IP;

`/lib/dll/npm-tcpip-v4.so` — стандартная реализация стека протоколов NetBSD vl.5;

`/lib/dll/npm-tcpip-v6.so` — профессиональный стек протоколов TCP/IP, KAME-расширение стека NetBSD vl.5.

Файл npm-tcpip.so — это просто ссылка на один из указанных модулей. Для обеспечения безопасности в модуле профессионального стека TCP/IP используется IPSec, который защищает IP-пакет от несанкционированного изменения посредством присоединения криптографической контрольной суммы, вычисленной односторонней хэш-функцией, или/и шифрует содержимое пакета криптографическим алгоритмом с закрытым ключом.

Для пакетной фильтрации (firewall) и трансляции сетевых адресов (NAT) используется перенесенный в QNX вариант IP Filter 3.4.27. Основу пакета IP Filter составляет модуль ipfilter.so, загружаемый администратором io-net. В состав IP Filter также входит ряд утилит:

ipf — утилита для изменения списка правил фильтрации;

ipfs — сохраняет и восстанавливает информацию для NAT и таблицы состояний;

ipstat — возвращает статистику пакетного фильтра;

ipmon — монитор для сохраненных в журнале пакетов; ipnat — пользовательский интерфейс для NAT.

Конфигурация сети TCP/IP хранится в файле /etc/net.cfg. Чтобы изменения, сделанные в файле, вступили в силу, необходимо запустить администратор конфигурирования TCP/IP — утилиту netmanager. Обычно для изменения настроек TCP/IP используют графическую утилиту phlip, которая автоматически запускает netmanager при сохранении изменений.

Для получения информации о TCP/IP в QNX содержится полный набор общепринятых UNIX-утилит.

**Технология Jump Gate**

Для обеспечения прозрачности в графической оболочке Photon имеется механизм Jump Gate Technology, основанная на использовании серверного процесса phrelay**,** который передает клиентским программам информацию о графическом изображении в Photon. Клиентами phrelay могут быть: phditto, phindows, phinx. Подключиться к серверу можно либо через последовательный канал, либо через сеть TCP/IP. Для запуска phrelay в TCP/IP обычно используется **inetd.** В стандартном файле /etc/ineta.conf уже есть в закомментированном виде нужная запись, поэтому достаточно просто раскомментировать ее:

```
phrelay stream tcp nowait root
/usr/bin/phrelay phrelay
```

Проверьте, чтобы файл /etc/services содержал строку

```
phrelay 4 8 6 8/tcp
```

Программы-клиенты кэшируют получаемую информацию, поэтому им достаточно получать данные только об изменениях «картинки».

При удаленном подключении к Photon microGUI по умолчанию создается дополнительная сессия. Чтобы подключиться к существующей сессии, необходимо указать имя Named Special Device-файла нужной сессии. Например, подключимся к текущей сессии Photon узла host1:

```
phditto -n /dev/photon hostl
```

Для доступа к phrelay из Windows используется клиентская программа Phindows.

Для доступа к phrelay из ЭВМ, использующей графическую среду X Window System независимо от аппаратной платформы и операционной системы, применяется QNX-утилита phinx.

Сервер phrelay может подключить клиента как к существующей сессии photon (тогда клиент получит полный доступ к запущенным приложениям), так и к отдельной, специально созданной сессии (тогда пользователи будут работать независимо в разных окружениях). Запретить или разрешить подключение к своей графической среде пользователь может, установив или сбросив флаг в окне утилиты phrelaycfg.

Для того чтобы передавать не все изображение рабочего стола, а только окно нужного приложения, можно использовать так называемые сервисы phrelay. Сервисы определяются в файле конфигурации `/etc/config/phrelay.имя узла` (если такого файла нет, то используется /etc/config/phrelay). Формат этого файла определен так:

```
service user [-W pwm_options] command
```

где `service` — произвольное символическое имя сервиса, которое передается клиентской утилите с опцией `-s`;

`user` определяет, как интерпретировать аргумент, передаваемый клиенту с опцией `-u`;

pwm_options — необязательные опции оконного менеджера pwm;

command — командная строка для запуска приложения Photon (ради которого, собственно, и затевалось все дело).

Особое внимание следует уделить, пожалуй, параметру `user`. Он может принимать такие значения:

- имя_пользователя (имеет смысл, если имя пользователя не задано посредством опции `-u`). Если для пользователя существует пароль, то будет запрошен ввод пароля;
- имя_пользователя:пароль (имеет смысл, если имя пользователя не задано посредством опции `-u`). Пароль не запрашивается;
- % — будут запрошены имя и пароль (имеет смысл, если не использована опция `-u`);
- ? — имя и пароль будут запрашиваться всегда (опция -u игнорируется);
- ! — возвращает ошибку, если не задана опция `-u`;
- =имя_пользователя — имя пользователя задается жестко (опция `-u` игнорируется);
- =имя__пользователя:пароль — имя пользователя и пароль задаются жестко (опция `-u` игнорируется).

Пример строки файла /etc/config/phrelay:

```
mylable ivan pfm
```

Теперь, выполнив команду `phindows -smylable`, мы получим такое окно, запущенное от имени пользователя ivan. Использование сервисов phrelay снижает сетевой трафик.

## Вопросы для самопроверки

1. Назовите назначение администратора ресурсов в ОС QNX.
2. Приведите классификацию файловых систем ОС QNX.
3. Расскажите о реализации файловых систем в ОС QNX.

4. Для чего используются инсталляционные пакеты и репозитарии в ОС QNX?

5. Какие символьные устройства ввода/вывода существуют в QNX?

6. Приведите структуру сетевой подсистемы ОС QNX.

7. Какие сетевые протоколы поддерживаются ОС QNX?

8. Что представляет собой технология FLEET?

# 7. РАБОТА В QNX

## 7.1. Начальная загрузка QNX Neutrino

При включении питания ЭВМ, на которой инсталлирована ОСРВ QNX, запускается процесс начальной загрузки. Начальная загрузка QNX протекает поэтапно [11]:

1) выполнение кода начального загрузчика IPL (Initial Program Loader). IPL осуществляет минимальное конфигурирование аппаратуры (конфигурирует контроллер памяти, системные часы и т. п.) поиск на носителе и загрузку в ОЗУ загружаемого образа ОС и передает ему управление;

2) выполнение кода загрузчика ОС (startup), входящего в состав загрузочного образа QNX. Загрузчик startup копирует и, если нужно, разжимает загрузочный образ QNX, определяет состав и конфигурацию аппаратуры, заполняет системную страницу данных ОС и передает управление модулю procnto, который также входит в состав загрузочного образа QNX;

3) выполнение модуля procnto (микроядро Neutrino + администратор процессов). Модуль procnto запускает остальные процессы, входящие в состав загрузочного образа QNX.

В QNX обычно используются два загрузочных образа: основной образ помещается в файл /.boot, а резервный — в файл /.altboot. Выбрать загрузку резервного загрузочного образа можно при выполнении вторичного загрузчика startup. Загрузчик startup предлагает нажать клавишу ESC для загрузки резервного образа: Hit Esc for .altboot, иначе загружаться будет основной образ.

Модуль procnto запускает последовательно все процессы, входящие в состав загрузочного образа. Одним из процессов стандартного образа является diskboot. Перед началом работы diskboot предлагает нажать «пробел» для ввода опций загрузки:

```
Press the space bar to input boot options.
```

Если ничего не нажимать, то загрузка продолжится в автоматическом режиме. Если нажать «пробел», то на экране появится сообщение (приведем сразу как первичный английский текст, так и русское обозначение):

F1Safe modes. Загружаться в «безопасном режиме».

F5Start a debug shell after mounting file-systems. Запустить командный интерпретатор после подключения файловых систем.

F6Be Verbose. Установить режим подробного вывода диагностических сообщений.

F7Mount read-only partitions read/write if possible. Попытаться подключить с возможностью записи разделы, доступные только для чтения.

F8Enable a previous package configuration. Вернуться к предыдущей конфигурации пакетов.

F9Target output to device defined in starup code. Выводить диагностическую информацию на устройство, указанное в модуле startup.

F10 Force a partition install. Приступить сразу к инсталляции системы[1].

F11 Enumerator disables. Отключить автоматическое распознавание устройств.

F12 Driver disables. Отключить драйверы.

Enter Continue boot process. Продолжить загрузку.

Please select one or more options via functional keys. Selection? Выберите, пожалуйста, одну или более опций, нажав соответствующие функциональные клавиши. Ваш выбор?

Далее процесс diskboot сканирует аппаратуру. Если найдено более одного раздела с данными, то предлагается выбрать раздел, который будет корневым. Последним действием diskboot является вызов сценария /etc/system/sysinit, задача которого — запуск процессов, обеспечивающих необходимую функциональность ОС:

1) запуск сервиса регистрации системных событий slogger;

2) запуск администратора неименованных каналов pipe;

3) если это первый запуск системы после инсталляции, то запускается сценарий /etc/rc.d/rc.setup-once. Факт первого

[1] Клавиша F10 доступна только при загрузке с инсталляционного компакт-диска.

запуска устанавливается по отсутствию файла начальной конфигурации базовой системы /etc/system/package/packages. Сценарий создает ряд каталогов: /tmp, /var, /pkgs, /root (домашний каталог системного администратора), а также файл для свопинга. Если существует файл /boot/setup.inf (в нем сохраняются настройки, выполненные в процессе инсталляции), то запускается сценарий /etc/rc.d/rc.setup-info. Затем перезапускается администратор пакетной файловой системы fs-pkg и генерируется начальная поисковая база данных программы-просмотрщика электронной документации helpviewer. В этом сценарии также создается рабочая копия файла /etc/passwd;

4) установка часового пояса. Информация берется из файла /etc/TIMEZONE;

5) запуск командного сценария /etc/rc.d/rc.rtc для настройки часов реального времени;

6) определение имени ЭВМ. Информация берется из файла /etc/HOSTNAME;

7) запуск командного сценария /etc/rc.d/rc.devices. Этот скрипт инициирует распознавание аппаратных устройств. Запускается администратор псевдотерминалов devc-pty, затем определяются каталоги, содержащие информацию о поддерживаемых устройствах. После этого запускается администратор конфигурирования аппаратуры enum-devices, сканирующий подключенные устройства.

Если существует файл /etc/system/config/useqnet и запущен администратор сетевого ввода/вывода io-net, то загружается администратор сетевого протокола Qnet. Администратор протокола Qnet реализован в виде DLL, расширяющей функциональность администратора io-net.

Если существует файл /.swapfile, то он подключается в качестве устройства свопинга;

8) запуск командного сценария /etc/rc.d/rc.sysinit. Этот скрипт продолжает инициализацию системы (выполняет настройки, специфичные для данной ЭВМ, и запускает необходимые сервисы). Запускается процесс dumper, сохраняющий core-файлы процессов, завершившихся аварийно. Затем запуска-

ется сценарий `/etc/rc.d/rc.local` (если он существует). Этот сценарий нужен, если есть необходимость добавить свои команды инициализации, не редактируя созданные системой файлы. Последнее действие сценария `rc.sysinit` — запуск программы инициализации терминала `tinit`. Эта программа запускает на терминале утилиту входа в систему `login` или графическую оболочку Photon с графической утилитой входа в систему `phlogin`. Вариант запуска определяется по наличию или отсутствию файла `/etc/system/config/nophoton`. Если не удается запустить `rc.sysinit`, то делается попытка запустить командный интерпретатор Korn Shell в интерактивном режиме. Если стандартный интепретатор не может запуститься, делается попытка запустить интерпретатор с меньшими требованиями к ресурсам Fat EmbeddedShell (fesh).

## 7.2. Графический интерфейс пользователя Photon microGUI

По аналогии с операционной системой QNX, графическая среда Photon microGUI представляет собой графическое микроядро с семейством процессов, расширяющих функциональность графического микроядра (или «графического сервера») и взаимодействующих посредством стандартного QNX-механизма передачи сообщений. Само графическое микроядро Photon представляет собой небольшой процесс, реализующий только несколько фундаментальных примитивов, которые используются внешними процессами для выполнения различных задач пользовательского интерфейса.

***Сервер Photon*** не работает с окнами, не взаимодействует с устройствами ввода (мышью, клавиатурой и т. п.). За непосредственную прорисовку изображения отвечает ***графический драйвер***. Задача графического драйвера — отображение информации, получаемой от сервера шрифтов и от интерпретатора ***графического потока***. За ввод информации при помощи мыши, клавиатуры и других устройств отвечает ***драйвер ввода***.

Следует заметить, что Photon наследует сетевую прозрачность ОС QNX — дополнительные графические драйверы могут использоваться для расширения графического пространства Photon за счет физических дисплеев других узлов сети. При этом можно легко обеспечить дублирование изображения.

Photon использует кодировку Unicode, что обеспечивает ввод и вывод текста, написанного на разных языках.

Для того чтобы обеспечить полнофункциональную графическую среду, позволяющую пользователям манипулировать окнами приложений, изменением их размеров, перемещением, сворачиванием и т. п., используется ***оконный менеджер***. Он же поддерживает панель задач. Рабочий стол с меню быстрого запуска приложений реализован с помощью процесса ***администратора рабочего стола***.

Из сказанного видно, что полная функциональность графического интерфейса пользователя ОС QNX достигается набором процессов и DLL, расширяющих возможности графического сервера Photon. Поэтому легко конфигурировать системы с ограниченными ресурсами — нужно просто выбрать только необходимые компоненты. Кроме того, облегчается перенос драйверов видеокарт и устройств ввода.

**Реализация графической среды**

В обычной настольной системе Photon может запускаться одним из двух способов:

1) вручную из командной строки в любое время после регистрации в системе;

2) автоматически утилитой tinit, если не существует файла `/etc/config/system/nophoton`.

В любом случае используется командный сценарий `/usr/bin/ph`, выполняющий запуск всех необходимых компонентов графической среды в зависимости от конфигурации системы.

Сценарий состоит из этапов последовательных действий.

1. Запуск утилиты зондирования графического оборудования `crttrap`; если такое оборудование найдено и опознано, то `crttrap` запускает программу `devgt-iographics` для определения доступных графических режимов видеокарты. Программа

devgt-iographics записывает результаты своей работы в файл /etc/system/config/graphics-modes.

2. Запуск графического сервера Photon. Если существует переменная окружения LOGNAME (а это означает, что вы ее либо нарочно инициализировали в командных скриптах, либо вы уже прошли регистрацию с помощью утилиты login), то Photon стартует сразу. В противном случае Photon запустит утилиту phiogin для регистрации пользователя в системе. Можно запретить пользователю выход из Photon в командную строку, присвоив переменной PHEXIT_DISABLE значение 1.

3. Запуск администратора графического вывода io-graphics программой crttrap с использованием командной строки из файла /etc/system/config/graphics-modes. Процесс io-graphics загружает интерпретатор графического потока gri-photon.so и сервер шрифтов. По умолчанию io-graphics загружает сервер шрифтов, реализованный в виде разделяемого объекта pnfont.so. Можно указать администратору графического вывода запуск сервера шрифтов в виде отдельного процесса phfont, создав переменную окружения PHFONT_USE_EXTERNAL. Отдельный процесс сервера шрифтов может понадобиться для того, чтобы его можно было использовать с других узлов сети. Заметим, что в дистрибутиве QNX есть несколько серверов шрифтов с различными ограничениями функциональности для встраиваемых систем.

4. Запуск процесса inputtrap для зондирования устройств ввода. Он определяет, с какими аргументами необходимо запускать драйвер-администратор графического ввода devi-hirun и запускает его. Результат своей работы inputtrap сохраняет в файле /etc/config/trap/input.имя_узла.

5. Запуск утилиты fontsieuth, указывающей серверу шрифтов phfont, в каких каталогах находятся шрифты.

6. Запуск процессов bkgdmgr (рисует фон рабочего стола), wmswitch (позволяет переключаться между открытыми окнами приложений, используя комбинацию клавиш «Alt-Tab»), saver («хранитель экрана»). Кроме того, сам сервер Photon запускает оконный менеджер pwm и администратор рабочего стола shelf.

**Утилиты конфигурирования**

Компоненты Photon можно конфигурировать с помощью нескольких утилит.

Для того чтобы изменить настройки оконного менеджера используется утилита. Вкладка `Background` утилиты `pwmopts` позволяет настраивать параметры администратора фона `bkgdmgr`.

Для настройки подсистемы шрифтов предназначена утилита `fontadmin`. Эта утилита позволяет задавать необходимые псевдонимы для инсталлированных шрифтов. Настройка администратора графического вывода `io-grafics` выполняется с помощью программы phgrafx. В окне этой утилиты можно задать используемый видеодрайвер, значения разрешения, глубины цветности и частоты обновления экрана.

Настройка мыши может выполняться с помощью утилиты `input-cfg`.

Разрешить либо запретить удаленное подключение к локальной сессии Photon можно с помощью утилиты `phrelaycfg`.

Установить необходимый хранитель экрана, время отсутствия сигналов ввода, через которое срабатывает хранитель экрана, пароль хранителя можно с помощью утилиты `savecfg`.

## 7.3. Печать в ОС QNX

**Традиционная система печати (lpd)**

Эта система широко известна в UNIX-подобных ОС. И, как следствие, достаточно хорошо документирована. Поэтому мы сделаем лишь краткий ее обзор.

Система печати `lpd` условно делится на три части:

1) сервер печати (спулер) `lpd`;

2) файл конфигурации принтеров `/etc/printcap`;

3) клиентские утилиты `lpr, lprq, lprrm, lprc`.

Самой многофункциональной из всех утилит системы `lpd` является `lprc`. Она позволяет выполнять все то, что делают другие утилиты, плюс выполняет активизацию/деактивизацию принтеров, управляет очередью и делает другую полезную работу.

Утилита `lpr` выполняет постановку задания в очередь для печати. Утилитой `lprq` можно просматривать очередь заданий. Удалить задание из очереди можно утилитой `lprrm`.

Данная система достаточно старая и многие поставщики ОС предлагают более удобные решения.

**Собственная система печати QNX**

Основой системы печати QNX является серверный процесс-администратор `spooler`. Назначение spooler — это обеспечение бесконфликтного доступа нескольких пользователей к контролируемому им устройству. Этот процесс автоматически запускается администратором нумерации устройств `enum-devi-ces`. По умолчанию spooler контролирует доступ к параллельному порту /dev/par1. Если к параллельному порту подключить принтер, то spooler автоматически распознает его и путем зондирующих запросов получает параметры принтера.

В каталоге `/etc/printers` содержится несколько файлов конфигурации spooler для разных типов принтеров. При запуске spooler выполняет несколько операций:

- определяет тип принтера и выбирает соответствующий этому типу файл конфигурации;
- сравнивает свойства, указанные в файле конфигурации, с параметрами принтера, полученными при его сканировании;
- регистрирует в пространстве имен префикс-каталогов `/dev/printers/имя_принтера/`. В этом каталоге создаются каталог `spool/`, файлы устройств `phs, raw` и файл, соответствующий типу принтера;
- создает каталог `/var/spool/printers/имя_принтера.имя_хоста/`, в который отображается содержимое `/dev/printers/имя_принтера /spool`;
- в каталог `spool` записывает файл с результирующими настройками принтера.

Когда пользователь выдает задание на печать, соответствующий файл помещается в каталог `spool`, и `spooler` вызывает необходимые фильтры для обработки файла. Результатом обработки является файл в формате, понятном принтеру. Конечный файл посылается в устройство печати.

В составе дистрибутива ОС QNX поставляется несколько фильтров для наиболее популярных наборов принтеров. Кроме того, в состав ОС QNX входит Printer DDK (Driver Development Kit), представляющий собой подробно комментированный пример исходного кода фильтра с инструкцией для разработчиков.

Для управления заданиями можно воспользоваться фотоновской утилитой prjobs.

## 7.4. Средства анализа

QNX — операционная система жесткого реального времени. Это значит, что она в состоянии обеспечить выполнение приложений в условиях критического лимита времени. Однако человеку свойственно ошибаться, и приложения не всегда ведут себя так, как ожидается. Поэтому у разработчика и у администратора обязательно должны быть эффективные инструменты анализа и диагностики.

**Инструментальный комплект для анализа системы**

В состав QNX Momentics PE входит пакет System Analysis Toolkit (SAT). SAT позволяет отслеживать:

- вызовы микроядра;
- передачу сообщений;
- обработку прерываний;
- изменения состояний потоков.

Основу этого пакета составляет модуль `procnto-instr` — администратор процессов, с которым скомпоновано микроядро, оборудованное средствами диагностики.

В операционной системе происходят различные события, все они, так или иначе, проходят через микроядро. Средства, входящие в инструментальное расширение, сохраняют трассу событий в буферах ядра в виде записей. Для сброса информации из буферов в файл используется утилита `tracelogger`**.** Для увеличения скорости этой операции трасса записывается в двоичном формате, поэтому для конвертирования данных в читабельную форму следует или воспользоваться утилитой `traceprinter`**,** или написать свою программу обработки журнала трассировки

(в составе SAT есть специальная библиотека `libtraceparser.a` — SAT API**).** Использовать SAT можно следующим образом:

1) в файле построения загрузочного образа заменить имя модуля `procnto` на `procnto-instr`;

2) собрать заново загрузочный образ с помощью утилиты `mkifs`;

3) полученный образ сохранить в файле `/.boot` системы;

4) включить в состав целевой системы утилиты `trace-iogger` и `traceprinter`.

5) перезагрузить целевую систему и выполнить трассировку событий микроядра.

**Средства ведения журналов событий**

В ОС QNX реализованы две системы ведения журналов событий: syslog и slogger.

Система syslog традиционно используется в различных UNIX-подобных ОС. В QNX она включена в основном для обеспечения переноса программных продуктов из других операционных систем. Основу syslog составляет серверный процесс syslogd, считывающий при старте конфигурационный файл `/etc/syslog.conf`. Информация для записи в журнал (например, в файл `/var/log/syslog` — имя задается в файле конфигурации) поступает от клиентов, вызывающих для этой цели соответствующие функции. Из командных сценариев информация может посылаться с помощью утилиты `logger`. Кроме журнального файла, `syslogd` может писать информацию непосредственно на экран или передавать для обработки процессу `syslogd`, запущенному на другой хост-машине сети. Сервисный процесс `syslogd` записывает в журнал:

- дату и время возникновения события;
- идентификатор узла сети, на котором событие произошло;
- имя/идентификатор процесса/пользователя, запросившего регистрацию события;
- текстовое сообщение (определяется разработчиком программы-клиента).

Штатная QNX-система ведения журналов событий `slogger` проще и эффективнее системы `syslog`.

Основу системы составляет серверный процесс `slogger`, регистрирующий по умолчанию префикс `/dev/slog`. Этот файл устройства и является журналом событий. Процессы могут посылать серверу `slogger` информацию для записи в журнал с помощью специальных функций. Информация сохраняется в виде записей определенного формата:

- дата и время возникновения события;
- «уровень серьезности» сообщения (от 0 до 7);
- старший и младший коды события (значения старших кодов определены в файле `/usr/include/sys/slogcodes.h`);
- текстовое сообщение (определяется разработчиком программы-клиента).

Для просмотра журнала используется утилита `sloginfo`.

**Получение информации об оборудовании**

Для проверки работы оборудования в QNX есть несколько утилит:

nicinfo — информация о работе сетевой карты;

pin — выдает информацию об устройствах PC Card;

pci — выводит информацию о всех PCI-устройствах;

crttrap — выполняет автоопределение видеокарты;

inputtrap — выполняет автоопределение устройств ввода (клавиатура, мышь и т. п.).

## 7.5. Среда визуальной разработки программ

Операционная система QNX (в реализациях RTP 6.XX) содержит в своём составе инструментарий визуального проектирования приложений Photon Application Builder (PhAB).

PhAB, по сути дела, не является IDE в привычном смысле (да и к QNX существуют несколько развитых IDE от сторонних производителей): он не содержит собственного редактора исходных текстов, символьного отладчика и, самое главное, — не генерирует некоторый «проект», в том смысле, как это делают, например, CodeForge, MS Visual CPP или Borland Builder. Это, скорее, построитель GUI-образов приложения: PhAB избавляет

разработчика от рутинной работы по отслеживанию размеров, взаимных положений, цветов и множества других характеристик GUI-компонентов приложения и позволяет привязать обрабатывающий код (или другие widget[1]) к GUI-событиям (нажатие кнопок, перемещение окон, ввод с клавиатуры). Весь программный код реакции на события разработчик прописывает традиционными методами на C/C++.

Результатом работы генератора PhAB является достаточно понятная структура файлов заголовков, определений и т. д. и, главное, традиционных Makefile, которые могут далее обрабатываться традиционно средствами утилиты make. Показательной является, например, возможность переноса первоначально сгенерированного в PhAB приложения в CodeForge, генерации проекта на основании Makefile и последующего развития в Code-Forge. Такое сочетание визуального проектирования GUI-компо-нентов приложения с возможностями задания большого числа разнообразных механизмов связывания событий GUI с программным кодом, а также сочетание с традиционным написанием программного кода даёт, как это ни кажется странным, очень высокую итоговую результативность разработки. Другими несомненными достоинствами PhAB являются простота адаптации и интуитивная понятность происходящего. В результате программист с хорошим знанием языка C++, но никогда не сталкивавшийся с QNX, вполне готов создавать приложения в PhAB после 3–4-х часов работы с системой.

**Создание проекта**

Для создания проекта первое, что нам нужно сделать, — это запустить Photon Appli-cation Builder. Производится выбор в системном меню «Launch – Development – Builder» или возможен запуск PhAB командной строкой `/usr/photon/appbuilder/ab`, например, из окна терминала. В любом случае, должно получиться окно приложения PhAB, что-то подобное тому, что показано на рис. 7.1.

[1] Widget в терминологии Photon, да и многих графических экранных систем в UNIX, принято обобщённо называть любой отдельный графический компонент: кнопку, окно, диалог и т. д.

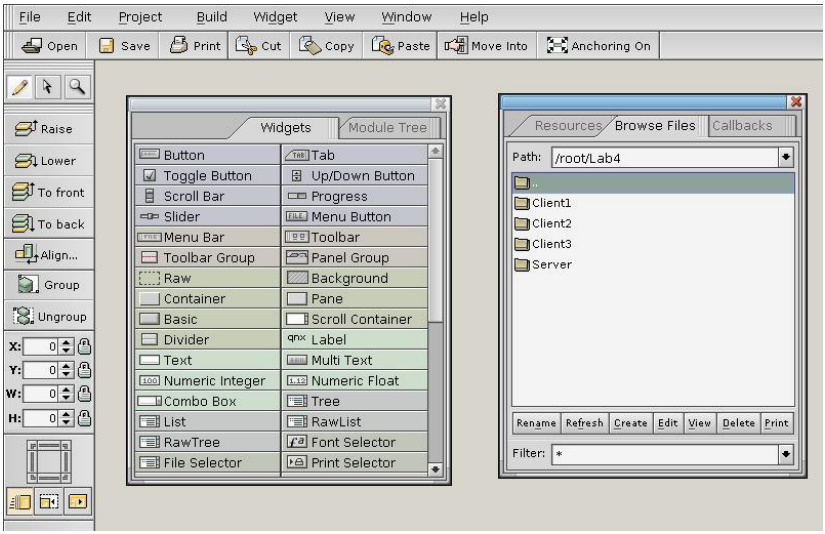


Рис. 7.1. Демонстрация приложения PhAB

Первоначально мы должны создать для своего приложения главный widget-контейнер. В терминологии PhAB требуемый нам widget-контейнер — это модуль.

Для создания главного модуля приложения воспользуемся меню «File-New», где представлен на выбор список из 13 типовых видов окна главного модуля приложения. PhAB обеспечивает работу с модулями следующих типов: window, dialog, menu, icon, picture и др. В качестве главных модулей приложения используются первые два. Можно выбрать тип, например Plain, позже можно его поменять и произвольно расширить.

Выбрав тип модуля приложения, пользователь получает изображение этого модуля на экране, и модулю по умолчанию будет присвоено имя base. Имя любого widget в проекте можно посмотреть или сменить на закладке «Resources» (см. рис. 7.1), отметив требуемый widget в окне проекта. При создании нового widget в проекте (сверх начального окна приложения) ему будет присво-ено стандартное имя, совпадающее с наименованием типа wid-get. Это начальное имя может быть сохранено за widget только в том случае, когда для него не определяется какая-либо

специфическая реакция на события GUI (пассивный widget). В противном случае вы обязаны присвоить widget произвольное уникальное имя на закладке «Resources».

После определения главного модуля приложения целесообразно сразу же определить некоторые его параметры, воспользовавшись меню «Application-Startup Info Modules».

*Фактическое создание проекта в PhAB* происходит при первоначальном выборе из меню «File-Save As …» для нового проекта. При этом PhAB предлагает вам определить имя нового проекта (и тем самым его местоположение в файловой системе), как это показано на рис. 7.2. — новый проект назван «xxx», и он будет помещён во вновь созданный каталог $HOME/xxx.

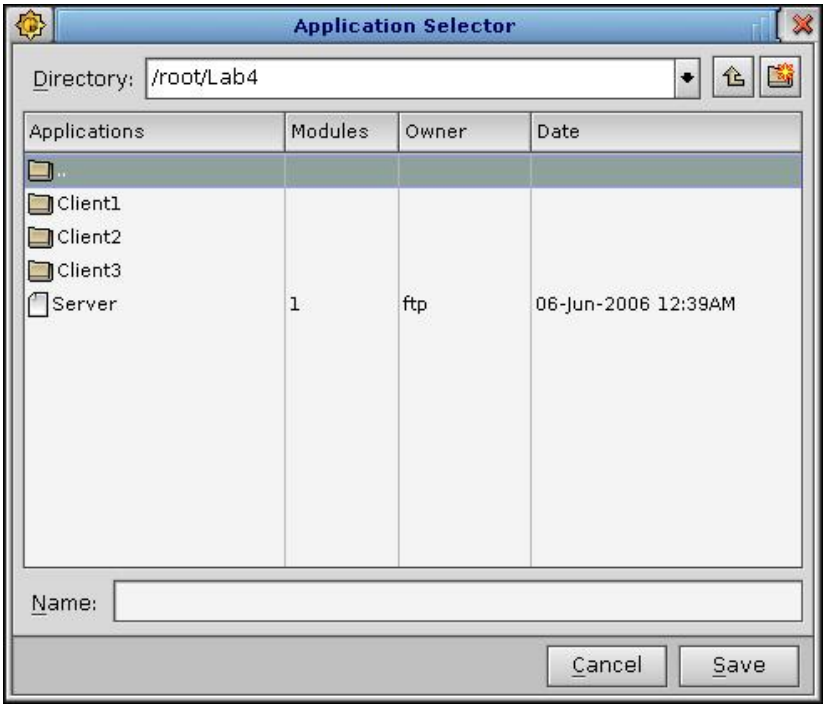


Рис. 7.2. Демонстрация сохранение проекта в PhAB

Далее необходимо произвести начальную генерацию проекта, воспользовавшись для этого меню «Application-Build+Run» (для первой фазы этого процесса — генерации — можно воспользоваться «Application-Generate»).

Пользователю необходимо выполнять «Generate» для вновь созданного проекта. Необходимость «Generate» неоднократно возникает при любых изменениях во внешнем виде GUI: появлении новых widget, изменении размеров, положений (но необязательна при коррекции C++ программного кода, не затрагивающей widget или взаимодействия widget с кодом).

При генерации QNX предлагает выбрать аппаратную платформу. Данная операционная система реализует поддержку около десятка процессорных платформ. Как правило, в учебных целях система инсталлирована с поддержкой только платформы x86. В окне выбора следует согласиться с предложенным «gcc».

После выбора процессорной платформы последовательно выполняются «Generate» и «Make» — после генерации файлов проекта и их компиляции пользователь получает работающее бинарное приложение с именем xxx (совпадающим с именем проекта) в каталоге ./src/ntox86. Проверить новое приложение можно, выбрав «Run» в меню построения проекта.

Собственно, вся дальнейшая работа над приложением — это придание ему функциональности, для чего пользователь должен написать соответствующие участки программного кода и связать его с событиями GUI (нажатия кнопок, ввод текстовых полей, выбор из селективного списка и т. д.).

Каждый widget (сложное окно-контейнер, или простейшая кнопка) может генерировать реакцию на события GUI-образов: нажатие кнопок, перемещение окон и изменение размеров. Часто события, вызывающие реакцию, — это действия пользователя, но не всегда, более того, это только малая их часть. Событиями являются и фазы «внутренней жизни» widget: запуск приложения, момент прорисовки окна на экране, истечение времени таймаута и т. д. (хотя для таких «внутренних» событий в их длинных цепочках всё равно «спусковым механизмом» где-то в начале цепочки является действие пользователя).

С помощью PhAB-визуального построителя можно определить реакцию в программе, которую вызовет событие widget.

Это достигается регистрацией «callback» при создании приложения. Определяются callback на закладке «Callback» построителя при выборе определяемого widget.

Существуют callback нескольких классов, но основные из них два: «widget» — открытие другого окна при наступлении события, «code» — функция в программном коде, которая вызывается при наступлении события (другие классы могут быть построены как комбинация этих).

Callback класса «widget» может порождать другое окно (по имени этого widget) при наступлении события, которое, в свою очередь, может по своему созданию порождать следующий widget и т. д., создавая целую цепочку порождаемых widget. В частности, такой callback может порождать новую копию того же widget, который реагирует механизмом callback на событие, на манер рекурсивных вызовов функций, при этом может создаваться целая цепочка однотипных GUI-компонентов. Естественно, и в этом есть определённая опасность, что, как и при рекурсии, необходимо обеспечить завершение этого процесса, например по числу созданных копий.

Callback класса «code» (наверное, самый широкоиспользуемый) — это вызов программной функции по событию, с передачей функции в качестве параметров характерных особенностей события. После выполнения очередной перегенерации PhAB добавит в программный файл «заглушку» — функцию с заказанным вами при определении callback именем. Это важно, потому что параметры callback-функций, которые несут информацию о событии, имеют достаточно сложную структуру. Заметим здесь, кстати, что при удалении или переопределении callback в проекте, когда необходимость в функции с заданным именем отпадает, PhAB не может удалить текст callback-функций из программного кода, и вы должны будете удалить его вручную.

Для одного и того же события вы можете определить несколько реакций, часть из которых могут быть класса «widget», а другие — класса «code». Тогда все они отработаются поочередно при наступлении события (а иначе не было бы никакой возможности псевдорекурсивного создания widget, как описывалось выше: кто-то должен считать созданные компоненты). Бо-

лее того, вы можете переопределять последовательность срабатывания этих callback в списке.

Для многих событий вы можете определить срабатывание callback непосредственно до или после их наступления. С другой стороны, каждый класс widget имеет набор так называемых «ресурсов» — текущих свойств класса widget. Конкретный набор ресурсов определяется видом класса widget: ресурсы кнопки существенно отличаются от ресурсов текстового поля ввода. Ресурс определяет внешний вид класса widget на экране: цвет, размер, текст, отображаемый в поле ввода-вывода и т. д.

Пользователь из программного кода всегда имеет возможность прочитать или изменить значение любого ресурса любого (созданного, прорисованного на экране) класса widget. Для этого используется большая группа функций GetResource(s)–SetResource(s), составляющих основу программирования под Photon.

Все widget, которые вы добавляете в приложение Photon, упорядочиваются в древовидную иерархию, которая определяет и иерархию видимости widget. Изменить положение каждого widget в иерархии можно просто «перетаскивая» его в дереве на закладке «Module Tree».

Освоившись с логикой модели приложения PhAB, строить приложения становится весьма просто. Весь процесс происходит как многократная последовательность (для каждого нового widget) нижеперечисленных шагов.

1. На закладке «Widgets» PhAB выбираем требуемый тип добавляемого widget. Выбранный widget «перетаскиваем» (мышью) в окно того контейнера, где ему надлежит быть по логике приложения. Можно таким же образом подкорректировать (сейчас или позже) положение и размер устанавливаемого widget. Но точное местоположение и размер (с точностью до 1-го пикселя) лучше устанавливать с помощью наборных полей "X" "Y" "W" "H" в левой части окна PhAB).

2. Для установленного widget на закладке «Callback» для требуемых событий устанавливаем требуемые реакции (чаще всего класса «code»). Общий синтаксис записи имени требуемой функции реакции на событие (то же относится и к функциям инициализации приложения и его главного окна) выглядит так:

Class::Function@File.Type, где любой компонент записи, кроме «Function», может быть опущен. Компонентами записи являются:

- Function — имя функции обработчика callback. После перегенерации проекта в ваш программный код будет добавлена заготовка функции с таким именем и требуемым списком параметров;
- File — имя файла исходного кода программы, в который будет помещён обработчик. Если исходный файл с таким именем не существует в проекте, то он будет создан. Если вы не указываете имя файла, то PhAB создаст файл с именем, совпадающим с именем функции обработчика, и поместит его туда (но эта идея «на каждое событие — отдельный файл программы обработчика» не кажется самой удачной).
- Type — расширение имени файла («c» или «cpp»). Если тип не указан, предполагается C. Это поле определяет, синтаксис какого языка (C или C++) задаст PhAB компилятору gcc. Во избежание недоразумений, особенно при работе с C++, лучше это поле указывать явно;
- Class — используется только с C++, и определяет класс, функция-член Function которого будет использована в качестве обработчика. В качестве таких функций могут быть использованы только static-члены, которые должны быть определены вне зависимости от создания конкретных объектов этого класса.

После выполнения очередного «Generate» из окна построения приложения в программный код будет добавлен шаблон функции Function с требуемым набором параметров.

3. Дописываем в Function требуемый по логике приложения программный код. Из этого кода можно «добираться» к указателю вызвавшего реакцию widget, или любого другого, в приложении, и изменять его ресурсы, используя GetResource(s) — SetResource(s).

4. Выбираем «Generate» — «Make» — «Run» из окна построения приложения и проверяем, достигли ли мы желаемого эффекта.

*Примечание:* созданное приложение может запускаться не только по кнопке «Run» из окна построения приложения, но и обычной командой запуска по имени приложения из окна pterm.

***Функции, используемые для работы с сообщениями*:**

*PtGetResource() //* взять данные по ресурсу из компоненты формы, например, из поля для ввода текста изъять сам текст:

```
#define PtGetResource(widget, type, value, len)...
```

где `widget` — название ресурса (в данном случае — название поля, компонента, в который вводится сообщение, посылаемое клиентом серверу);

`type` — тип ресурса (Pt_ARG_COLOR, Pt_ARG_TXT);

`value` — адрес, по которому отправляется сообщение, либо записываемая переменная;

`len` определяется в зависимости от типа ресурса, здесь это длина посылаемого сообщения.

Для того чтобы взять текст, посланный сервером клиенту в ответ на его сообщение, и поместить в окно редактирования ввода, необходимо использовать функцию *SetResource()* (установить ресурс для данного элемента формы (например, для поля ввода текста):

```
#define PtSetResource(widget, type, value, len)...
```

**Пример использования функции *SetResource()***

```
PtWidget_t *widget;
PtSetResource( widget, Pt_ARG_FILL_COLOR,
Pg_BLUE, 0 );
```

Обе функции возвращают значение 0 при удачной работе и значение –1 при возникновении ошибки.

## Вопросы для самопроверки

1. Опишите основные шаги при первоначальной загрузке ОС QNX.
2. Расскажите о командном сценарии /usr/bin/ph.
3. Какие системы печати существуют в ОС QNX?
4. Для чего используется пакет System Analysis Toolkit?
5. Какие средства для ведения журналов существуют в ОС QNX?
6. Опишите процесс создания приложений в Photon Application Builder.

## Литература

1. Бэкон Д. Операционные системы / Д. Бэкон, Т. Харрис. — СПб.: Питер; Киев: BHV, 2004. — 800 с.

2. Верхалст Э. Задача разработки ОСРВ для цифровой обработки сигналов / Э. Верхалст // Мир компьютерной автоматизации. — 1997. — № 4. — С. 23-31.

3. Timmerman M. RTOS Evaluation Kick Off! / M. Timmerman, B. Van Beneden, L. Uhres // Real-Time Magazine. — 1998. — N 3. — P. 6–10.

4. Алексеев Д. Практика работы с QNX / Д. Алексеев, Е. Ведревич, А. Волков и др. — М.: КомБук, 2004. — 432 с.

5. Liu C. L. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment / C. L. Liu, James W. Layland // Journal of the Association for Computing Machinery. — 1973. — Vol. 20. — N I.

6. Sha Lui. Goodenough. Rate Monotonic Analysis for Real-Time Systems / Lui Sha, Mark H. Klein, B. John // Technical Report CMU/ SEI-91-TR-6. — 1991. —March. — Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania.

7. Zalewski J. What Every Engineer Needs To Know About Rate-Monotonic Scheduling: A Tutorial / J. Zalewski // Real-Time Magazine. — 1995. — N 1. — P. 6–24.

8. Keeling N.J. Missed it! — How Priority Inversion messes up real-time performance and how the Priority Ceiling Protocol puts it right / N.J. Keeling // Real-Time Magazine. — 1999. — N 4. — P. 46–50.

9. Золотарев С. Lynx OS-178 — коммерческая ОСРВ для авиации [Электронный ресурс]: публикация сайта. — М.: PCweek online, 2005. — Режим доступа к сайту: http://kis.pcweek.ru/Year 2005/N22/CP1251/Industrial_built-in/chapt1.htm

10. Гордеев А.В. Системное программное обеспечение / А.В. Гордеев, А.Ю. Молчанов. — СПб.: Питер, 2002. — 736 с.

11. Зыль С.Н. Операционная система реального времени QNX: от теории к практике / С.Н. Зыль. — СПб.: БХВ-Петербург, 2004. — 192 с.

Учебное издание

**Гриценко Юрий Борисович**

Системы реального времени

Учебное пособие

Редактор Коновалова Н.В.
Корректор Полещук О.В.

Подписано в печать 09.08.06. Формат 60x84/16. Бумага офисная.
Печать трафаретная. Гарнитура Times New Roman.
Усл. печ. л. ???. Уч.-изд. л. ???. Тираж 100. Заказ ???.

Томский государственный университет систем управления
и радиоэлектроники. 634050, Томск, пр. Ленина, 40. Тел. (3822) 53-30-18.