QNX MOMENTICS

ОСНОВЫ ПРИМЕНЕНИЯ

Зыль Сергей Николаевич, к. т. н., специалист по информационной безопасности в системах жесткого реального времени, с 2001 года преподает разработку и администрирование для операционных систем семейства QNX.

Перед Вами пособие, предназначенное для разработчиков встраиваемых систем реального времени, не знакомых с технологиями и решениями QNX. В основу книги положены материалы учебного курса "Основы администрирования и разработки приложений реального времени для ОСРВ QNX Neutrino", разработанного и читаемого автором в учебном центре компании SWD Software Ltd. (http://www.swd.ru), аттестованном в системе QNX Approved Trainer.

Книга позволит Вам понять логику работы операционной системы QNX Neutrino, состав и назначение инструментария QNX Momentics Professional Edition, а также некоторые приемы, используемые для разработки специализированного программного обеспечения.

Компакт-диск содержит ряд полезных программ для QNX Neutrino: популярные СУБД, файловые менеджеры, текстовые редакторы, игры, библиотеки для обработки данных различных форматов и многое другое.

ISBN 5-94157-430-4

БХВ-Петербург
190005, Санкт-Петербург, Измайловский пр., 29
E-mail: mail@bhv.ru
Internet: www.bhv.ru
тел.: (812) 251-42-44
факс: (812) 251-12-95

bhv

QNX MOMENTICS: ОСНОВЫ ПРИМЕНЕНИЯ

bhv

СЕРГЕЙ ЗЫЛЬ

QNX MOMENTICS

ОСНОВЫ ПРИМЕНЕНИЯ

- Архитектура QNX Neutrino RTOS
- Инструментарий QNX Momentics
- Сетевые протоколы Qnet и TCP/IP
- Создание загружаемых образов
- Трассировка событий ядра

+CD-ROM

bhv

Глава 1. Начинаем работу в QNX Neutrino

Любишь кататься — люби и катайся!
(Народная мудрость)

Эта глава предназначена для тех, кто первый раз в жизни видит не только QNX, но и UNIX-подобную систему, поэтому здесь рассмотрены следующие вопросы:

- инсталляция резидентной среды разработки QNX Momentics и дополнительного программного обеспечения;
- вход в систему, конфигурирование операционной системы QNX Neutrino и ее графической оболочки Photon microGUI после инсталляции;
- начальные сведения о принципах работы пользователя в операционной системе QNX Neutrino, а также о создании простейших программ на языке C.

1.1. Инсталляция QNX Momentics

Процедура инсталляции комплекта разработчика QNX Momentics описана в инструкции производителя, прилагающейся к инсталляционным компакт-дискам продукта. Тем не менее, как показывает практика, нелишним будет немного прокомментировать эту нехитрую в общем-то процедуру.

1.1.1. Общие сведения

Прежде чем приступить к установке QNX Momentics, следует определиться, на какой аппаратно-программной платформе мы хотим работать. Momentics можно развернуть на одной из четырех программно-аппаратных платформ:

- в операционной среде Solaris (только SPARC-версии);
- в операционной системе Windows XP (только x86-версии);
- в операционной системе RedHat Linux (только x86-версии);
- в операционной системе QNX Neutrino (только x86-версии).

Производители всех этих операционных систем публикуют списки поддерживаемого оборудования — ознакомьтесь с ними *до* того, как приступите к инсталляции QNX Momentics.

Независимо от выбранной инструментальной платформы комплект разработчика QNX Momentics можно использовать для создания приложений, работающих в ОСРВ QNX Neutrino на нескольких аппаратных платформах — ARM (включая StrongARM и XScale), MIPS, PowerPC, SH-4 и x86.

Поскольку в России наибольшей популярностью пользуется резидентная (по-аглицки — self-hosted, иногда пишут Neutrino-hosted) разработка, то мы будем рассматривать именно ее. Если выбрана другая операционная система, то сначала следует установить и настроить выбранную ОС в соответствии с указаниями фирмы-производителя, а затем установить QNX Momentics, пользуясь инструкцией по инсталляции, прилагаемой к соответствующему компакт-диску.

1.1.2. Процесс инсталляции QNX Momentics (Neutrino-hosted)

ОС QNX инсталлируется в собственный раздел диска. Перед инсталляцией (а лучше — до покупки "железа") ознакомьтесь со списком аппаратуры, протестированной QSS на совместимость с QNX. Этот список находится на сайте компании QSS (**http://www.qnx.com**). В любом случае, для использования QNX Momentics Professional Edition на ЭВМ под управлением QNX Neutrino требуется:

- процессор — не ниже Pentium III 700 МГц (желательно Pentium 4 2 ГГц);
- ОЗУ — не меньше 256 Мбайт (желательно 512 Мбайт);
- свободное дисковое пространство — не меньше 1,5 Гбайт.

Требования к процессору и ОЗУ вызваны значительной ресурсоемкостью интегрированной среды разработки (IDE, Integrated Development Environment), написанной на языке Java. Требования к дисковому пространству связаны с тем, что в состав QNX Neutrino SDK входят компоненты для построения целевых систем для нескольких аппаратных платформ. Если вам не нужна IDE, то потребность в аппаратных ресурсах будет меньше.

В процессе инсталляции операционной системы QNX Neutrino выполняются следующие действия:

1. На жестком диске создается раздел QNX.
2. Базовые средства ОС копируются на жесткий диск.
3. Устанавливаются средства, необходимые для разработки целевых систем QNX Neutrino и приложений для разных аппаратных платформ — QNX Neutrino SDK.
4. Устанавливается интегрированная среда разработки.
5. Устанавливаются GNU-утилиты для целевых систем.

> ***Примечание***
> Прежде чем вы приступите к инсталляции, хочу предупредить во избежание недоразумений — я описываю процедуру инсталляции QNX Neutrino с компакт диска QNX Momentics Professional Edition ver. 6.3.0 (Neutrino hosted) от 12 мая 2004 года. Какой дистрибутив будет прилагаться к книге, и будет ли вообще прилагаться какой-нибудь диск — я не знаю и на момент написания книги, к сожалению, точно знать не могу.

Итак, для начала инсталляции загрузите свой компьютер с компакт-диска QNX Neutrino.

> ***Примечание***
> Если ЭВМ не поддерживает загрузку с компакт-диска, то на компакт-диске можно найти файл instflop.dat, представляющий собой образ загрузочной инсталляционной дискеты. В любой UNIX-подобной системе такую дискету можно создать командой вроде следующей:
>
> ```
> dd if=instflop.dat of=/dev/fd0
> ```
>
> Для Windows тоже есть подобные утилиты, например rawrite.

Если ничего не нажимать и тестирование устройств EIDE пройдет успешно, то через некоторое время на экране появится сообщение:

```
Please select a boot option. Option F2 is great for testing
QNX compatibility on new hardware without writing anything to
the hard disk. It can also be used for system recovery.
F2 – Run from CD (Hard Disk filesystems mounted under /fs)
F3 – Install QNX to a new disk partition
Select?
```

(Выберите, пожалуйста, вариант загрузки. Выбор клавиши <F2> удобен для проверки совместимости QNX с новой аппаратурой без записи какой-либо информации на диск. Этот вариант можно также использовать для восстановления системы после сбоя.

<F2> — загрузить систему непосредственно с компакт-диска (файловые системы монтируются в /fs);

<F3> — установить QNX в новый раздел диска.

Ваш выбор?)

Если нажать клавишу <F2>, то QNX загрузится с компакт-диска. При этом будет предложено изменить настройки видеорежима. Инсталляция QNX выполняться не будет.

Нажмите клавишу <F3>. На экране появится сообщение:

```
This installation will create a QNX partition on your hard
disk and create a bootable QNX Neutrino image. You may abort
this installation at any prompt by pressing the F12 key.
Press F1 to continue.
Press F2 to set verbose (debug) mode.
Choice (F1, F2)?
```

(Во время инсталляции будут созданы раздел на жестком диске и загрузочный образ QNX. Процесс инсталляции можно прервать в любой момент нажатием клавиши <F12>.

Нажмите клавишу <F1> для продолжения инсталляции.

Нажмите клавишу <F2> для перехода в режим подробного вывода информации (отладочный режим).

Ваш выбор (<F1>, <F2>)?)

В дальнейшем мы будем рассматривать инсталляцию в обычном режиме. Чаще всего этого вполне достаточно. Итак, нажмите клавишу <F1>.

В любом случае на экране появится приглашение ввести лицензионный ключ (license key), который указан в лицензионном сертификате, поставляемом с компакт-диском:

```
Please enter your license key:
```

Аккуратно напечатайте требуемый номер и нажмите клавишу <Enter>.

Если вы ошибетесь, система вежливо сообщит об этом печальном обстоятельстве:

```
** Invalid or expired license key entered. **
```

Если же введен верный номер, то на экране появится текст лицензии и вам нужно будет сделать мучительный выбор — принять или отвергнуть ее условия. Итак:

```
F1 - accept   F2 - reject
```

(<F1> — принять условия; <F2> — отвергнуть условия)

Если нажать клавишу <F2>, то инсталляция прекратится, программа-инсталлятор предложит извлечь компакт-диск QNX из дисковода и перезагрузить ЭВМ. Если вы согласны с условиями лицензии, то нажмите клавишу <F1>. На экране появится сообщение:

```
Please enter the disk your would like to install QNX Neutrino
on. The disk must be bootable from your BIOS.
```

(Выберите, пожалуйста, диск, на который вы бы хотели установить QNX Neutrino. Этот диск должен загружаться с помощью BIOS вашей ЭВМ.)

За этой фразой следует список обнаруженных дисков. Каждая строка списка начинается с названия функциональной клавиши, которую нужно нажать для установки ОС на соответствующий диск. На моей ЭВМ установлен только один диск, поэтому выбора у меня нет — жму клавишу <F1>. В ответ получаю сообщение:

```
*** WARNING***
You have a disk which is greater than 8.4 Gigabytes. The
original BIOS calls to access the disk are unable to read data
above 8.4G. If your BIOS is older then 1998 you may be forced
to choose option 2. Newer BIOSes support on extended disk read
calls which can access the entice drive.
F1 Allow the QNX partition to be anywhere on the disk.
F2 Keep the QNX partition below 8.4G.
Choice (F1, F2)?
```

(Объем диска на вашем компьютере превышает 8,4 гигабайта. Обычные функции BIOS, предназначенные для доступа к дискам, не могут читать данные, расположенные за пределами 8,4 Гбайт. Если ваша BIOS произведена до 1998 года, то вам следует нажать клавишу <F2>. Более новые BIOS поддерживают расширенные вызовы чтения, позволяющие работать с любыми дисками.

<F1> — раздел QNX может находиться в любом месте диска;

<F2> — раздел QNX должен находиться в пределах 8,4 Гбайт.

Ваш выбор (<F1>, <F2>)?)

Надеюсь, что ваш компьютер выпущен после 1998 года ☺. В противном случае вам нужно нажать клавишу <F1>, чтобы избежать аппаратных конфликтов при работе с жесткими дисками.

Жму клавишу <F1>. Эх, инсталлятор обнаружил, что на моей машине установлена QNX Neutrino 6.2.1B:

```
You already have a QNX partition (type 97). You may delete it,
in which case all files will be lost. If you choose not to
delete the partition, the installation will be aborted because
QNX uses this partition type.
```

```
F1 Abort          F2 Delete
```

(На вашей ЭВМ уже есть QNX-раздел типа 97. Можете удалить его, при этом все файлы будут утрачены. Если вы не выберете удаление раздела, то инсталляция будет прекращена, потому что QNX использует этот тип раздела.

<F1> Прервать инсталляцию <F2> Удалить раздел)

Смело жмем клавишу <F2> (если, конечно, все нужные данные сохранены). Следующее сообщение будет зависеть от того, сколько свободного места имеется на диске. У меня оно выглядит так:

```
Your disk has room for a 9766 megabyte QNX partition?
Please select the size of the partition you would like to
create for QNX Neutrino.
F1            all           9766 M
F2            half          4883 M
F3            quarter       2441 M
F4            eighth        1220 M
F5 Display partition table, so you can delete an existing
partition.
```

(На диске есть 9766 мегабайт свободного пространства для раздела QNX. Выберите размер, который вы хотели бы выделить для QNX Neutrino.

<F1>	все место	9766 Мбайт
<F2>	половина	4883 Мбайт
<F3>	четверть	2441 Мбайт
<F4>	одна восьмая	1220 Мбайт

<F5> — показать таблицу разделов, чтобы можно было удалить существующий раздел.)

Вы можете использовать под раздел QNX либо все свободное место на диске, либо его часть. При необходимости, нажав клавишу <F5>, можно удалить какой-нибудь из существующих разделов диска. Я выбрал вариант с клавишей <F3> — этого вполне достаточно для QNX Neutrino, и на экране появилось сообщение:

```
You have more then one partition on your hard disk. You need a
special partition boot loader to let you choose which
partition to load when you boot. Your choices are:

F1 Install the QNX partition boot loader. When you boot, it
   will prompt you to select which partition (OS) to boot. If
   your partition may start above 8.4G, we recommend this
   choice.
```

```
F2 Install the QNX partition boot loader for machines with old
   BIOS (before 1996/1997). It should not be used with drives
   greater then 8.4 G.

F3 Use your existing boot loader, which may already provide
   this capability. Examples include System Commander or LILO.
   If it does not provide this capability you will be able to
   boot only the currently set active partition. If your
   partition starts above 8.4G, this existing loader will need
   to use the new extended BIOS disk calls.

Choice (F1, F2, F3)?
```

(На вашем жестком диске больше одного раздела. Чтобы можно было выбирать, какой из них использовать при загрузке, необходим специальный загрузчик. У вас есть варианты:

<F1> — установить "родной" загрузчик QNX. Он будет выдавать приглашение выбрать раздел (т. е. ОС) для загрузки. Если раздел QNX находится за пределами 8,4 Гбайт, то мы рекомендуем этот вариант;

<F2> — установить "родной" загрузчик QNX для компьютеров со старой BIOS (до 1996–1997 г выпуска). Его нельзя использовать с дисками объемом свыше 8,4 Гбайт.

<F3> — оставить прежний загрузчик. QNX могут загружать "не родные" загрузчики, например System Commander или LILO. Однако если такой загрузчик не установлен, то вы сможете загружать только ОС, установленную в активном на данный момент разделе диска. Если раздел QNX находится за пределами 8,4 Гбайт, то прежний загрузчик должен поддерживать новые вызовы BIOS для дисков.

Ваш выбор (<F1>, <F2>, <F3>)?)

Лично меня вполне устраивает загрузчик QNX, однако некоторых пользователей он не удовлетворяет, т. к. не имеет графического интерфейса. Что ж, такие пользователи могут поэкспериментировать с мультизагрузчиками третьих производителей.

Вот и все. Программа инсталляции получила информацию, достаточную для установки как самой операционной системы, так и инструментов разработчика. На самом деле, процесс инсталляции может несколько отличаться от описанного выше в зависимости от используемого компьютера (например, на вашей ЭВМ есть несколько жестких дисков, или программа-инсталлятор обнаружила уже готовый раздел QNX). Но в главном последовательность действий сохраняется. Итак, нажимаем клавишу <F1> (установить загрузчик QNX) — на экране появляется сообщение:

```
Restarting driver and mounting filesystems ...
Copying files to the new QNX partition. Please wait ...
```

(Перезапускается драйвер и монтируются файловые системы ...

Файлы копируются в новый раздел QNX. Пожалуйста, подождите …)

В процессе копирования файлов на экране в реальном времени отображается, сколько процентов файла уже скопировано:

```
100% Copying core files to hard disk
100% COPY /cd/boot/fs/qnxbase.ifs to /hdisk/boot/fs
100% COPY /cd/boot/fs/qnxbase.ifs to /hdisk/.altboot
100% COPY /cd/boot/fs/qnxbasedma.ifs to /hdisk/boot/fs
100% COPY /cd/boot/fs/qnxbasedma.ifs to /hdisk/.boot
100% COPY /cd/boot/fs/qnxbasesmp.ifs to /hdisk/boot/fs
Recording licenses … complete
```

Те, кто уже работал с предыдущими версиями QNX Neutrino, вероятно заметили, что базовая часть ОС (core files) копируется на жесткий диск непосредственно, qfs-файл не используется.

По окончании копирования файлов на экране появляется сообщение:

```
QNX Momentics 6.3.0 Development Seat Installation
-------------------------------------------------
Throughout installation, you will be prompted for necessary
information. The default answers are displayed in brackets
after the question. If you press the <Enter> key, the default
answer will be used. After typing your answer, press the
<Enter> key to continue the installation.
```

(Инсталляция инструментальной среды QNX Momentics 6.3.0

В ходе инсталляции у вас будет запрошена необходимая информация. Ответы по умолчанию показаны в квадратных скобках после вопроса. Если вы нажмете клавишу <Enter>, то будет использован ответ по умолчанию. Напечатав ответ, нажмите клавишу <Enter> для продолжения инсталляции.)

Затем система начнет задавать эти самые вопросы:

```
Do you wish to install to the default location of
`/usr/qnx6301` (y/n)? [y]
```

(Хотите ли вы выполнить установку в назначенный по умолчанию каталог /usr/qnx630 (да/нет)? [да])

Соглашаюсь (нажимаю клавишу <Enter>).

```
Do you wish to install the QNX Neutrino SDK (All Targets)
(y/n)? [y]
```

(Хотите ли вы установить QNX Neutrino SDK (для всех целевых платформ) (да/нет)? [да])

Если не согласиться, то на этом инсталляция и завершится — будет предложено извлечь CD-ROM из привода и нажать любую клавишу для перезагрузки. Загрузившись, мы сможем-таки доустановить средства разработки, для этого надо будет поместить в привод инсталляционный CD-ROM, и запустить инсталляционный сценарий вручную:

```
/fs/cd0/install
```

Нам зададут те же самые вопросы, на которые мы не захотели отвечать сразу. Нам все же придется согласиться установить QNX Neutrino SDK для всех целевых архитектур.

> ***Примечание***
> Кстати, QNX Neutrino SDK — по сути, единственная составляющая QNX Momentics Standard Edition.

Следующий вопрос:

```
Do you wish to install the Integrated Development Environment
(y/n)? [y]
```

(Хотите ли вы установить интегрированную среду разработки (да/нет)? [да])

Конечно, мы хотим установить интегрированную среду разработки! Собственно, наличие IDE отличает Professional Edition от Standard Edition (нажимаем клавишу <Enter>).

Затем инсталлятор сообщает, что у нас есть возможность инсталлировать для целевых систем пакет программ, распространяемых под условием лицензии GNU GPL (General Public License, генеральной общедоступной лицензии GNU[1]), который содержит такие полезные утилиты, как GNU tar и GNU sed:

```
Do you wish to install the GNU Public License Utility package
(y/n)? [y]
```

(Хотите ли вы установить пакет утилит, распространяемых на условиях лицензии GNU GPL (да/нет)? [да])

Можем согласиться (нажать клавишу <Enter>). Но помните, что условия лицензии GPL отнюдь не безобидные.

Система сообщит, что она начала инсталляцию выбранных составных частей и предложит немного подождать:

---

[1] Аббревиатура раскрывается рекурсивно: "GNU's Not Unix" — "GNU — это не UNIX".

```
Installing qnx-host ... please wait.
Installing qnx-target ... please wait.
Installing qnx-qde ... please wait.
Installing qnx-target-gpl ... please wait.
```

Далее сообщается, в каких файлах содержится информация о том, что же мы установили, и выводится сообщение о завершении инсталляции с просьбой извлечь инсталляционный CD-ROM и нажать любую клавишу для перезагрузки ЭВМ:

```
** Installation complete **

Please remove the installation media then press 'Enter' to
reboot
```

Вынимаем CD-ROM и нажимаем какую-нибудь клавишу. Начинается первая загрузка QNX Neutrino после установки.

1.1.3. Первая загрузка QNX Neutrino после инсталляции

После включения питания ЭВМ, на которой инсталлирована ОСРВ QNX Neutrino, мультизагрузчик предлагает выбрать ОС для загрузки. (Вернее, он предлагает выбрать раздел диска, с которого следует выполнять загрузку.) После инсталляции QNX Neutrino раздел QNX будет активным и загрузится по умолчанию, если ничего не нажимать:

```
Press F1-F4 for select drive or select partition
1,2,3,4? 1
```

(Нажмите клавишу <F1>, <F2>, <F3> или <F4> для выбора дисковода или выберите раздел 1, 2, 3 или 4? 1)

Единица после вопросительного знака — это вариант, предлагаемый по умолчанию.

Следует заметить, что в QNX обычно используется два загрузочных образа: основной образ помещается в файл `/.boot`, а резервный — в файл `/.altboot`. Загрузчик предлагает нажать клавишу <Esc> для загрузки резервного образа операционной системы:

```
Hit Esc for .altboot
```

Если ничего не нажимать, то загружаться будет основной образ. Устанавливаемая с компакт-диска QNX Neutrino сконфигурирована так, что после загрузки образа и инициализации микроядра запускается утилита **`diskboot`**, выполняющая значительную часть работы по загрузке системы. Эта утилита предложит следующее:

```
Press the space bar to input boot options or D to disable
DMA...
```

(Нажмите клавишу <пробел> для ввода опций загрузки или клавишу <D>, чтобы запретить DMA[1])

Если ничего не нажимать, загрузка продолжится в автоматическом режиме. Если нажать клавишу <пробел> то на экране появится сообщение:

```
F1      Safe modes
F5      Start a debug shell after mounting filesystems
F6      Be Verbose
F7      Mount read-only partitions read/write if possible
F8      Enable a previous package configuration
F9      Target output to debug device defined in startup code
F10     Force a partition install
F11     Enumerator disables
F12     Driver disables
Enter   Continue boot process
Please select one or more options via functional keys.
Selection?
```

(<F1> — загружаться в "безопасном режиме";

<F5> — запустить командный интерпретатор после подключения файловых систем;

<F6> — установить режим подробного вывода диагностических сообщений;

<F7> — попытаться подключить с возможностью записи разделы, доступные только для чтения;

<F8> — вернуться к предыдущей конфигурации пакетов;

<F9> — выводить диагностическую информацию на устройство, указанное в модуле `startup`;

<F10> — сразу приступить к инсталляции системы;

<F11> — отключить автоматическое распознавание устройств;

<F12> — отключить драйверы;

<Enter> — продолжить загрузку.

Выберите, пожалуйста, одну или более опций, нажав соответствующие функциональные клавиши.

Ваш выбор?)

---

[1] DMA — Direct Memory Access (прямой доступ к памяти, ПДП).

Нажатие клавиши <F1> позволяет использовать несколько вариантов загрузки системы с ограниченной функциональностью. Опция <F10> имеет смысл только при загрузке с инсталляционного компакт-диска — без этой опции утилита **diskboot**, кроме инсталляции, предложит еще загрузить систему с одного из обнаруженных разделов QNX (на компакт-диске или на жестком диске). Опция <F11> позволяет отключить автоопределение некоторых типов устройств, а <F12> — отключить некоторые типы драйверов.

Выяснив, как загружать систему, **diskboot** ищет раздел QNX на жестком диске. Таких разделов, разумеется, может быть несколько. Утилита **diskboot** должна выбрать, какой из этих разделов сделать основным (т. е. "корневым"). Возможность использования раздела в качестве корневого **diskboot** определяет по наличию в этом разделе файла `/.diskroot`. Если найдено больше одного раздела с файлом `/.diskroot`, то будет предложено выбрать, какой из этих разделов назначить корневым:

```
You have more then one .diskroot file which wants to mount at
/
F1   /dev/hd0t78
F2   /dev/hd0t79
Which one do you wish to mount?
```

На моей ЭВМ в разделе `/dev/hd0t78` установлена ОС QNX Neutrino 6.2.1A, и утилита **biskboot** это определила. Нажимаем клавишу <F2>. Раздел `/dev/hd0t79` смонтируется как `/`, но и раздел `/dev/hd0t78` смонтируется куда-нибудь в каталог `/fs` (не пропадать же добру!).

Последним действием утилиты **diskboot** является вызов сценария `/etc/system/sysinit`, запускающего ряд других сценариев из каталога `/etc/rc.d` (если захотите добавить в процедуру загрузки свои команды, то добавляйте их в файл `/etc/rc.d/rc.local`). Появится сообщение о сканировании оборудования, а также сообщения об операциях, однократно выполняемых при первой загрузке (это выполняется сценарий `/etc/rc.d/rc.setup-once`):

```
Generating helpviewer search index
```

(Создание индекса для ускорения поиска информации в электронной документации)

Кстати, напоследок сценарий `/etc/rc.d/rc.setup-once` создаст файл `/etc/system/setupisdone`, по наличию которого система в

следующий раз поймет, что она загружается не в первый раз и что запускать сценарий `rc.setup-once` больше не надо.

После этого (речь все еще о первом запуске) система переключается в графический режим и выводит окно для первоначального конфигурирования графического драйвера. Вам необходимо задать четыре параметра:

- видеодрайвер;
- разрешение экрана;
- глубину цвета;
- частоту обновления экрана.

Кроме того, конфигуратор позволяет выбрать — запускать автоматически графическую оболочку Photon microGUI при каждой загрузке системы или нет (я из-за природной лени обычно указываю системе запускать графическую среду автоматически). Впрочем, в процессе эксплуатации системы эту настройку всегда можно изменить.

После выбора нужных значений параметров, нажимаем кнопку **Test** (тестовый режим) — это позволит изменить неудачные настройки. Как только получим приемлемое изображение, сразу нажимаем кнопку **Continue** (продолжить). Если в тестовом режиме не нажать эту кнопку, то система автоматически переключится на прежние настройки через 15 с. Заданные значения параметров можно будет снова изменить в любое время после загрузки системы.

Затем QNX предложит ввести имя пользователя и пароль. В свежеустановленной системе уже есть несколько зарегистрированных системных псевдопользователей и один вполне реальный — суперпользователь, имеющий неограниченные полномочия в операционной системе. Это вы ☺. Этот пользователь имеет имя `root` и пока не имеет пароля. Вводим имя, игнорируем запрос на ввод пароля — и (поздравляю!) мы в QNX Neutrino. Инсталляция и первая загрузка операционной системы успешно завершены.

1.1.4. Инсталляция дополнительного программного обеспечения

Разнообразные дополнительные программы для QNX Neutrino, как коммерческие, так и свободно распространяемые, часто поставляются в виде специальных пакетов.

Для установки таких пакетов предназначена программа QNX Software Installer (**`qnxinstall`**) (рис. 1.1), которую можно запустить через меню кнопки **Launch**.

Рис. 1.1. Окно программы QNX Software Installer

Программе-инсталлятору все равно, где находится репозитарий или отдельный пакет — в сети или на локальной ЭВМ. На сайте QSS (**http://www.qnx.com**) содержатся сведения о некоторых общедоступных репозитариях с программами для QNX Neutrino. Для того чтобы не задавать инсталлятору информацию о них вручную, можно в меню **File** выбрать элемент **Find Web Repositories**. Инсталлятор загрузит с сервера QSS информацию об известных репозитариях. Если щелкнуть на названии репозитария, то инсталлятор скачает с соответствующего сервера информацию о доступных пакетах. Самый содержательный репозитарий — 3rd-Party Software, имеющийся на сайте компании QSS[1]. Щелкнем на его названии. Инсталлятор начнет скачивать из Интернета архив со сведениями о пакетах, размещенных на сервере. Для примера установим флажок рядом с именем пакета AbiWord (x86) и нажмем кнопку **Install** (рис. 1.2).

Рис. 1.2. Установка текстового редактора AbiWord (версия для x86)

Обратите внимание на состояние пакета — значение **New** в столбце **Status** свидетельствует, что пакет еще не установлен в нашей системе.

Далее следуем указаниям инсталлятора. Сначала инсталлятор сообщит, какие пакеты и какого размера он собирается установить, затем предложит принять/отклонить условия лицензий. Некоторые пакеты могут потребовать ввести лицензионный ключ. Поскольку пакет скачивается из Сети, в зависимости от размеров пакетов и скорости вашего канала время выполнения установки может быть разным. По окончании инсталляции флажок в строке с названием пакета будет снят, а состояние (**Status**) изменится с **New** на **Active**.

[1] Этот репозитарий находится также на компакт-диске "QNX Momentics 3d Party Software".

1.2. Конфигурирование

В этом разделе рассматриваются регистрация пользователя в системе, конфигурирование сети, настройка русификации и некоторые дополнительные настройки.

1.2.1. Регистрация пользователя в системе

Итак, вход пользователя в инсталлированную систему (правильнее сказать, регистрацию пользователя в системе) можно выполнить двумя способами — из командной строки или через графический интерфейс. Способ, который система предложит, зависит от того, был ли установлен соответствующий флажок при настройке видеодрайвера при первом старте QNX Neutrino после инсталляции *(см. ранее)*. Если флажок не был установлен (т. е. операционной системе запрещено автоматически запускать графическую оболочку Photon), то в каталоге `/etc/system/config` был создан пустой файл с именем `nophoton`. Во время загрузки системы стартовые сценарии просто-напросто проверяют, существует ли файл `nophoton`. Создавая и удаляя этот файл, можно указывать — вручную или автоматически должна запускаться графическая оболочка Photon.

При командно-строковой регистрации, если введены правильные имя пользователя и пароль (а при первом входе, как вы помните, мы используем имя `root` без пароля), то стартует командный интерпретатор shell. Этот интерпретатор называют *входным* (login shell).

> ***Примечание***
> Подробнее о командном интерпретаторе рассказывается в *разд. 1.3.1*.

В нем можно запускать различные команды, в том числе, можно запустить Photon командой **`ph`** (при этом вводить пароль не потребуется).

Кстати, каким образом графическая оболочка Photon определяет — требовать у вас регистрацию или нет? Очень просто: она проверяет значение переменной окружения `LOGNAME`. Если вы зарегистрировались посредством **`login`**, то переменная `LOGNAME` будет содержать ваше регистрационное имя. Если же выполнялся автоматический запуск Photon, то, разумеется, значение переменной `LOGNAME` еще не задано. Отсюда вытекает интересная возможность для встроенных систем с графическим интерфейсом — можно в стартовых сценариях

принудительно задать для переменной LOGNAME значение и сразу запускать Photon от имени какого-либо пользователя.

Для выхода из системы есть следующие способы:

- в графической среде — выбрать элемент **Log out** меню **Launch** (запустится утилита **phshutdown**);
- в login shell — выполнить команду **logout** или **exit**;
- выполнить команду **shutdown**.

Утилиту **shutdown** может использовать только суперпользователь (пользователь root). Она посылает сигнал SIGPWR ("сейчас будет отключено питание") всем выполняющимся процессам, десять секунд ожидает их завершения и перезагружает систему. Если команде **shutdown** указать опцию **-f** (т. е. fast — "быстро"), то время ожидания будет сокращено до одной секунды.

Возможно несколько вариантов поведения утилиты **shutdown**, задаваемых опцией **-S**:

- **system** — остановить систему для выключения питания;
- **reboot** — перезагрузить систему;
- **photon** — закрыть графическую оболочку Photon и продолжать работу в "черном экране";
- **user** — закончить пользовательский сеанс. При этом на экране появится регистрационное приглашение для входа в систему.

Если существует файл /var/log/wtmp, то утилита **shutdown** добавит в него запись о своей работе.

Утилита **phshutdown** по своей функциональности аналогична утилите **shutdown**. Она тоже посылает сигнал SIGPWR всем выполняющимся процессам, десять секунд ожидает их завершения и перезагружает систему, однако вызывать ее имеет право любой пользователь. Чтобы запретить использование этой утилиты всем пользователям, кроме root, необходимо создать пустой файл /usr/photon/config/phshutdown.restrict. Если существует файл /var/log/wtmp, то утилита **phshutdown** добавит в него запись о своей работе.

Итак, **phshutdown** предлагает выбрать один из трех вариантов действий:

- завершить сеанс Photon;

- остановить систему. К сожалению, в QNX не реализовано автоматическое выключение блока питания. Поэтому после появления на экране сообщения о том, что ОС QNX завершила работу, вам следует, как в старое доброе "до-ATX-овое" время, нажать кнопку Power;
- перезагрузить систему.

Итак, будем считать, что мы вошли в систему и так или иначе запустили графическую оболочку. Мы оказываемся в так называемом *рабочем пространстве* (workspace) Photon. Как работать в графической оболочке — интуитивно понятно. Внизу расположена панель задач, в левой ее части находится кнопка **Launch**. Нажатие кнопки **Launch** открывает меню, позволяющее вызывать различные прикладные программы. В правой части экрана находится меню быстрого запуска приложений — **Shelf** ("полка", рис. 1.3).

Рис. 1.3. Меню **Shelf** быстрого запуска приложений

Следует обратить внимание на следующие "часто используемые" программы:

- Photon Terminal (**`pterm`**) — штатный псевдотерминал, позволяющий работать с инструментами командной строки;
- Helpviewer — программа доступа к штатной электронной документации. Надо сказать, что в составе QNX поставляется достаточно обширная и подробная документация;
- Photon File Manager (**`pfm`**) — штатный файловый менеджер. Для любителей файловых менеджеров а-ля Norton Commander есть продукт "третьего" производителя **`mqc`** (MiShell QNX Commander), работающий, как ему и положено, в "черном экране" или в псевдотерминале (эта программа есть в репозитарии 3rd-Party Software).

1.2.2. Конфигурирование сети

Для нормальной работы следует выполнить несколько простых операций. Если ЭВМ подключена к сети, то нам потребуется знать IP-адрес и сетевую маску нашей машины, IP-адреса шлюза и сервера имен. Если вы — системный администратор, то эти сведения вам известны, в противном случае обратитесь к системному администратору. Заметим,

что задать хотя бы имя хоста и его IP-адрес желательно даже для одиночного компьютера, потому что многие программы изначально рассчитаны для работы в сети и работают корректно только при наличии сетевых настроек.

Итак, запустим программу конфигурирования сети, нажав кнопку **Network** в разделе **Configure** панели в правой части окна (меню **Shelf**) и вручную (**Manual**) зададим IP-адрес и маску подсети нашей машины в открывшемся окне **TCP/IP Configuration** (рис. 1.4).

Рис. 1.4. Привязка IP-адресов к сетевым интерфейсам

При желании можно указать, чтобы ОС QNX Neutrino запрашивала параметры для своих сетевых интерфейсов у определенного DHCP[1]-сервера. Разумеется, DHCP-сервер должен быть доступен и правильно сконфигурирован. Кстати, DHCP-сервер может работать и на QNX Neutrino (читайте описание программы **dhcpd** в Utilities Reference).

Теперь перейдем на вкладку **Network** и введем имена ЭВМ (**Host Name**) и домена (**Domain Name**), IP-адреса шлюза (**Default Gateway**) и серверов имен (**Name Servers**) (рис. 1.5).

Рис. 1.5. Ввод имен хоста и домена, IP-адресов шлюза и DNS-серверов

Кстати, имя хоста, которое здесь введено (в данном случае — `myname`), будет использоваться не только протоколами TCP/IP, но и собственным протоколом QNX — Qnet. Замечу, что никаких других настроек для Qnet не требуется.

Если требуется задать дополнительные IP-адреса интерфейсов и шлюзов, то щелкните на гиперссылке **Click here to toggle the display of advanced options such as network routing**.

1.2.3. Настройка локализации

Теперь можно перейти к настройке локализации. Конечно, локализация в QNX далека от совершенства, но кое-что есть. Надо сказать, что есть три объекта русификации:

---

[1] DHCP (Dynamic Host Configuration Protocol) — протокол динамической конфигурации хостов.

- графическая оболочка Photon microGUI;
- графический псевдотерминал **pterm**;
- консоль ("черный экран").

Поскольку в Photon microGUI используется кодировка Unicode, кириллические шрифты уже, по сути дела, есть. Нужно настроить только раскладку клавиатуры. Для этого на "полке справа" (см. рис. 1.3) в уже знакомом разделе **Configure** нажмем кнопку **Localization**. Запустится программа User's Configuration. В ее окне можно выбрать соответствующую вкладку для установки часового пояса, времени и даты. Вкладку **Language** можете не трогать — русского языка там нет. Зато есть замечательная вкладка **Keyboard**. Откройте эту вкладку и выберите строку **Russian**. Не забудьте нажать кнопки **Apply** (применить изменения) и **Done** (закрыть окно).

А теперь придется немного "поработать ручками". Дело в том, что программа User's Configuration считает, что пользователь использует либо английскую, либо русскую раскладку. А нам все же нужна английская раскладка, поэтому надо выполнить нехитрые действия: открыть файл `/etc/system/trap/.KEYBOARD.myname` (обратите внимание: `myname` — это имя нашей машины) и отредактировать его так, чтобы он содержал две строки:

```
en_US_101.kbd
ru_RU_102.kbd
```

Раскладка, указанная первой, будет использоваться по умолчанию. Обратите внимание, что имя файла начинается с точки (`.`). Это означает, что файл скрытый, и средства просмотра файловой системы по умолчанию его не отображают. Чтобы видеть скрытые файлы в штатном файловом менеджере графической оболочки Photon, выберите элемент **Preferences** меню **Edit** и в открывшемся окне снимите самый верхний флажок **Hide 'dot' Files** (скрывать файлы, имя которых начинается с точки) (рис. 1.6).

Рис. 1.6. Окно **Preferences** файлового менеджера **pfm**

Теперь при запуске приложений графической оболочки Photon можно будет вводить как русский, так и английский текст (для переключения между раскладками клавиатуры используются комбинация клавиш <Alt> + <Shift> (слева)).

Однако в командной строке (как "черного экрана", так и псевдотерминала) мы не сможем ни вводить, ни читать русские буквы. Для решения этой проблемы предназначен пакет SWD Cyrillic Pack, поставляемый с пошаговой инструкцией на русском языке, добавить к которой мне нечего.

1.2.4. Некоторые дополнительные настройки

Photon microGUI, как и всякая более-менее функциональная графическая оболочка, имеет различные средства конфигурирования, которые нетрудно отыскать, нажав кнопку **Launch** и выбрав меню **Configure**. Не буду даже перечислять эти утилиты — сами разберетесь, не маленькие.

Если требуется, чтобы какие-то команды выполнялись при каждом старте QNX Neutrino, то создается файл `/etc/rc.d/rc.local`, доступный по исполнению для администратора (пользователя `root`):

```
touch /etc/rc.d/rc.local
chmod 744 /etc/rc.d/rc.local
```

Затем этот файл открывается в любом текстовом редакторе и в него вписываются в виде сценария те команды, которые должны выполняться при каждом запуске Neutrino. На моей ЭВМ так запускается, например, программа **`inetd`**.

> ***Примечание***
> Подробнее командные сценарии описаны в *разд. 1.3.1*.

Если требуется, чтобы какие-то приложения автоматически запускались при старте Photon, то создается файл `$HOME/.ph/phapps` с атрибутом "исполняемый" для владельца:

```
touch ~/.ph/phapps
chmod 744 ~/.ph/phapps
```

Затем этот файл открывается в любом текстовом редакторе и в него вписываются в виде сценария те команды, которые должны выполняться при каждом запуске оболочки Photon microGUI от вашего имени. На моей ЭВМ так запускается, например, программа чтения электронной почты.

> ***Примечание***
> Сценарий `phapps`, в отличие от сценария `rc.local`, пишется для каждого пользователя отдельно.

Напоследок вспомним, что любой приличный файловый менеджер умеет что-то делать при двойном щелчке на имени файла. Если это "что-то" (назовем его "обработчиком по умолчанию") нам не подходит, то на имени файла можно щелкнуть правой кнопкой мыши и в открывшемся меню выбрать подходящий элемент (тоже "обработчик", но не по умолчанию). Файловый менеджер **pfm** позволяет изменять обработчики, соответствующие элементам меню **Open**, **View** и **Edit**. Обработчик **Open** является обработчиком по умолчанию, т. е. вызывается двойным щелчком на имени файла. Итак, для настройки обработчиков надо выбрать элемент **Associations** меню **Edit** — откроется соответствующее окно (рис. 1.7).

Рис. 1.7. Привязка программ-обработчиков к именам файлов

Окно ассоциаций представляет собой таблицу, в которой для файлов с различными расширениями имени задаются программы-обработчики. Для добавления новых ассоциаций нажмем кнопку **Add**. В открывшемся окне (рис. 1.8) заполним соответствующие поля. Например:

Рис. 1.8. Ассоциируем PDF-файлы с программой **xpdf**

Если нужно установить одинаковые обработчики для файлов с разными расширениями имени, то в поле **Pattern** (шаблон) можно перечислить шаблоны имен через логическое "И" (знак конъюнкции — "|"). И пожалуйста, не забывайте нажимать кнопку **Done** (простите за назойливость, но уж очень часто пользователи забывают это делать!).

> ***Примечание***
> Программа **xpdf** входит в пакет XPDF, доступный в репозитарии 3rd-Party Software. Для ее работы необходимо из того же репозитария скачать пакет XPhoton и запустить сервер **XPhoton** на своей ЭВМ. Кстати, XPhoton удобно запускать из сценария **phapps**.

1.3. Основы работы в QNX Neutrino

Операционная система — это, конечно, здорово. Однако сама по себе она, например, мне не нужна. Ценность операционной системы заключается в программах, которые в ней работают, причем работают так, как от них ожидается. В предисловии мы обсуждали предназначение Neutrino и следствие этого предназначения — малое

число готовых приложений и богатый инструментарий их создания. Тем не менее, в соответствии с требованиями стандарта POSIX.2 в штатной поставке ОСРВ Neutrino есть сотня-другая утилит командной строки. Не пользоваться ими было бы по меньшей мере неразумно.

1.3.1. Командная строка

С командным интерпретатором можно работать как в "черном экране", так и в графической среде. Для использования командного интерпретатора в графической оболочке Photon нужно запустить псевдотерминал **`pterm`**. Командный интерпретатор выведет на экран приглашение (для "простого" пользователя — **`$`**, для суперпользователя — `#`). В ответ на это приглашение можно напечатать *команду* и нажать клавишу <Enter>. Интерпретатор выполнит команду (либо сообщит об ошибке, если ему это не удалось) и вновь выведет приглашение для ввода следующей команды. И так до тех пор, пока не получит команду **`exit`** (выход).

Что такое "командная строка"?

Командный интерпретатор — тоже программа. Есть несколько программ такого назначения:

- Bourne Shell (**`sh`**) — интерпретатор, стандартизованный в рамках POSIX 1003.2;
- Korn Shell (**`ksh`**) — расширение **`sh`**;
- Bourne Again Shell (**`bash`**) — модернизированный интерпретатор **`sh`**;
- C-Shell (**`csh`**) — интерпретатор с "Си"-подобным синтаксисом сценариев.

(Существуют и другие интерпретаторы.)

В ОС Linux, например, интерпретатором по умолчанию является **`bash`**, в QNX — **`ksh`**. Поскольку **`ksh`** полностью реализует функциональные возможности **`sh`**, ради экономии места, с одной стороны, и по традиции — с другой, в QNX файл `/bin/sh` — символическая ссылка на `/bin/ksh`. В одной системе можно одновременно использовать любые интерпретаторы.

Бывает, что использовать интерпретатор **`ksh`** в системах с ограниченными ресурсами — "дорогое" удовольствие. Поэтому в составе QNX есть и "урезанные" интерпретаторы:

- **esh** — встраиваемый командный интерпретатор (Embedded Shell). Предоставляет подмножество функциональных возможностей **ksh**, при запуске выполняет простые команды из файла `/etc/esh`. Этот интерпретатор не позволяет писать программы с условными операторами и т. п.;
- **uesh** — малый встраиваемый интерпретатор (Micro-embedded Shell), получен в результате урезания **esh**;
- **fesh** — "жирный" встраиваемый интерпретатор (Fat embedded shell). В противоположность **uesh**, **fesh** является расширением **esh**. Этот вариант shell появился относительно недавно из-за того, что аппаратура современных встраиваемых систем достаточно производительна, чтобы не использовать столь "хилые" интерпретаторы, как **esh** и **uesh**.

Некоторые часто используемые команды

Посмотрим, в каком месте файловой системы мы находимся, т. е. введем команду **pwd** в ответ на приглашение `#` и нажмем клавишу <Enter>:

```
# pwd
/root
```

В ответ на экране появится путевое имя каталога, в котором мы находимся (`/root`).

> ***Примечание***
> Файловая система операционных систем семейства QNX имеет общепринятую для UNIX-подобных систем древовидную структуру. Корнем этого дерева является каталог, обозначаемый символом `/` (слэш) и называемый "root".

Для удобства (моего — не вашего) я больше не буду помещать символ приглашения интерпретатора и упоминать о клавише <Enter>. Теперь перейдем в другой каталог, например, в `/tmp`:

```
cd /tmp
```

Обратите внимание, что я указал *абсолютное путевое имя* каталога, в который перехожу. Путевое имя — это полное имя каталога или файла, однозначно указывающее место каталога (файла) в дереве файловой системе начиная от каталога "root". Имя можно указывать и *относительное*, т. е. относительно *текущего* каталога. Если имя начинается не с `/`, то Neutrino считает, что указано относительное имя.

Это значит, что если бы мы в предыдущем примере ввели такую команду:

```
cd tmp
```

то операционная система решила бы, что мы хотим перейти в каталог `/root/tmp`. Чтобы из любого места файловой системы вернуться в домашний каталог, надо ввести команду **`cd`** без аргументов. Чтобы перейти в вышестоящий каталог (т. е. приблизиться к каталогу `/`), надо ввести такую команду:

```
cd ..
```

Можно создавать свои каталоги, например, создадим каталог `/tmp/myfilder`. Если мы находимся в каталоге `/tmp`, то достаточно указать относительный путь:

```
mkdir myfolder
```

> ***Примечание***
>
> Кстати, о путевых именах. Каким образом интерпретатор определяет — где в дереве файловой системы находится запускаемая нами программа (утилита)? Интерпретатор перебирает по очереди все каталоги, перечисленные через двоеточия в переменной окружения `PATH`. То есть запускаемая программа должна находиться в одном из каталогов, указанных в переменной `PATH`, в противном случае программа просто не будет найдена. Значение этой переменной можно посмотреть с помощью команды:
>
> ```
> echo $PATH
> ```
>
> Значения всех переменных окружения можно посмотреть с помощью команды **`set`** без аргументов.

Теперь создадим в новом каталоге файл `myfile`. Из каталога `/tmp` это можно сделать так:

```
touch myfolder/myfile
```

или выполнить две команды:

```
cd myfolder
touch myfile
```

или указать абсолютное путевое имя создаваемого файла:

```
touch /tmp/myfolder/myfile
```

Посмотрим на содержимое каталога `/usr/bin`:

```
ls /usr/bin
```

Создадим копию файла `/etc/services`, присвоив ей имя `/tmp/myfilder/myfile_2`:

```
cp /etc/services /tmp/myfolder/myfile_2
```

Переименуем `/tmp/myfolder/myfile_2` в `/tmp/myfolder/myfile_3`:

```
mv /tmp/myfolder/myfile_2  /tmp/myfolder/myfile_3
```

Посмотрим содержимое файла `/tmp/myfolder/myfile_3`:

```
cat /tmp/myfolder/myfile_3
```

Содержимое файла не уместилось на экране? Не беда, воспользуемся "конвейером" (подробнее конвейеры и перенаправление ввода/вывода описаны в *разд. 1.3.1*):

```
cat /tmp/myfolder/myfile_3 | more
```

Теперь можно ограничивать вывод на экран фрагментами, по размеру соответствующими доступному пространству экрана, нажимая клавишу <пробел> (прекратить вывод — клавиша <Q>).

Удалим файл `/tmp/myfolder/myfile`:

```
rm /tmp/myfolder/myfile
```

Удалим каталог `/tmp/myfolder`:

```
rmdir /tmp/myfolder
```

Что, не получается? Конечно, команда **`rmdir`** умеет удалять только пустые каталоги, а в `/tmp/myfolder`, если вы выполнили все мои предыдущие инструкции, находится файл `myfile_3`. Уничтожить непустой каталог поможет следующая команда:

```
rm -R /tmp/myfolder
```

Впрочем, закончим разминку. Список доступный утилит с подробным описанием каждой из них можно посмотреть, запустив Helpviewer и найдя там документ Utilities Reference. Обращу лишь ваше внимание на то, что помимо аргументов (например, имени файла) утилиты часто имеют по несколько опций, модифицирующих поведение утилиты. Например, утилита **`ls`** с опцией **`-l`** (от англ. long — длинный) выводит содержимое указанного в качестве аргумента каталога (или текущего каталога, если аргумент не указан) в "длинном" формате, показывающем различные атрибуты файла:

```
ls -l /etc/rc.d
```

Содержимое текстовых файлов (часто называемых ASCII-файлами) можно изменять, например, с помощью текстовых редакторов. В Photon есть редактор **`ped`** (**P**hoton **Ed**itor). Поскольку графическая оболочка далеко не всегда доступна (при удаленной работе, нехватке ресурсов и

т. п.), администратору необходимо уметь пользоваться редактором **vi** *(см. разд. 1.3.2)*. Не пожалейте немного времени на изучение **vi** — иногда это единственный доступный инструмент редактирования файлов конфигурации.

Дополнительные сведения о командной строке

Строго говоря, команды могут быть как отдельными программами (утилитами), так и "встроенными" командами shell. К встроенным командам относятся **cd**, **pwd**, **set**, **echo**, **alias** и т. д.

Можно задать несколько команд в одной командной строке, разделив их точкой с запятой (;). Если команда длинная, и ее необходимо перенести на следующую строку, то перед переводом строки ставят обратный слэш (\). Чтобы запустить команду в фоновом (неинтерактивном) режиме, в ее конце ставиться амперсанд (&). Например, запустим утилиту **ped** — штатный текстовый редактор Photon из окна псевдотерминала:

```
ped &
```

Интерпретатор сообщит номер фонового процесса и выдаст приглашение ввести новую команду не дожидаясь завершения фоновой программы.

Если вы забыли добавить амперсанд, введя только

```
ped
```

то можно остановить интерактивный процесс, нажав комбинацию клавиш <Ctrl> + <Z>, а затем перевести его в фоновый режим с помощью команды **bg**. Перевод последней программы, запущенной в фоновом режиме, в интерактивный режим выполняется с помощью команды **fg**.

Если требуется, чтобы фоновый процесс завершился прежде, чем будут выполнены какие-либо действия, используется команда **wait**. Если в качестве аргумента для **wait** указать PID[1] конкретного процесса, то она будет ждать именно его завершения, а если этот параметр не задавать, то **wait** будет ожидать завершения всех фоновых процессов, дочерних для данного shell.

Интерактивный процесс можно принудительно завершить, используя комбинацию клавиш <Ctrl> + <C>, а фоновый процесс — с помощью POSIX-утилиты **kill** или QNX-утилиты **slay**. В качестве аргумента

---

[1] Process ID — идентификатор процесса.

для утилиты **kill** требуется указать PID уничтожаемого процесса, а для утилиты **slay** — имя уничтожаемого процесса. Идентификатор процесса, в свою очередь, можно получить либо с помощью POSIX-утилиты **ps**, либо с помощью QNX-утилит **sin** и **pidin**.

Двойной амперсанд && предписывает интерпретатору shell выполнять следующую команду при условии нормального завершения предыдущей, иначе — игнорировать. Например:

```
mkdir /tmp/myfolder2 && touch /tmp/myfolder2/myfile2
```

В этом случае сначала будет предпринята попытка создать каталог `/tmp/myfolder2`, а затем (если удалось создать каталог) в нем будет создан файл `myfile2.txt`.

Символ `||`, напротив, предписывает интерпретатору shell выполнять следующую команду при ненормальном завершении предыдущей, иначе — игнорировать. Например:

```
mkdir /tmp/myfolder2 || rm /tmp/*
```

В этом случае сначала будет предпринята попытка создать каталог `/tmp/myfolder2`, и, если каталог создать не удалось, — будет удалено содержимое каталога временных файлов `/tmp`.

Потоки ввода/вывода и конвейеры

При запуске с "обычной" программой связывается три так называемых *стандартных потока ввода/вывода*. Каждый из этих потоков представлен в виде файлового дескриптора:

- 0 — стандартный ввод (stdin);
- 1 — стандартный вывод (stdout);
- 2 — стандартный поток диагностических сообщений (stderr), обычно его называют "стандартным потоком ошибок".

Эти потоки нужны для того, чтобы могли работать стандартные функции языка C и C++, которые считывают информацию с консоли и выводят что-либо на экран. Поток 0 по умолчанию связан с клавиатурой, а потоки 1 и 2 — с экраном терминала. Убедиться в этом можно, выполнив команду просмотра дескрипторов открытых файлов:

```
sin fd
```

Командный интерпретатор позволяет перенаправлять стандартные потоки ввода, т. е. связывать их с какими-либо файлами. Перенаправление стандартного вывода выполняется с помощью символа >. Например:

ls /usr > /tmp/file1

Команда **ls** сформирует список содержимого каталога /usr и вместо того, чтобы вывести результат на экран, поместит его в файл /tmp/file1. Если файла с таким именем не было, то он будет создан. Если же файл /tmp/file1 уже существовал, то его прежнее содержимое будет уничтожено.

Выполним другую команду:

pwd > /tmp/file1

Команда **pwd** сформирует полное имя текущего каталога и поместит его в файл /tmp/file1, уничтожив прежнее содержимое файла. А если нам нужно добавить новую информацию к прежней? Тогда используется директива >>, т. е. предыдущая команда будет выглядеть так:

pwd >> /tmp/file1

В этом случае вывод утилиты **pwd** будет добавлен в конец содержимого файла /tmp/file1.

Для перенаправления стандартного ввода используется символ <. Например:

wc –c < /tmp/file1

Команда **wc** подсчитает и выдаст на экран число символов в файле /tmp/file1.

Строго говоря, при перенаправлении следует указывать номер перенаправляемого потока, однако shell по умолчанию знает, что ">" — это "1>", а "<" — это "0<". Если же требуется оперировать другими потоками (например, с stderr), то значение дескриптора нужно задавать явно. Проиллюстрируем:

ls /tmp11

Если каталог /tmp11 не существует, то команда выдаст сообщение об ошибке:

```
ls: No such file or directory (/tmp11)
```

Допустим, мы хотим все диагностические сообщения команды "сбрасывать" в файл /tmp/errlog. Тогда команда должна выглядеть так:

ls /tmp11 2> /tmp/errlog

Если же сообщения об ошибках нас вообще не интересуют, и мы не хотим "засорять" ими экран, то сделаем так:

ls /tmp11 2>/dev/null

Потоки можно объединять, используя конструкцию >&. Для иллюстрации выполним команду, пишущую в стандартные потоки вывода и ошибок. Для этого попытаемся вывести на экран содержимое существующего файла (/tmp/file1) и несуществующего файла (/tmp/file2), а стандартный вывод направим в файл /tmp/log:

```
cat /tmp/file1 /tmp/file2 > /tmp/log
```

Теперь модифицируем команду так, чтобы поток ошибок направлялся туда же, куда и поток вывода:

```
cat /tmp/file1 /tmp/file2 > /tmp/log 2>&1
```

В приведенной команде поток 2 добавляется к потоку 1.

Допускается произвольно комбинировать операции перенаправления потоков в одной команде:

```
wc -c </tmp/file1 >/tmp/file2
```

В данном случае утилита **wc** осуществит ввод из файла /tmp/file1, а результат запишет в файл /tmp/file2.

Вывод одной программы можно объединять с вводом другой. Конструкция, обеспечивающая такое объединение, обозначается символом | и называется *конвейером*:

```
ls –l /usr/bin | wc -l
```

В данном примере утилита **ls** сформирует вывод, состоящий из информации о содержимом каталога /usr/bin, результат будет передан на "вход" утилиты **wc**, которая отобразит на экране количество строк в выводе **ls**.

Если программа "умеет" работать и со стандартным вводом, и со стандартным выводом, то ее называют *фильтром* (типичный фильтр — утилита **wc**). В конвейере можно использовать несколько команд, а значит, все команды конвейера, кроме первой и последней, должны быть фильтрами. Например:

```
ls /usr/bin | sort | wc -l > /tmp/file2
```

Уместно вспомнить утилиту **tee**, позволяющую дублировать поток вывода. Один из потоков вывода, например, можно записать в файл, а другой — вывести на экран:

```
ls – l /usr | tee /tmp/lslog
```

Говоря точнее, утилита **tee** направляет свой стандартный ввод на свой стандартный вывод, дублируя его в указанные файлы (в примере задан только один файл — /tmp/lslog).

В заключение разговора о потоках хочу напомнить, что устройства в QNX представлены в виде специальных файлов, а значит механизм перенаправления потоков можно использовать для "сырых" операций чтения/записи с устройствами. Под "сырыми" я в данном случае подразумеваю такие операции, при выполнении которых пользователь сам отвечает за то, что формат передаваемых им данных будет понятен драйверу, а формат вывода драйвера будет понятен ему самому.

Например, выведем сообщение "Hello folks!" в окно псевдотерминала **pterm**. Для этого в одном окне **pterm** введем команду **tty**. В ответ мы получим имя файла псевдотерминала, ассоциированного с данной копией **pterm** Пусть это будет `/dev/ttyp1`. Тогда в любом другом окне **pterm** введем команду:

```
echo Hello folks! > /dev/ttyp1
```

Содержимое файла файла `/tmp/file1` можно обнулить:

```
cat >/tmp/file1 < /dev/null
```

Можно передавать результат выполнения одной команды в качестве аргумента другой команде, для этого команду-"аргумент" следует заключить в одиночные косые кавычки (`).

> ***Примечание***
> Клавиша этого символа обычно находится под клавишей <Esc> в левой верхней части клавиатуры — не путайте с клавишей апострофа ('), расположенной рядом с клавишей <Enter>.

Проиллюстрируем:

```
less `which ph`
```

Команда **which** ищет указанный файл по очереди во всех каталогах, перечисленных в переменной окружения `PATH` (т. е. там же, где ищет команду интерпретатор **shell**), как только найдет — выводит на экран путевое имя найденного файла и прекращает поиск. Команда **less** позволяет "прокручивать" длинные текстовые файлы, дозируя их фрагментами, равными доступному пространству экрана (прокручивание выполняется нажатием клавиши <пробел>, выход — <Q>). То есть указанная команда позволяет просмотреть текст сценария **ph**, не задумываясь о том, где он физически находится.

Таким образом, перенаправление, слияние, объединение потоков ввода/вывода предоставляет администратору мощный инструмент управления системой. Этот механизм подобен технологиям компонентного программирования, получившим широкое распространение в последние годы. При этом компонентами являются

любые программы, как входящие в дистрибутив, так и написанные самим пользователем. Shell же позволяет сочетать их произвольным образом.

Рассмотрим такой пример: пусть требуется получить список файлов каталога `/bin`, являющихся символическими ссылками, причем для каждого файла в этом списке должна быть указана контрольная сумма.

> ***Примечание***
> Символические ссылки описаны в *главе 4*.

Можно написать программу, которая будет делать эту работу, а можно использовать штатные утилиты QNX. Поиск файлов по заданным критериям выполняет утилита **`find`**:

```
find /bin –type l
```

Контрольное суммирование выполняет утилита **`cksum`**. Она принимает аргумент — список объектов, поэтому передадим ей в качестве такого аргумента результат выполнения утилиты **`find`**:

```
cksum `find /bin –type l`
```

И, наконец, форматирование потока вывода выполняет утилита **`awk`**. От **`awk`** нам нужна возможность оставить в выводе утилиты **`cksum`** только первый и третий столбцы, разделенные символом табуляции, причем сначала выведем третий столбец (имя файла), а затем — первый (контрольная сумма). Результат сохраним в файле `result.txt`. Итак:

```
cksum `find /bin –type l` | awk '{print $3 "\t" $1}' > \
/tmp/result.txt
```

Как видите, с помощью стандартных POSIX-утилит и командного интерпретатора можно легко решать достаточно разнообразные и сложные задачи.

1.3.2 Использование редактора *vi*

Редактор **`vi`** запускается очень просто: **`vi`** ***`имя_файла`***. Если файл с именем ***`имя_файла`*** не существует, то он будет создан. Редактор может работать в одном из двух режимов:

- режим ввода;
- командный режим.

После запуска редактор находится в командном режиме. Переход из командного режима в режим ввода осуществляется нажатием клавиши <I> (input) или <A> (add). После нажатия клавиши <I> текст будет

вводиться, начиная с текущей позиции курсора, а после нажатия клавиши <A> — начиная со следующей позиции после курсора. Возврат в командный режим осуществляется нажатием клавиши <Esc>.

Надо сказать, что возможностей у **vi** достаточно много. Если вы хотите хорошо изучить этот редактор, то можете прочитать соответствующую литературу. Но, честно говоря, освоение работы в редакторе **vi** сводится к запоминанию нескольких простых команд, представленных в табл. 1.1.

Таблица 1.1. *Основные команды редактора* **vi**

Команда	Назначение
i	Перейти в режим ввода (insert)
a	Перейти в режим ввода (add)
yy	Скопировать текущую строку в буфер
{*n*}yy	Скопировать ***n*** строк, начиная с текущей, в буфер
dd	Удалить текущую строку в буфер
{*n*}dd	Удалить ***n*** строк, начиная с текущей, в буфер
p	Вставить содержимое буфера
x	Удалить текущий символ
G	Перейти в конец файла
{*n*}G	Перейти на строку с номером ***n***
/шаблон	Поиск шаблона ***шаблон***
n	Повторить поиск
:w	Сохранить изменения
:w *файл*	Сохранить изменения в файле ***файл*** ("Сохранить как…")
:q	Выход
:q!	Выход без сохранения изменений

Так что ничего сложного или страшного в работе с редактором **vi** нет. По правде сказать, в QNX используется не **vi**, а его модернизированная версия **elvis**; **vi** — просто символическая ссылка на него (что такое символическая ссылка, мы обсудим в *главе 4*). Многие программисты и

администраторы предпочитают другой клон **vi** — **vim** (от **V**i **IM**provement). Одно из достоинств **vim** — поддержка выделения синтаксиса для разных языков программирования (для этого в командном режиме введите **:syntax on**).

1.3.3. Написание собственных программ на языке C

Для того чтобы получить собственную программу, нужно написать ее исходный текст на языке C или C++. Файл или файлы с исходным текстом сначала обрабатываются *препроцессором*, затем то, что получилось, обрабатывается соответственно *компилятором* C или C++. В результате получаем объектный файл. В этот объектный файл *компоновщик* ("линкер") включает объектные файлы из библиотек и в результате получается исполняемый файл — программа.

Итак, для получения приложения исходный текст обрабатывается таким конвейером: препроцессор—компилятор—компоновщик. Эту цепочку удобнее всего вызывать командой **qcc**. Удобство создается тем, что у этой команды есть конфигурационные файлы, позволяющие ей задавать по умолчанию оптимальную комбинацию опций для всех перечисленных инструментов конвейера (см. каталог `/etc/qcc`). Давайте напишем хрестоматийную программу, выдающую на экран избитую фразу "Hello, world!". Для этого создадим текстовый файл с именем `hello.c` и наполним его текстом на языке C:

```
main()
{
      printf("Hello, world!\n");
}
```

Обратите внимание, что я не потрудился написать директиву препроцессора `#include <stdio.h>` — **qcc** сама "знает", что без заголовочного файла `stdio.h` не обойтись.

Для того чтобы получить программу **hello** для Intel-совместимых целевых систем, следует выполнить команду

```
qcc -Vgcc_ntox86 hello.c -o hello
```

Получить полный список целевых платформ, для которых можно сгенерировать приложение, можно командой

```
qcc -V
```

Например, `gcc_ntoppcbe` — для целевых систем PowerPC BigEndian, а `gcc_ntomipsle` — для MIPS LittleEndian.

На самом деле, по умолчанию и так создается исполняемый файл для аппаратуры x86. Поэтому в рассматриваемом частном случае достаточно ввести такую команду:

```
qcc hello.c -o hello
```

Запустим программу. Поскольку она вряд ли содержится в переменной `PATH` *(см. разд. 1.3.1)*, укажем ее относительное путевое имя:

```
./hello
```

Каталог "точка" (`.`) — это текущий каталог, т. е. имя `./hello` обозначает файл `hello` в текущем каталоге.

> ***Примечание***
> Можно добавить в переменную `PATH` текущий каталог, чтобы интерпретатор всегда сначала "заглядывал" в него и только потом искал в остальных каталогах:
>
> ```
> PATH=.:$PATH
> ```
>
> А если эту команду поместить в файл `$HOME/.profile`, то она будет автоматически выполняться перед каждым запуском интерпретатора.

С QNX Neutrino 6.3 поставляются два GCC-компилятора — версия 2.95.2 (используется по умолчанию) и 3.3.1. Можно откомпилировать ту же программу компилятором посвежее:

```
qcc -V3.3.1,gcc_ntox86 hello.c -o hello_new
```

1.3.4. Управление программными проектами с помощью утилиты *make*

Рассмотрим быть может, глуповатый, но вполне удобный для учебных целей проект. Функция *main()*[1] определена в файле `main.c`:

```
#include "defs.h"
int main()
```

---

[1] Для тех кто не в курсе дела. Функция *main()* — единственная обязательная функция программы, написанной на языке C или C++. Она является точкой входа в программу, т. е. местом, с которого программа начинает выполняться. Остальные функции вызываются уже из функции *main()*. Впрочем, других функций может и не быть.

```
{
      printf("I'm main\n");
      aaa();
      return 0;
}
```

Все, что делает функция *main()*, — выводит на экран фразу "I'm main", затем вызывает функцию *aaa()* и завершает работу с кодом возврата 0. В файле `main.c` есть директива для вставки заголовочного файла `defs.h`[1], который содержит объявления функций *aaa()*, *bbb()* и *ccc()*:

```
#ifndef            _MY_DEFS_
#define            _MY_DEFS_
#ifndef            _EXT
#define            _EXT        extern
#endif            // _EXT
_EXT void aaa();
_EXT void bbb();
_EXT void ccc();
#endif            // _MY_DEFS_
```

> ***Примечание***
>
> Директивы препроцессора (все они начинаются с символа #), используемые в `defs.h`, нужны для избежания многократного включения файла `defs.h` и конфликтов при объявлении функций в разных комбинациях.

Каждая из этих функций определена в отдельном файле, содержащем директиву включения того же самого заголовочного файла `defs.h`. Функция *aaa()* определена в файле `aaa.c`. Она выводит на экран текст `"I'm aaa"` и вызывает функцию *bbb()*:

```
#include"defs.h"
void aaa()
{
```

---

[1] Помните, мы говорили, что для получения исполняемой программы файл с текстом программы на языке C или C++ обрабатывается конвейером, состоящим из инструментов препроцессор–компилятор–компоновщик? Так вот, *препроцессор* (утилита **cpp**) выполняет *директивы препроцессора*, одной из которых является `include`, указывающая препроцессору вставить на ее место содержимое заданного заголовочного файла.

```
   printf("I'm aaa\n");
   bbb();
}
```

Функция *bbb()* определена в файле `bbb.c`. Она выводит на экран текст `"I'm bbb"` и вызывает функции *ccc()*:

```
#include"defs.h"
void bbb()
{
   printf("I'm bbb\n");
   ccc();
}
```

Функция *ccc()* определена в файле `ccc.c` и выполняет всего одно действие — выводит на экран текст `"I'm ccc"`:

```
#include"defs.h"
void ccc()
{
   printf("I'm ccc\n");
}
```

Для того чтобы из этих пяти файлов получить программу **`program`**, следует выполнить такие команды:

```
qcc –c main.c
qcc –c aaa.c
qcc –c bbb.c
qcc –c ccc.c
qcc –o program main.o aaa.o bbb.o ccc.o
```

Выполнение такой цепочки команд называют *сборкой проекта*. Если изменить файл `ccc.c`, то для получения свежей версии программы **`program`** потребуется вновь выполнить две последние команды, а если изменить файл `main.c`, то потребуется повторить первую и последнюю команды. Хорошо еще, что мы не используем многочисленные дополнительные опции, которые могут быть разными для компиляции разных файлов. Конечно, можно было задать одну команду:

```
qcc –o program main.o aaa.o bbb.o ccc.o
```

Однако в этом случае сборка проекта будет выполняться как одна, если можно так выразиться, транзакция и при сборке свежей версии проекта будут перекомпилироваться все файлы.

Короче говоря, проще написать некий конфигурационный файл, описывающий зависимости между файлами проекта и то, какую последовательность команд надо выполнить, чтобы пересобрать проект при внесении изменений, не затрагивая при этом те файлы, на которые изменения не повлияли. Таким файлом управления сборкой проекта является `Makefile`, а анализируется этот файл утилитой **`make`**. Точнее, при запуске утилита **`make`** ищет сначала файл с именем `makefile`, затем, если не нашла, — файл с именем `Makefile`. В принципе, можно задать любое имя, указав его утилите **`make`** с помощью опции **`-f`**.

> ***Примечание***
> Кстати, интегрированные среды разработки также используют утилиту **`make`**, скрывая это от программиста за графическим интерфейсом.

Простейший файл `Makefile` содержит только два типа синтаксических конструкций — цели и макросы.

Цель — это имя файла, который следует сгенерировать. Описание цели имеет такой вид:

```
ЦЕЛЬ: ЗАВИСИМОСТЬ-1 ЗАВИСИМОСТЬ-2 ... ЗАВИСИМОСТЬ-N
      КОМАНДА-1
      КОМАНДА-1
      ...
      КОМАНДА-N
```

`ЦЕЛЬ-`*`i`* — непустой список файлов, которые предполагается создать. `ЗАВИСИМОСТЬ-`*`i`* — список файлов, из которых строится цель. В качестве зависимости может использоваться другая цель. Имя цели и список зависимостей записываются в одну строку и разделяются двоеточием. Они составляют *заголовок цели*. Ниже следует список *команд*, каждая команда — в отдельной строке.

> ***Примечание***
> Каждая команда ОБЯЗАТЕЛЬНО начинается с символа табуляции.

Утилиту **`make`** можно запускать или с указанием имени конкретной цели, или без имени цели. Если цель не указана, то **`make`** строит первую цель, заданную в файле `Makefile`, имя которой не начинается с точки. Построение цели заключается в выполнении команд, заданных для цели. Перед тем как строить цель, **`make`** сравнивает время последней модификации цели и зависимостей. Если какая-либо из зависимостей была изменена после последней сборки цели, то цель пересобирается.

Если какая-то зависимость сама является целью (т. е. "подцелью"), то сначала собирается "подцель".

Итак, с учетом сказанного, напишем файл управления сборкой нашего проекта с именем `mf1`:

```
program: main.o aaa.o bbb.o ccc.o
	qcc -o program main.o aaa.o bbb.o ccc.o

main.o : main.c defs.h
	qcc –c main.c

aaa.o : aaa.c defs.h
	qcc –c aaa.c

bbb.o : bbb.c defs.h
	qcc –c bbb.c

ccc.o : ccc.c defs.h
	qcc –c ccc.c
```

Выполним сборку:

```
make –f mf1
```

Примечание

Если вы выполняли все описанные команды в порядке изложения материала, то программа (цель) **`program`** уже существует в самой свежей, так сказать, версии. Из-за этого утилита **`make`** не будет выполнять сборку, а выведет сообщение:

```
make: 'program' is up to date
```

Для верстки: следующий абзац - перевод предыдущей строки, сделать такой же отступ, а шрифт сделать такой же, как в предыд. переводах

(make: исходный код программы program не изменялся со времени последней компиляции)

Поэтому сначала удалите файл `program` или сделайте, например, файл `defs.h` более "старым", чем `program`:

```
touch defs.h
```

Поскольку утилита **make** вызвана без указания цели, будет вызвана первая цель — program. Для цели program заданы четыре зависимости, каждая из которых является подцелью. Утилита **make** сначала соберет подцели, а затем и цель. Можно было задать произвольную цель:

```
make -f mf1 ccc.o
```

Тогда бы собиралась только цель ccc.o, т. к. среди ее зависимостей нет подцелей.

На самом деле утилита **make** достаточно умна, поэтому если в качестве цели задан объектный файл (т. е. файл с расширением .o) и существует одноименный C-файл, то **make** сама вызовет **qcc** с опцией **-c**. Поэтому мы можем записать наш пример так (назовем новый файл mf2):

```
program: main.o aaa.o bbb.o ccc.o
      qcc -o program main.o aaa.o bbb.o ccc.o

main.o aaa.o bbb.o ccc.o: defs.h
```

Заметьте, что мы один и тот же список объектных файлов в трех строчках записали три раза. Чтобы этого не делать, **make** поддерживает макросы, имеющие синтаксис:

ПЕРЕМЕННАЯ=ЗНАЧЕНИЕ

ЗНАЧЕНИЕ может являться произвольной последовательностью символов, включая пробелы и обращения к значениям ранее определенных переменных. После объявления переменной мы можем обратиться к ней так: $(ПЕРЕМЕННАЯ). Вновь откорректируем наш пример (получим файл mf3):

```
OBJS=main.o aaa.o bbb.o ccc.o
program: $(OBJS)
      qcc -o program $(OBJS)
$(OBJS): defs.h
```

Некоторые переменные уже предопределены. К ним относятся, например:

- AR=ar — программа-архиватор;
- AS=as — ассемблер;
- CC=cc — компилятор C (**cc** — это ссылка на **qcc**);
- CXX=g++ — компилятор C++;
- CPP=cpp — препроцессор;

- `LEX=lex` — лексический анализатор С-программ;
- `RM=rm -f` — команда удаления файлов.

Есть также специальные встроенные макросы, значение которых изменяется в процессе выполнения утилиты **`make`**:

- **`$@`** — имя текущей цели;
- **`$?`** — список зависимостей, более свежих, чем цель;
- **`$^`** — полный список зависимостей для данной цели;
- **`$*`** — имя цели без расширения.

> ***Примечание***
> Полный перечень всех предопределенных макропеременных и зависимостей можно получить с помощью команды:
>
> **`make –p </dev/null 2>/dev/null`**

Вставим служебные переменные в наш пример (получим файл `mf4`):

```
OBJS=main.o aaa.o bbb.o ccc.o
program: $(OBJS)
      $(CC) -o $@ $(OBJS)
$(OBJS): defs.h
```

Для удобства, да и для корректности, следовало бы добавить служебную цель, позволяющую удалять результаты компиляции. Такая цель имеет общепринятое название — `clean` (получим файл `mf5`):

```
OBJS=main.o aaa.o bbb.o ccc.o
program: $(OBJS)
      $(CC) -o $@ $(OBJS)
$(OBJS): defs.h
clean:
      $(RM) program $(OBJS)
```

Иногда добавляют и другие цели, например, `install`.

Поскольку имена целей задаются фактически произвольно, могут возникать некорректные совпадения имен целей и имен файлов проекта. Для обработки возникающих при этом конфликтов предназначена переменная `.PHONY`, в качестве значений которой указывают имена служебных целей. Проиллюстрируем это в файле `mf6`:

```
OBJS=main.o aaa.o bbb.o ccc.o
.PHONY=clean
program: $(OBJS)
```

```
	$(CC) -o $@ $(OBJS)
$(OBJS): defs.h
clean:
	$(RM) program $(OBJS)
```

Ну и последний, пожалуй, штрих. В файле `Makefile` могут присутствовать комментарии. Началом комментария считается символ #, а окончанием — конец строки. Остается только направить читателя к документации утилиты **`make`** для получения дополнительной информации.

1.3.5. Встроенные подсказки

Пользователям Linux и других UNIX-подобных систем хорошо известна справочная система man-страниц. Man-страницы представляют собой файлы особого формата, обычно хранящиеся в каталоге `/usr/man`. В операционных системах семейства QNX используются usage-подсказки, главная особенность которых в том, что они хранятся *внутри* исполняемого файла. "Вытаскиваются" такие подсказки с помощью утилиты **`use`**. Например, получим подсказку об утилите **`ls`**:

```
use ls
```

> ***Примечание.***
> Утилита **`use`** ищет файл для извлечения подсказки в переменной `PATH`.

Добавить в исполняемый файл Neutrino usage-подсказку очень просто. Для этого надо выполнить команду:

```
usemsg program text
```

где **`program`** — наша исполняемая программа, а `text` — текстовый файл, содержащий usage-подсказку, которую мы хотим добавить.

Глава 2. Основы разработки приложений и командных сценариев

Научить стучать по клавиатуре можно и обезьяну, а вот головой думать...

Андрей Сеньков

Эта глава, как и предыдущая, предназначена для начинающих пользователей QNX Neutrino, но касается она преимущественно основ разработки программного обеспечения. В данной главе рассмотрены следующие вопросы:

- обзор инструментальных средств, входящих в комплект разработчика QNX Momentics Professional Edition;
- способы создания и уничтожения процессов;
- создание динамически и статически компонуемых библиотек;
- POSIX-механизмы управления жизненным циклом процесса и взаимодействия между процессами из собственного приложения;
- написание сценариев для командного интерпретатора.

2.1. Инструментарий QNX Momentics

В состав QNX Momentics входят разнообразные инструменты, необходимые разработчику. Для некоторых из этих инструментов, думаю, нужны отдельные книги, чтобы подробно рассказать об их возможностях и приемах работы с ними. Но и небольшой обзор, представленный вашему вниманию, полагаю, вполне уместен и небезынтересен.

2.1.1. Системы программирования

Основной системой программирования, входящей в комплект поставки QNX Momentics, является GNU Compiler Collection (GCC). В нее входят компиляторы C и C++, а также различные инструменты, необходимые для создания приложений.

Кроме того в состав QNX Momentics версии для Neutrino входят средства разработки Java на основе JIT-компилятора IBM j9.

2.1.2. Комплекты разработчика

Комплекты разработчика могут поставляться либо как отдельные продукты, либо в составе QNX Momentics. К основным комплектам можно отнести:

- DDK (Driver Development Kit) — пакеты разработки драйверов. Включает пакеты по типам драйверов: Audio DDK, Network DDK, Graphics DDK и т. д. Каждый DDK представляет собой что-то вроде проекта разработки драйвера конкретного устройства. Слова "что-то вроде" я употребил потому, что DDK содержит исходные тексты только тех файлов, которые требуется модифицировать для создания нового драйвера, — неизменная (по мнению разработчиков) часть поставляется в двоичном виде;
- TDK (Technology Development Kits) — дополнительные пакеты, в которые включены сгруппированные по функциональному назначению программные компоненты, как правило, отсутствующие в дистрибутивах операционных систем реального времени, но имеющиеся в большинстве дистрибутивов UNIX-подобных операционных систем реального времени. Например, Extended Networking TDK (Net TDK) содержит компоненты поддержки протоколов IPSec, IPv6, SCTP, SNMP v2, а также IP Filter, сервера DHCP, DNS и т. д. Есть еще Symmetric Multiprocessing TDK, 3D Graphics TDK и ряд других. Достоинством TDK является отсутствие лицензионных отчислений за использование компонентов TDK в тиражируемых целевых системах. Кроме того, большинство TDK можно купить с исходными текстами (это, конечно, будет стоить дороже);
- BSP (Board Support Packages) — дополнительные пакеты для работы QNX Neutrino на процессорных платах различных производителей (разумеется, если существует версия самой Neutrino для данной процессорной архитектуры). Включают специализированные модули IPL, Startup и, при необходимости, драйвер флэш-памяти, интегрированной с материнской платой. BSP можно адаптировать для неподдерживаемых процессорных плат, например, для плат собственной разработки. Для этого в составе QNX Momentics PE есть исходные коды IPL, Startup и драйверов для

доступа к ППЗУ. Несколько подробнее об IPL и Startup рассказано в *главе 6*;

- Embedding Tools — средства встраивания. К ним относятся инструменты построения загружаемых образов QNX Neutrino, образов встраиваемых файловых систем, комбинирования образов, работы с флэш-памятью и т. п. Эти средства входят в состав QNX Momentics;
- SAT (System Analysis Toolkit) — набор средств, необходимых для системного профилирования, т. е. для выполнения трассировки событий микроядра и т. п. Эти средства имеются в QNX Momentics (по крайней мере, в Professional Edition).

Кроме того, в состав QNX Momentics могут входить средства, необходимые для управления режимом некоторых типов устройств питания (Power management framework), комплект для построения специализированных вариантов протокола Qnet (Qnet SDK) и другие полезные вещи.

2.1.3. Интегрированная среда разработки

При выборе элемента меню **Launch > QNX Momentics 6.3.0 > Integrated Development Environment** открывается окно, называемое *рабочим местом* (workbench). При первом запуске QNX IDE создаст в домашнем каталоге пользователя папку с именем `workspace` (рабочее пространство). В рабочем пространстве будут размещаться папки проектов, с которыми IDE и будет работать.

Методы работы в IDE достаточно традиционны для программных продуктов такого рода. Особенностью QNX IDE является его внутренняя структура: использование "движка" Eclipse, написанного на языке Java. Eclipse, по сути, реализует микроядерную архитектуру для инструментов разработки — "движок" является механизмом подключения и взаимодействия *модулей расширения* (plug-in), к которым относятся различные редакторы и представления. *Редактором* (editor) называется модуль расширения, позволяющий просматривать и/или модифицировать ресурсы, например, редактор С-файлов, редактор файлов `Makefile`. *Представлением* (view) называется модуль расширения, позволяющий манипулировать ресурсом, отображая (представляя) его в каком-то логическом виде. К представлениям относятся, например, различные навигаторы и таблицы свойств элементов проектов.

Набор редакторов и представлений, оптимизированный для выполнения какой-либо специализированной задачи, составляет инструментальный профиль, который называется *перспективой* (perspective). При первом старте IDE вы увидите окно **Resource** одноименной перспективы. Кроме нее, есть ряд предопределенных перспектив, например:

- C/C++ Development — для проектов на языках C/C++;
- Java — для Java-проектов;
- Debug — для отладки программ;
- Plug-in Development — для разработки новых элементов IDE;
- QNX Application Profiler — для профилирования разработанных программ;
- QNX System Builder — для формирования встраиваемых образов QNX и загрузки их на целевые ЭВМ;
- QNX System Profiler — для визуализации трассы событий ядра, построенной с помощью пакета System Analysis Toolkit;
- QNX System Information — для мониторинга процессов, выполняющихся на целевых системах.

По сути дела, перспектива представляет собой файл в XML[1]-формате, описывающий используемые редакторы и представления, а также их размещение на "рабочем месте". Поэтому любые имеющиеся редакторы и представления можно разместить так, как нравится, и объявить полученную конфигурацию рабочего места как новую перспективу. Созданные таким образом свои перспективы можно удалять, стандартные — нельзя.

На рис. 2.1 представлено окно **Debug** одноименной перспективы.

Рис. 2.1. Окно перспективы Debug

Выбрав элемент меню **Window > Properties**, можно задать различные настройки "рабочего места", например:

- **Perform build automatically on resource modification** — выполнять сборку проекта при сохранении изменений в каком-либо ресурсе проекта;

[1] XML (eXtensible Markup Language) — расширяемый язык разметки.

- **Save all modified resources automatically prior to manual build** — сохранять измененные ресурсы при запуске сборки проекта вручную;
- **Link Navigator selection to active editor** — активизировать редактор ресурса при выделении имени ресурса в представлении **Navigator**. И наоборот — при активизации редактора с открытым ресурсом автоматически выделять имя ресурса в навигаторе.

Можно задавать массу других параметров: способ открытия новых перспектив и проектов, размещение вкладок редакторов и представлений, параметры сравнения разных версий одного файла, параметры хранения истории изменения ресурсов и т. п. Кроме того, можно настраивать различные аспекты поведения инструментов, например зависимости между проектами, указывать необходимость генерации отладочных версий файлов и т. д.

2.1.4. Средства поддержки визуального моделирования

Моделировать приложения для QNX Neutrino можно с помощью самого популярного инструмента визуального моделирования — IBM Rational Rose RealTime (сокращенно — RoseRT).

В основе линейки продуктов IBM Rational Rose лежит использование языка UML (Unified Modeling Language — унифицированный язык моделирования), позволяющего рассматривать разрабатываемый программный продукт в различных ракурсах с помощью диаграмм различного типа, например, диаграммы сценариев (use case diagram), диаграммы классов (class diagram), диаграммы последовательности (sequence diagram), диаграммы топологии (deployment diagram) и т. д.

Диаграммы сценариев, например, позволяют описать и задокументировать желаемое поведение системы с точки зрения взаимодействия с ней внешних субъектов (в терминологии UML, "актеров" — actors). Диаграммы последовательностей используются для точного определения логики взаимодействия приложения с каждым из субъектов. Лежащая в основе UML модель интегрирует все диаграммы во внутренне согласованную систему.

Однако применение UML в "чистом" виде не позволяет нам полностью смоделировать поведение сложной системы реального времени во времени. Для этого в RoseRT включена поддержка языка моделирования ROOM (Realtime Object-Oriented Modeling — объектно-

ориентированное моделирование систем реального времени). Считается, что структуру сложной системы реального времени можно полностью описать комбинацией диаграмм классов и диаграмм взаимодействий. Для моделирования структуры используются следующие конструкции:

- *капсулы* — представляют собой физические (возможно распределенные) архитектурные объекты, взаимодействующие со своим окружением через один или несколько портов;
- *порт* — это объект, реализующий специфический интерфейс. Каждый порт капсулы играет конкретную роль во взаимодействии капсулы с другими объектами и ассоциирован с протоколом, определяющим корректный поток информации (сигналов) между соединенными портами капсул. Ограничение взаимодействия между капсулами посредством портов полностью отделяет внутреннюю реализацию капсул от знания об их окружении;
- *коннекторы* — служат абстрактным представлением коммуникационных каналов, соединяющих два или более портов. По сути коннекторы представляют ключевые коммуникационные отношения между капсулами. Эти отношения архитектурно значимы, поскольку они определяют, какие капсулы могут влиять друг на друга непосредственно.

Идея заключается в том, что капсулы по протоколу передают друг другу сигналы. На получение капсулой сигнала можно назначить триггер, переводящий капсулу в следующее состояние в соответствии с диаграммой состояний капсулы. Таким образом, капсула — это обособленная машина состояний, реагирующая на внешние сигналы и посылающая сигналы другим капсулам. Спроектированную модель поведения можно откомпилировать и даже отладить. При этом на экране во встроенном в RoseRT отладчике можно увидеть, как передаются сигналы и как капсулы переходят из одного состояния в другое. RoseRT поддерживает генерацию кода на языках C, C++ и Java, а также обратное проектирование (формирование диаграмм на основе исходного кода) на языках C и C++.

Для моделирования приложений QNX Neutrino с использованием IBM Rational Rose RealTime сначала необходимо в среде Windows инсталлировать сам этот продукт, затем развернуть самораспаковывающийся архив продукта Rational Rose Adaptation Layer. Архив можно скачать с сайта **www.qnx.com**, зарегистрировав

свою копию QNX Momentics Professional Edition (см. инструкции в разделе myQNX на сайте **www.qnx.com**).

Для того чтобы лучше ознакомиться с IBM Rational Rose RealTime, можно обратиться к официальным дистрибьюторам IBM. Они, если сочтут нужным, могут предоставить дополнительную информацию и даже оценочную версию продукта (обычно полнофункциональную, ограниченную по времени использования).

2.1.5. Средство разработки графических интерфейсов пользователя

Некоторые программы требуют наличия графического интерфейса пользователя (GUI, Graphic User Interface). Такие интерфейсы можно писать "вручную", но удобнее использовать специальный инструмент визуальной разработки графических интерфейсов пользователя Photon Application Builder (PhAB), входящий в состав QNX Momentics. Это достаточно простое и удобное в использовании средство. Оно не столь богато возможностями в сравнении с аналогичными продуктами для Windows, но для большинства задач его вполне хватает. Некоторые разработчики создают весьма сложные GUI-приложения с помощью PhAB.

При использовании PhAB функция *main()* генерируется автоматически. Вручную пишутся:

- функция инициализации приложения;
- обработчики сообщений и сигналов;
- функции-обработчики событий (callback).

Разработку приложения в среде PhAB можно условно разделить на несколько этапов.

- *Создание модулей.* Модули представляют собой блоки, которые обычно выглядят и работают как окна в большинстве приложений для Photon. Чтобы создать GUI-приложение, необходимо сначала создать хотя бы один модуль. PhAB поддерживает несколько типов готовых модулей, таких как окно, диалоговое окно (диалог), меню, файл-селектор и др.;
- *Добавление виджетов.* После создания модуля добавляются необходимые виджеты (widget) и, если необходимо, их атрибуты (ресурсы) корректируются с помощью редактора ресурсов;

- *Прикрепление обработчиков события*. У каждого виджета есть набор ассоциированных с ним событий. Например, у кнопки есть события вроде "кнопку нажали", "кнопку отпустили". Таким образом, прикрепление обработчиков к интересующим нас событиям это, по сути, определение логики работы GUI-приложения. Каждый виджет Photon поддерживает несколько типов обработчиков событий. Для их использования нужно добавить соответствующий код;
- *Генерация и компиляция кода*;
- *Запуск приложения на исполнение*. Проект можно запускать на исполнение как в обычном режиме, так и в режиме отладки.

Вообще, самый практичный способ изучать PhAB — это использовать его на практике. Для того чтобы освоиться с этим инструментом, целесообразно выполнить несколько пошаговых инструкций (tutorial), которые есть в составе электронной документации QNX Momentics.

2.1.6. Средства управления версиями

Для управления версиями в QNX Neutrino можно использовать CVS (Concurrent Version control System) — стандартный инструмент Linux-проектов. В QNX IDE имеется достаточно функциональный графический клиент для CVS.

> ***Примечание.***
> QNX IDE может работать в качестве клиента самой популярной коммерческой системы управления версиями для крупных проектов — Rational ClearCase.

Идея системы управления параллельными версиями проста: имеется центральный сервер с архивной копией проекта, называемой *репозитарием* (repository). Каждый программист команды, работающей над проектом, получает экземпляр проекта, называемый *рабочей копией*. После внесения изменений в любой файл проекта разработчик может внести эти изменения в центральный репозитарий. Изменения получают свои номера и всегда можно посмотреть, какие и когда были внесены изменения, вернуться к любой предыдущей версии, от любой версии начать отдельную ветвь проекта. Другими словами, система управления версиями — чрезвычайно полезный инструмент даже при индивидуальной работе, не говоря о командной.

QNX как CVS-сервер

Если требуется создать CVS-сервер под QNX, то мы можем выполнить такие действия (предупреждаю — это не догма и не панацея):

1. Создадим в системе бюджет некоторого пользователя, назовем его `cvsuser`. Домашним каталогом для этого пользователя пусть будет `/var/cvs` (подробнее создание бюджетов пользователей описано в *разд. 4.6.3*). Зарегистрируемся под именем `cvsuser`:

   `su - cvsuser`

2. Зададим переменную `CVSROOT`, указав в ней имя каталога для репозитария, и переменную `EDITOR`, указав в ней имя вашего любимого редактора (при отсутствии переменной `EDITOR` по умолчанию используется **`vi`**):

   `export CVSROOT=/var/cvs`

   `export EDITOR=ped`

3. Затем нужно создать пустой репозитарий:

   `cvs init`

   При этом в каталоге `/var/cvs` будет создан служебный каталог репозитария `CVSROOT`.

4. Для импортирования в репозитарий, например, каталога `/home/myname/myproject` нужно войти в импортируемый каталог и выполнить команду импортирования:

   `cd /home/myname/myproject`

   `cvs import myproject cbd_bc init`

   где `myproject` — имя проекта в репозитарии, `swd` — тег производителя (произвольный), `init` — тег версии (произвольный).

   На терминале запустится редактор, указанный в переменной `EDITOR` (или **`vi`**, если переменная не задана), чтобы мы могли ввести комментарий к своим действиям (CVS всегда перед тем, как что-нибудь сделать, предлагает ввести комментарий). Действие выполняется после того, как мы сохраним комментарий и закроем редактор. Репозитарий готов к использованию.

5. Теперь настроим запуск CVS-сервера. Добавим в файл `/etc/services` строку:

   ```
   cvspserver      2401/tcp
   ```

   а в файл `/etc/inetd.conf` строку:

```
cvspserver stream tcp nowait root /usr/bin/cvs cvs --allow-
root=/var/mycvs pserver
```

Заметим, что опций `-allow-root` может быть несколько.

6. Теперь можно запускать суперсервер TCP/IP **`inetd`**, который сам запустит **`cvs`** при поступлении запроса. Только для корректной работы **`cvs`** процесс **`inetd`** не должен заполучить переменную окружения `HOME` (здесь нет смысла, это баг), поэтому делаем так:

```
unset HOME
inetd
```

QNX как CVS-клиент

Теперь о клиенте CVS. Клиент для работы также оперирует переменной `CVSROOT`. Значение переменной задается в таком формате:

`:`*`Метод_доступа`*`:`*`Пользователь`*`@`*`Хост`*`:`*`Репозитарий`*

Например:

```
export CVSROOT=:pserver:scvuser@192.168.3.184:/var/cvs
```

При работе в QNX IDE задавать переменную не требуется — эти параметры там вводятся в графическом режиме.

Затем следует создать рабочую копию проекта. Для этого необходимо выполнить команду Check Out: в IDE это делается путем выбора нужного элемента в контекстном меню, открываемом щелчком правой кнопки мыши на имени интересующего проекта. Вообще, в IDE с CVS работать, конечно, удобно — все действия можно выполнять щелчком мыши (не надо помнить командно-строковых директив) и результат получаем в графическом виде. Например, так выглядит окно сравнения модифицированного файла с одной из его предыдущих версий (рис. 2.2).

Рис. 2.2. Сравнение версий файла

Сразу видно — что удалено, что добавлено, что изменено.

Вы, наверное, заметили, что для доступа к репозитарию используется только одно имя, в нашем случае `cvsuser`. Для того чтобы регулировать доступ к репозитарию, так сказать, индивидуально, нужно, по всей видимости, добавить несколько CVS-пользователей. Регистрировать каждого такого пользователя в системе — непрактично. Проще настроить отображение CVS-пользователей на какого-нибудь одного QNX-пользователя, например, на `cvsuser`. Для этого в папке

/var/cvs/CVSROOT создадим файл passwd. Формат файла прост, одна строка соответствует одному CVS-пользователю:

Имя_пользователя_CVS:*Зашифр_пароль*:*Имя_пользователя_QNX*

Создадим CVS-пользователей myself и basily без пароля:

```
myself::cvsuser
basily::cvsuser
```

Не забудьте только сделать владельцем этого файла пользователя cvsuser:

```
chown cvsuser /var/cvs/CVSROOT/passwd
```

Теперь клиенты могут регистрироваться на CVS-сервере, используя CVS-имена myself и basily (без паролей). Пароли следует задавать в зашифрованном виде, по аналогии с /etc/shadow *(см. разд. 4.6.2)*. Для шифрования используется та же стандартная функция *crypt()*. Можно написать свою утилиту, использующую фукнкцию *crypt()*, но мне, например, лень. Проще всего использовать учетную запись пользователя cvsuser (для надежности можно в качестве login shell этого пользователя указать /dev/null — это не позволит использовать учетную запись cvsuser для входа в систему). Другими словами, задаете любой пароль для cvsuser и копируете его зашифрованное представление из /etc/shadow в файл CVSROOT/passwd между двоеточиями. Для каждого CVS-пользователя, как видно из формата файла CVSROOT/passwd, можно задать свой CVS-пароль.

2.2. Управление процессами

Использование процессов, работающих в изолированных друг от друга адресных пространствах, для выполнения различных задач — отличительная черта всех операционных систем семейства QNX. Такой подход в первую очередь позволяет повысить надежность разрабатываемых систем.

2.2.1. "Шо цэ такэ?"

Процесс — это выполняющаяся программа. Грубо говоря, процесс состоит из образа процесса и метаданных процесса. *Образом процесса* будем называть совокупность кода (т. е. инструкций для процессора — выполнение этих инструкций и есть выполнение программы) и данных (как раз ими манипулируют с помощью инструкций). *Метаданные процесса* — это информация о процессе, которая хранится в структурах

данных операционной системы и сопровождается операционной системой. Метаданными является, например, информация о физическом размещении кода и данных в оперативной памяти, а также атрибуты процесса, к которым относятся:

- идентификатор процесса (Process ID, PID) — уникальный номер, присваиваемый процессу при его порождении операционной системой;
- идентификатор родительского процесса (Parent PID, PPID) — PID процесса, породившего данный процесс, т. е. выполнившего запрос к операционной системе для создания данного процесса;
- реальные идентификаторы владельца и группы (User ID, UID и Group ID, GID) — номер, позволяющий механизмам защиты информации от несанкционированного доступа (НСД) определять, какому пользователю принадлежит процесс и к какой группе пользователей принадлежит этот пользователь. Эти идентификаторы присваиваются при регистрации пользователя в системе или командному интерпретатору (login shell), если выполнялась командно-стоковая регистрация через утилиту **`login`**, или графической оболочке Photon, если регистрация выполнялась в графическом режиме через утилиту **`phlogin`**. Процессы, запускаемые пользователем, наследуют UID и GID той программы, из которой они запускаются (т. е. *родительского процесса*);
- эффективные идентификаторы владельца и группы (Effective UID, EUID и Effective GID, EGID) — предназначены для повышения гибкости механизмов защиты информации от НСД. Пользователь при наличии соответствующих полномочийможет в ходе работы менять эффективные идентификаторы. При этом реальные идентификаторы не меняются. Механизмы, реализующие дискреционную защиту информации от НСД, для проверки прав доступа используют эффективные идентификаторы;
- текущий рабочий каталог — путь (разделенный слэшами список каталогов), который будет автоматически добавляться к относительным именам файлов. Выводится на экран командой **`pwd`**;
- управляющий терминал — терминал, с которым связаны потоки ввода, вывода и ошибок;
- маска создания файлов (umask) — атрибуты доступа, которые будут заданы для файла, созданного процессом;

Примечание
Подробно файловые атрибуты в QNX Neutrino описаны в *главе 4*.

- значение приоритета;
- дисциплина диспетчеризации.

Примечание
Подробно приоритеты и дисциплины диспетчеризации в QNX Neutrino описаны в *главе 3*.

- использование ресурсов процессора (статистика по времени выполнения программы) — включает: время выполнения программы в прикладном контексте (user time — время выполнения инструкций, написанных программистом), время выполнения в контексте ядра (system time — время выполнения инструкций ядра по запросу программы, т. е. системных вызовов), суммарное время выполнения всех дочерних процессов в прикладном контексте, суммарное время выполнения всех дочерних процессов в контексте ядра.

Программа начинает выполняться с *точки входа* — функции *main()*. Проиллюстрируем, каким образом программа может получить информацию о значении своих атрибутов (файл `process.c`):

```
#include<stdlib.h>
#include <sys/resource.h>

int main(int argc, char **argv, char **env)
{
      struct rusage r_usage;

      printf("\nProcess Informaition:\n");
      printf("Process name = \t\t%s\n", argv[0]);
      printf("User ID = \t\t< %d >\n", getuid());
      printf("Effective User ID = \t< %d >\n", geteuid());
      printf("Group ID = \t\t< %d >\n", getgid());
      printf("Effective Group ID = \t< %d >\n", getegid());
      printf("Process Group ID = \t< %d >\n", getpgid(0));
      printf("Process ID ( PID )= \t< %d >\n", getpid(0));
      printf("Parent PID ( PPID )= \t< %d >\n", getppid(0));
      printf("Process priority= \t< %d >\n", getprio(0));
      printf("Processor utilisation:\n");
      getrusage (RUSAGE_SELF, &r_usage);
```

```
      printf("\t< user time = %d sec, %d microsec >\n",
r_usage.ru_utime.tv_sec,  r_usage.ru_utime.tv_usec);
      printf("\t< system time = %d sec, %d microsec >\n",
r_usage.ru_stime.tv_sec,  r_usage.ru_stime.tv_usec);
      return EXIT_SUCCESS;
}
```

Жизненный цикл процесса можно разделить на четыре этапа:

- создание процесса *(см. разд. 2.2.2)*;
- загрузка образа процесса;
- выполнение процесса;
- завершение процесса *(см. разд. 2.2.3)*.

2.2.2. Создание процесса

"Предком" всех процессов является процесс *Администратор процессов* (процесс **procnto**), идентификатор PID которого равен 1. Остальные процессы порождаются в результате вызова соответствующей функции другим процессом, именуемым *родительским*. Таких функций несколько:

- семейство функций *exec()* — отличаются набором аргументов функций, заменяют образ вызвавшего процесса указанным исполняемым файлом;
- *fork()* — создает дочерний процесс путем "клонирования" родительского процесса;

  Примечание

  Долгое время в UNIX-подобных системах новые процессы порождались путем вызова сначала функции *fork()*, а затем из дочернего процесса-клона — функции семейства *exec()*.

  Рекомендуется использовать *fork()* только в однопоточных программах. Подробнее потоки описаны в *главе 3*.

- *vfork()* ("виртуальный *fork()*") — используется как "облегченная" альтернатива паре вызовов *fork()-exec()*. В отличие от стандартной функции *fork()*, она не выполняет реального копирования данных, а просто блокирует родительский процесс, пока дочерний не вызовет *exec()*;

- семейство функций *spawn*()* — сразу порождает дочерний процесс, загрузив указанный исполняемый файл. Наиболее эффективный способ порождения процессов в QNX Neutrino;
- *system()* — порождает shell для выполнения командной строки, указанной в качестве аргумента.

Посмотрим, как применяют функцию *fork()* на примере файла `fork.c`. В этом примере видно, что функция *fork()* возвращает целое число, которое в родительском процессе равно нулю, а в дочернем — идентификатору процесса:

```
#include<stdlib.h>
#include <sys/resource.h>
int main(int argc, char **argv, char **env)
{
      pid_t pid;
      char *Prefix;
      Prefix = (char *) malloc (sizeof(char));
      pid = fork();

      if (pid == 0)      sprintf ( Prefix, "CHILD:");
      else sprintf ( Prefix, "PARENT:");

      printf("%s Process name = %s\n", Prefix, argv[0]);
      printf("%s PID = %d\n", Prefix, getpid(0));
      printf("%s PPID = %d\n", Prefix, getppid(0));
      return EXIT_SUCCESS;
}
```

> ***Примечание***
>
> На всякий случай напомню — применение функции *fork()* в многопоточных приложениях не рекомендуется.

На примере `exec.c` проиллюстрируем порождение нового процесса с помощью комбинации вызовов *vfork()* и *exec()*:

```
#include<stdlib.h>
#include <sys/resource.h>
int main(int argc, char **argv, char **env)
{
      pid_t pid;
      pid = vfork();
```

```
      if (pid == 0)
      {
            execlp("process", "process", NULL);
            perror("Child");
            exit( EXIT_FAILURE);
      }
      waitpid(0, NULL, 0);
      printf("Parants's PID = %d\n", getpid(0));
      printf("Parants's PPID = %d\n", getppid(0));
      return EXIT_SUCCESS;
}
```

В файле `spawn.c` представлен пример наиболее простого и быстрого способа порождения нового процесса:

```
#include<stdlib.h>
#include <process.h>
int main(int argc, char **argv, char **env)
{
      spawnl ( P_WAIT, "process", "process", NULL );
      printf("Parants's PID = %d\n", getpid(0));
      printf("Parants's PPID = %d\n", getppid(0));
      return EXIT_SUCCESS;
}
```

Загрузка процесса выполняется операционной системой (Администратором процессов) и заключается в помещении сегментов кода и данных исполняемого файла в оперативную память ЭВМ. После загрузки начинается выполнение функции *main()*.

2.2.3. Завершение процесса

Завершить процесс можно, послав ему сигнал с помощью утилиты **`slay`** или **`kill`** в командной строке, или из программы с помощью таких функций, как *kill()*, *sigqueue()* и др.

Завершение процесса выполняется в две стадии.

- На первой стадии происходит "физическое" уничтожение процесса, т. е. закрываются открытые файлы, освобождается оперативная память и т. д. Эта стадия осуществляется потоком-завершителем Администратора процессов, выполняющимся от имени уничтожаемого процесса.

- На второй стадии уничтожается структура данных о процессе, хранящаяся в операционной системе. Эта стадия выполняется Администратором процессов внутри самого себя.

Процесс, если во время его работы не произошло никаких фатальных сбоев, обычно завершается путем вызова функции *exit()* или выполнением в функции *main()* оператора *return*. Число, передаваемое при этом в качестве аргумента функции *exit()* или с оператором *return*, называют *кодом возврата* программы. Если программа запускалась из командной строки в интерактивном режиме, то ее код возврата можно посмотреть в служебной переменной окружения ?:

```
echo $?
```

Примечание

Для того чтобы в родительском процессе прочесть код возврата дочернего процесса, можно воспользоваться какой-либо из функций семейства *wait*()*.

Обратите внимание, что из всего этого семейства POSIX-функциями являются только *wait()* и *waitpid()*.

Так вот, после окончания первой стадии уничтожения процесса и до того, как родительский процесс вызвал функцию *wait*()*, Администратор процессов не заканчивает вторую стадию уничтожения, а завершаемый процесс находится в состоянии, называемом "зомби". То есть "зомби" — это такой процесс, который физически уже не существует, но его код завершения все еще хранится в Администраторе процессов.

Примечание

Потоки процесса-"зомби" находятся в состоянии "DEAD-блокирован". Кстати, а что если функция *wait*()* вызвана до завершения дочернего процесса? Тогда родительский процесс (вернее, вызвавший эту функцию поток) станет WAIT-блокированным и будет оставаться в таком состоянии до завершения дочернего процесса.

При порождении дочернего процесса с помощью функции *spawn*()* можно указать флаг P_NOWAITO. В этом случае порожденный процесс никогда не сможет стать процессом-"зомби", т. к. Администратор процессов не будет хранить код возврата. Соответственно, нельзя будет использовать функции *wait*()* для получения кода возврата.

Добавим, что существует функция *atexit()*, позволяющая зарегистрировать для процесса функции-деструкторы, т. е. функции, вызываемые при нормальном завершении процесса непосредственно

перед завершением. Таких деструкторов можно зарегистрировать 32 штуки, причем вызываться они будут в порядке LIFO ("Last In — First Out", т. е. последняя зарегистрированная функция-деструктор будет вызвана первой).

2.2.4. Получение информации о процессах

Для быстрого получения информации о текущем состоянии процессов и потоков в Neutrino используют несколько утилит:

- **ps** — основная POSIX-утилита для мониторинга процессов. Она включена в QNX как для совместимости POSIX, так и для удобства администраторов, недавно работающих в QNX;
- **sin** — весьма информативная QNX-утилита мониторинга процессов. С помощью **sin** можно, задав соответствующую опцию, получить информацию о процессах на другом узле сети Qnet. По умолчанию **sin** выдает для каждого процесса следующую информацию — PID, размер кода, размер стека и использование процессора. С помощью аргументов-команд можно получить дополнительную информацию:
  - **args** — аргументы процессов;
  - **cpu** — использование ЦПУ;
  - **env** — переменные окружения процессов;
  - **fds** — открытые файловые дескрипторы;
  - **flags** — флаги процессов;
  - **info** — общую информацию о системе;
  - **memory** — память, используемую процессами;
  - **net** — информацию об узлах сети;
  - **registers** — состояние регистров;
  - **signals** — сигнальные маски;
  - **threads** — информацию по потокам;
  - **timers** — таймеры, установленные процессами;
  - **users** — реальные и эффективные идентификаторы владельцев и групп процессов.

Для выполнения команд достаточно ввести первые два символа, например команда **`sin flags`** равнозначна команде **`sin fl`**. У утилиты **`sin`** есть вариант с графическим интерфейсом — утилита **`psin`** (рис. 2.3).

Рис. 2.3. Окно **System Process Inspector** утилиты **`psin`**

`pidin` — эта утилита появилась в QNX только с 6-й версии и предназначена для получения детальной информации о потоках.

Кроме этих утилит, есть эффективное средство мониторинга процессов в составе QNX IDE — перспектива QNX System Information. Чтобы воспользоваться этим средством, нужно запустить целевой агент **`qconn`** на узле, подлежащем мониторингу. Если требуется выполнить мониторинг локального узла, значит, целевой агент **`qconn`** нужно запустить на локальной ЭВМ. Окно перспективы QNX System Information показано на рис. 2.4.

Рис. 2.4. Окно **QNX System Information** одноименной перспективы, вкладка **System Summary**

В левой части окна отображается список текущих процессов, а содержимое правой части зависит от выбранной вкладки (на рис. 2.4 показана вкладка **System Summary**).

2.2.5. "Посмертная" диагностика процессов

"Посмертная" диагностика процессов основана на использовании программы **`dumper`**, сохраняющей на диске образ "аварийного" процесса — так называемый *core-файл* (иногда именуемый core-образом процесса). Эту программу можно использовать и для немедленного получения образа какого-либо работающего процесса. Если программа **`dumper`** запущена системой, то по умолчанию она сохраняет core-образы в каталоге `/var/dumps` с именами в формате *`имя_процесса`*`.core`.

Полученный образ процесса можно анализировать двумя способами: либо утилитой **`coreinfo`**, сразу выдающей информацию о core-образе, либо с помощью GNU-отладчика **`gdb`**:

`gdb` *`имя_программы имя_программы`*`.core`

Анализ core-файла в отладчике можно сделать более приятным. С помощью IDE, разумеется.

> ***Примечание***
> Многие поставщики программного обеспечения требуют, чтобы при обращении в их службы технической поддержки клиенты предоставляли соответствующие core-образы.

2.3. Построение статических и разделяемых библиотек

В этом разделе мы рассмотрим:

- общие сведения о библиотеках;
- создание статической библиотеки;
- создание разделяемой библиотеки;
- использование DLL.

2.3.1. Общие сведения о библиотеках

Почти во всех рассмотренных примерах для вывода сообщений на экран использовалась функция *pirntf()*. Эта функция объявлена в заголовочном файле `stdio.h`, а ее объектный код содержится в стандартной библиотеке `libc.a`. При компоновке объектный код функции *printf()* вставляется в исполняемый файл и становится его неотъемлемой частью. Такие библиотеки называют статическими. *Статическая библиотека* представляет собой обычный архив `o`-файлов. Создается статическая библиотека с помощью утилиты **`ar`**. Для удобства имена файлов статических библиотек начинаются с `lib` и имеют расширение `a` (например, `libsocket.a`).

Существуют также разделяемые библиотеки. *Разделяемая библиотека* представляет собой файл того же формата, что и приложение (ELF, Executable and Linking Format), но, в отличие от приложения, не имеет функции *main()*. Имена файлов разделяемых библиотек начинаются с `lib` и имеют расширение `so` (от англ. Shared Object — разделяемый объект), после которого может стоять номер версии (например, `libmy.so.2`).

Разделяемые объекты предоставляют следующие преимущества:

- экономия дискового пространства и оперативной памяти;

- возможность обновления программного обеспечения путем замены разделяемых объектов, без перекомпиляции использующих их программ;
- сокращение времени загрузки запускаемой программы.

Недостатки разделяемых объектов:

- требуется дополнительное время при первом вызове разделяемого объекта для загрузки его в оперативную память;
- непреднамеренное удаление разделяемого объекта приводит к неработоспособности всех использующих его программ.

Разделяемая библиотека должна быть обязательно загружена либо до запуска приложения, либо при запуске. Однако можно загружать и выгружать разделяемые объекты с помощью специальных функций *dlopen()* и *dlclose()* по мере надобности. Такие разделяемые объекты в документации QSS, удобства ради, называются принятым в Windows термином DLL (Dynamic Link Library — библиотека динамической компоновки). Другими словами, DLL отличается от Shared Library не физической реализацией, а только способом использования.

2.3.2. Создание статической библиотеки

Для иллюстрации создания библиотек будем использовать тот же проект, который рассматривался при обсуждении утилиты **make** *(см. главу 1)*. Этот "проект" состоит из файлов `defs.h`, `main.c`, `aaa.c` и `bbb.c`.

Напишем Make-файл `static.mf` для создания статической библиотеки:

```
OBJS = aaa.o bbb.o ccc.o
.PHONY=clean
libmy.a: clean $(OBJS)
      ar -q libmy.a $(OBJS)
$(OBJS): defs.h
clean:
      $(RM) *.o libmy.a
```

Создадим статическую библиотеку `libmy.a`, выполнив команду:

```
make –f static.mf
```

При этом сначала будет сгенерировано три объектных файла `aaa.o`, `bbb.o` и `ccc.o`, из которых с помощью утилиты **ar** будет создана статическая библиотека `libmy.a`. Теперь библиотеку можно использовать для создания других программ:

```
qcc –o my_static –L ./ main.c -lmy
```

С помощью опции **-l** мы указали компоновщику, в какой библиотеке искать нужные объектные модули, а с помощью опции **-L** — где искать статическую библиотеки кроме стандартных путей.

> ***Примечание***
> Обратите внимание, что при указании опции **-l** мы отбросили префикс `lib` и постфикс `.a`. Компоновщик сам добавит их к указанному нами фрагменту имени `my`.

2.3.3. Создание разделяемой библиотеки

Теперь напишем Make-файл `shared.mf` для создания разделяемой библиотеки:

```
OBJS = aaa.o bbb.o ccc.o
CFLAGS+=-shared
.PHONY=clean
libmy.so: $(OBJS)
	$(CC) -o $@ -shared $(OBJS)
$(OBJS): defs.h
clean:
	$(RM) *.o libmy.so
```

Создадим разделяемую библиотеку `libmy.so`, предварительно удалив старые объектные файлы:

```
make –f shared.mf clean
make –f shared.mf
```

Сначала будет сгенерировано три объектных файла `aaa.o`, `bbb.o` и `ccc.o`, при этом компиляция этих файлов будет выполнятся с опцией **-shared.** Затем в результате компоновки из объектных файлов будет создана разделяемая библиотека `libmy.so`. Теперь библиотеку можно использовать для создания других программ:

```
qcc –o my_shared –L ./ main.c -Bdynamic –lmy
```

С помощью опции **-l** мы указали компоновщику, в какой библиотеке искать нужные объектные модули, а с помощью опции **-L** — в каких каталогах искать библиотеки в дополнение к стандартным путям поиска. Перед именем библиотеки мы опцией **-Bdynamic** указали компоновщику, что эта библиотека — разделяемая.

Примечание

Обратите внимание, что при указании опции **-l** мы отбросили префикс lib и постфикс .so. Компоновщик сам добавит их к указанному нами фрагменту имени my.

Чтобы можно было запускать программу **my_shared**, необходимо, чтобы динамический компоновщик знал, где искать разделяемую библиотеку libmy.so. Поиск выполняется в следующем порядке:

1. Динамический компоновщик ищет разделяемый объект в переменной окружения LD_LIBRARY_PATH программы, желающей загрузить этот объект.
2. Динамический компоновщик просматривает содержимое тега DT_RPATH динамической секции программы.
3. Динамический компоновщик ищет разделяемый объект в переменной окружения LD_LIBRARY_PATH Администратора процессов операционной системы QNX Neutrino *(см. разд. 6.2)*.

Итак, запустим программу, добавив в переменную LD_LIBRARY_PATH текущий каталог, т. е. каталог ".":

```
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
./my_shared
```

Теперь выполним команду **ls -l** и обратим внимание на размеры файлов my_shared и libmy.so. Сравним их с размерами файлов my_static и libmy.a.

Примечание

Для того чтобы жестко прописать в самом приложении путь поиска разделяемых объектов, надо заполнить тег DT_RPATH. Значение этого тега задается с помощью опции **-rpath** компоновщика **ld**. Чтобы задать эту опцию для компоновщика при компиляции с использованием утилиты **qcc**, как это и рекомендуется делать, нужно указать для **qcc** опцию **-Wl,-rpath** ***путь_к_разд_объекту***. То есть команда компиляции выглядела бы так:

```
qcc –o my_shared –L ./ main.c -Bdynamic \
–lmy –Wl,-rpath ./
```

2.3.4. Использование DLL

Напишем простую программу dll.c, использующую функцию *aaa()* из разделяемой библиотеки libmy.so:

```
#include<dlfcn.h>
#include"defs.h"
int main()
{
      void *dll;
      void (*my_funk)();

      printf("I'm main\n");
      dll = dlopen("libmy.so", RTLD_NOW);
      if (!dll)
      {
            perror("dll");
            exit(1);
      }
      my_funk=dlsym(dll, "aaa");
      (*my_funk)();
      return 0;
}
```

Эта программа работает следующим образом. Сначала с помощью функции *dlopen()* загружаем разделяемый объект `libmy.so`. Затем с помощью функции *dlsym()* определяем адрес нужной нам функции *aaa()* внутри модуля `libmy.so`. И, наконец, по полученному адресу вызываем функцию *aaa()*.

2.4. POSIX-механизмы взаимодействия между процессами

Процессы операционной системы QNX Neutrino, в каких бы "родственных" отношениях они ни находились, выполняются каждый в своем изолированном адресном пространстве и не могут обмениваться информацией без использования механизмов *межпроцессного взаимодействия* (IPC, InterProcess Communication). В QNX Neutrino реализован ряд таких механизмов, как стандартных, так и уникальных. В этой главе мы рассмотрим некоторые из стандартных механизмов (читайте — POSIX-механизмов) IPC:

- именованные и неименованные программные каналы;
- разделяемая память;
- очереди сообщений POSIX.

Кроме того, мы поговорим об именованных и неименованных семафорах.

2.4.1. Программные каналы

Программные каналы достаточно широко используются для обмена данными между процессами. В нашем распоряжении два типа программных каналов — неименованные и именованные. Каждый из этих типов обладает особенностями, которые следует принимать во внимание при организации межпроцессного взаимодействия.

Неименованные программные каналы

Неименованный программный канал — это механизм, который командный интерпретатор использует для создания конвейеров *(см. главу 1)*. Этот механизм реализован в администраторе ресурсов **`pipe`** *(см. главу 3)* и обеспечивает передачу данных через промежуточный буфер в оперативной памяти (в адресном пространстве администратора **`pipe`**). Неименованный канал создается функцией *pipe()*, которой в качестве аргумента передается массив из двух целых чисел. В этот массив записывается два файловых дескриптора, один из которых в последующем используется для записи информации, другой — для чтения.

К достоинствам неименованных программных каналов можно отнести то, что они являются стандартным POSIX-механизмом IPC, а также то, что это — достаточно быстрый механизм. К недостаткам — то, что обмен данными возможен только между процессами-"родственниками", т. к. процесс может получить файловые дескрипторы для работы с неименованным каналом, только наследуя их от родительского процесса.

Именованные программные каналы

Именованные каналы — это POSIX-механизм, поддержка которого реализована в файловой системе Neutrino. В основе механизма лежит особый тип файла — FIFO *(см. главу 4)*, выполняющего функцию буфера для данных. Файл типа FIFO можно создать двумя способами:

- из командной строки — утилитой **`mkfifo`**;
- из программы — функцией *mkfifo()*.

Поскольку FIFO — это файл, имеющий имя, то, во-первых, работа с ним выполняется практически так же, как с обычным файлом (*open()*, *read()*,

write(), *close()* и т. п.), во-вторых, именованный канал медленнее неименованного, но данные, записанные в именованный канал, сохраняются в случае сбоя (например, отключения питания).

2.4.2. Разделяемая память

Разделяемая память — чрезвычайно важный и широко используемый POSIX-механизм обмена данными большого объема (впрочем, не обязательно большого) между процессами. Использовать этот механизм можно, к примеру, выполнив на стороне каждого из взаимодействующих процессов такие действия:

1. С помощью функции *shm_open()* создается или открывается существующий регион памяти, который будет разделяться.
2. С помощью функции *shm_ctl()* задаются нужные атрибуты разделяемого региона.
3. С помощью функции *mmap()* на разделяемый регион памяти отображается фрагмент адресного пространства процесса, после чего с этим фрагментом выполняются необходимые операции (например, чтения или записи данных).
4. Для отмены отображения адресного пространства процесса на разделяемый регион используется функция *unmap()*, для закрытия разделяемого региона — функция *shm_close()*, для уничтожения — функция *shm_unlink()*.

Рассмотрим пример из двух программ — **`shm_creator`** и **`shm_user`**. Пример работает так:

1. Запускается программа **`shm_creator`**, которая создает регион разделяемой памяти, задает его параметры и отображает на него некий буфер, содержащий текстовую строку.
2. Запускается программа **`shm_user`**, которая отображает регион разделяемой памяти, созданный программой **`shm_creator`**, на свой буфер и печатает содержимое этого буфера.

Приведем исходные тексты обеих программ. Текст файла `shm_creator.c` выглядит так:

```
#include<stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <inttypes.h>
int main()
```

```
{
      int fd, status;
      void* buffer;
      fd = shm_open("/swd_es", O_RDWR | O_CREAT, 0777);
          if( fd == -1 ) {
            perror("shm_creator");
            return EXIT_FAILURE;
          }
      status = ftruncate(fd,  100);
      if (status!=0) {
            perror("shm_creator");
            return EXIT_FAILURE;
      }
      buffer=mmap(0,100,PROT_READ|PROT_WRITE,MAP_SHARED,fd,0);
      if (buffer == MAP_FAILED) {
            perror("shm_creator");
            return EXIT_FAILURE;
      }
      sprintf(buffer, "It's a nice day today, isn't it?");
      printf("shm_creator: %s\n", buffer);
      return EXIT_SUCCESS;
}
```

Итак, сначала вызываем функцию *shm_open()*:

```
fd = shm_open("/swd_es", O_RDWR|O_CREAT, 0777);
```

Первый аргумент `"/swd_es"` — имя региона разделяемой памяти (или, как еще говорят, разделяемого объекта памяти);

> ***Примечание***
>
> Обратите внимание, что когда имя разделяемого объекта начинается с символа `/`, объект будет помещен в служебный "каталог" `/dev/shmem`. То есть реальное имя создаваемого нами региона — `/dev/shmem/swd_es`.

Второй аргумент представляет собой битовую маску из нескольких флагов, к этим флагам относятся:

- `O_RDONLY` — открыть объект только для чтения;
- `O_RDWR` — открыть объект для чтения и записи;
- `O_CREAT` — создать разделяемый объект с режимом доступа, заданным третьим аргументом функции *shm_open()*. Если объект уже

существует, то флаг `O_CREAT` игнорируется, за исключением случаев, когда указан еще и флаг `O_EXCL`;

- `O_EXCL` — этот флаг используют совместно с флагом `O_CREAT`. В результате, если разделяемый объект памяти уже существует, то функция *shm_open()* завершится с ошибкой;
- `O_TRUNC` — этот флаг работает, когда объект уже существует и успешно открыт для чтения/записи. При этом размер объекта становится равным нулю (режим доступа и идентификатор владельца сохраняются).

Третий аргумент задает атрибуты доступа к разделяемому объекту *(см. главу 4)*.

Функция вернет файловый дескриптор `fd`, который в последующем и будет использоваться для доступа к данному разделяемому объекту.

Теперь нужно сделать, чтобы разделяемый объект имел нужный размер и параметры. Для этого используем функцию *shm_ctl()*:

```
ftruncate(fd, 100);
```

В качестве первого аргумента используется тот самый идентификатор объекта разделяемой памяти `fd`, который вернула функция *shm_open()*;

> ***Примечание***
> Иногда вместо функции *ftrancate()* используют функцию *shm_ctl()*.

Теперь созданный объект, имеющий нужный размер, необходимо отобразить на виртуальное адресное пространство нашего процесса:

```
buffer = mmap(0, 100, PROT_READ|PROT_WRITE, MAP_SHARED, fd,
0);
```

Первый и последний аргументы в нашем случае не нужны — они требуются при работе с физической памятью. Второй аргумент (`100`) указывает, какой величины фрагмент разделяемого объекта стоит отобразить на адресное пространство процесса (мы отобразили весь объект). Третий аргумент представляет собой битовую маску, которая может содержать несколько флагов:

- `PROT_EXEC` — разделяемый объект доступен для исполнения;
- `PROT_NOCACHE` — не кэшировать разделяемый объект;
- `PROT_NONE` — разделяемый объект недоступен;
- `PROT_READ` — разделяемый объект доступен для чтения;
- `PROT_WRITE` — разделяемый объект доступен для записи.

Четвертый аргумент определяет режим отображения региона. В нашем случае лучше задать `MAP_SHARED` (остальные режимы используются для работы с физической памятью).

Пятый аргумент — идентификатор разделяемого объекта.

Функция *mmap()* возвращает указатель на область виртуальной памяти процесса, на который отображен разделяемый объект (`buffer`). С полученым указателем мы можем поступать, как нам вздумается. Все, что мы запишем по этому адресу, будет отображено на разделяемый объект (конечно же, столько байт, сколько мы задали функции *mmap()*). Запишем в буфер текстовую строку:

```
sprintf(buffer, "It's a nice day today, isn't it?");
```

Теперь посмотрим на исходный текст программы `shm_user.c`:

```
#include<stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
int main()
{
      int fd;
      char *buffer;
      fd = shm_open("/swd_es", O_RDONLY, 0777);
          if( fd == -1 ) {
            perror("shm_user");
            return EXIT_FAILURE;
         }
      buffer = mmap(0, 100, PROT_READ, MAP_SHARED, fd, 0 );
      if (buffer == MAP_FAILED) {
            perror("shm_user");
            return EXIT_FAILURE;
      }
      printf("shm_user: %s\n", buffer);
      munmap(buffer, sizeof (buffer));
      return EXIT_SUCCESS;
}
```

Как видно из текста программы, для получения доступа к разделяемому объекту снова используется функция *shm_open()*. Для того чтобы отобразить разделяемый регион на адресное пространство процесса, используется функция *mmap()*. В результате получаем указатель `buffer`

на область виртуальной памяти процесса, который можно использовать. Распечатаем содержимое разделяемого объекта:

```
printf("shm_user: %s\n", buffer);
```

По аналогии мы можем передавать между процессами любые структуры данных. Но помните о том, что за правильность интерпретации данных, содержащихся в разделяемой памяти, отвечаете вы сами.

Хорошо если бы вы догадались спросить: а как процесс, считывающий данные из разделяемой памяти, определяет, что запись данных уже закончена и данные готовы для чтения? Ответ: никак. Чтобы избежать нарушений целостности данных, нам нужно использовать механизмы *синхронизации*.

Основным POSIX-механизмом синхронизации процессов являются *семафоры*.

2.4.3. Другие механизмы POSIX

В QNX Neutrino обеспечивается поддержка ряда других POSIX-механизмов синхронизации и межпроцессного взаимодействия. К таким механизмам можно отнести именованные и неименованные семафоры, очереди сообщений POSIX. Мы не будем рассматривать их подробно, поскольку они описаны в любой хорошей книге по UNIX.

Именованные семафоры

Механизм именованных семафоров в QNX Neutrino поддерживается с помощью администратора ресурсов **`mqueue`** *(см. главу 3)*.

Создаются именованные семафоры так:

```
sem_t * mySem;
mySem = sem_open( "/my_semaphore", O_CREAT, 0777, 5);
```

Функция *sem_open()* принимает в качестве аргумента имя семафора (в нашем случае — `/dev/sem/my_semaphore`), флаг создания семафора, атрибуты доступа к семафору и значение счетчика. Значение счетчика задает количество потоков, которые могут владеть семафором, не блокируясь.

> ***Примечание***
> Пока мы говорили только об однопоточных приложениях, поэтому в данном случае фраза "количество потоков" равнозначна фразе "количество процессов". О потоках мы будем говорить в *главе 3*.

Идея заключается в том, что если мы хотим ограничить количество потоков, работающих с ресурсом, числом 5, то мы создаем такой семафор `mySem`. Затем в программе, перед тем как использовать защищаемый ресурс (например, разделяемый объект памяти), мы должны вызвать функцию *sem_wait()*, передав ей ссылку на семафор:

```
sem_wait(mySem);
```

Первые пять потоков, которые успели вызвать *sem_wait()*, продолжат спокойно выполняться после вызова этой функции, но остальные будут заблокированы. Те потоки-счастливчики, которым удалось "проскочить", должны по окончании работы с защищаемым ресурсом вызвать функцию *sem_post()*, что означает: "Одно место освободилось". Какой же из заблокированных на семафоре потоков будет разблокирован? Тот, который имеет наибольшее значение *приоритета*. Если приоритеты равны — разблокирован будет *любой* поток на усмотрение Neutrino.

Но ложка дорога к обеду. Возможно, потоку вовсе не нужно быть заблокированным в ожидании открытия семафора — он может "захотеть" выполнять другую работу, пока ресурс занят. Для таких случаев существует функция *sem_trywait()*, которая проверяет, свободен ли семафор, и, если свободен, захватывает его. В противном случае она возвращается со статусом "занято" и поток может поступать, как знает, — заблокирован он не будет.

Весьма существенное достоинство именованных семафоров — их можно использовать для синхронизации процессов, выполняющихся на различных ЭВМ в сети.

Неименованные семафоры

Неименованные семафоры в распределенных по сети приложениях нам не помощники. Зато механизм неименованных семафоров (когда говорят просто "семафоры", имеют ввиду именно их) в QNX Neutrino поддерживается на уровне микроядра *(см. главу 3)*.

Для того чтобы создать такой семафор, используют функцию *sem_init()*, например:

```
sem_t * mySem;
sem_init(mySem, 1, 5);
```

В этом примеры мы получим семафор, на который ссылается указатель `mySem` со счетчиком `5` (как и в предыдущем примере). Обратите, пожалуйста, внимание на второй целочисленный аргумент — `1`. На самом деле *sem_init()* различает только два значения этого аргумента —

"ноль" и "не-ноль". Если задать 0, то семафор может быть использован только для синхронизации потоков *внутри* одного процесса. Если задать не-0, то семафор можно будет разделять с другими процессами данной ЭВМ посредством механизма разделяемой памяти.

Использование неименованного семафора полностью идентично использованию именованного семафора.

Очереди сообщений POSIX

Очереди сообщений POSIX реализованы в Neutrino с помощью уже известного вам администратора ресурсов **`mqueue`**. Для использования этого механизма из программы предназначены функции, среди которых есть следующие:

- создание/открытие очереди сообщений — *mq_open()*;
- отправка сообщения — *mq_send()*;
- прием сообщения — *mq_recieve()*.

Если при создании очереди сообщений POSIX имя объекта начиналось с символа "слэш" (*/*), то объект будет создан в каталоге `/dev/mqueue`. Размер файлового объекта указывает количество сообщений в очереди.

2.5. Командные сценарии

Любые программы, стандартные и собственные, входящие в состав операционной системы и доступные через вычислительную сеть, можно комбинировать произвольным образом с помощью перенаправления ввода/вывода, конвейеров и других возможностей командных интерпретаторов. Если для выполнения задачи нужно выполнить несколько команд, то удобно записать эти команды в текстовый файл, а затем указать интерпретатору shell "выполнить" этот файл. Такой файл называют *командным сценарием* (shell script — отсюда жаргонное "скрипт", или "шэльный скрипт"). По сути, интерпретатор реализует язык программирования высокого уровня. С помощью этого языка администраторы могут осуществлять управление вычислительной системой (что, в общем-то, они и делают).

Запуск сценариев на исполнение

Коль скоро shell — это еще и язык программирования, уместно начать его освоение со знаменитой программы "Hello world!". Итак, с помощью

любого текстового редактора создадим файл, скажем, `hello.sh`. В него поместим одну строку:

```
echo Hello world!
```

> ***Примечание***
> Обратите внимание, что последним символом в командном сценарии должен быть символ перевода строки. Ведь чтобы интерпретатор shell выполнил команду, введенную в командной строке, нужно нажать клавишу <Enter>, не так ли? Так что когда у вас не будет работать какой-нибудь сценарий, не поленитесь проверить его на наличие символа перевода строки в конце файла — начинающим shell-программистам помогает в 50% случаев ☺.

Хорошо, но как мы будем запускать сценарий? Я знаю по крайней мере четыре способа для этого:

- запустить интерпретатор, указав ему имя сценария в качестве аргумента или связав стандартный поток ввода интерпретатора с файлом сценария. Например:

  ```
  sh hello.sh
  ```

- запустить интерпретатор, связав стандартный поток ввода интерпретатора с файлом сценария. Например:

  ```
  sh < hello.sh
  ```

- выполнить сценарий в текущем экземпляре shell командой "точка" (`.`). Например:

  ```
  . hello.sh
  ```

- установить для сценария файловый атрибут "исполняемый" и запустить как обычную утилиту. Например:

  ```
  chmod 755 cmd
  hello.sh
  ```

Самым распространенным является последний способ. Только не забудьте, что в этом случае файл сценария должен или находится в одном из каталогов, перечисленных в переменной окружения `PATH`, либо для запуска надо указывать полное путевое имя сценария.

Команда "точка" тоже достаточно важна. Например, встраиваемая система управления базами данных реального времени Empress может использовать большое количество переменных окружения, особенно нужных при разработке приложений. Производитель поставляет командные сценарии, задающие значения этих переменных. Если запустить такой сценарий первым или третьим способом, то произойдет

вот что: запустится новый экземпляр shell, выполнит команды из сценария и … благополучно завершится, не изменив ничего в системе. А если применить команду "точка", то в нашем рабочем экземпляре shell будут установлены нужные переменные, позволяя нам наслаждаться инструментами Empress.

Все это замечательно, но вам уже известно, что в природе есть много командных интерпретаторов. Язык программирования каждого из них имеет свой синтаксис и свои особенности реализации. Как, например, при третьем способе запуска будет определяться интерпретатор, необходимый для выполнения сценария? Для этой цели в самой первой строке сценария помещают символы `#!` и, без пробела, имя интерпретатора, для которого написан сценарий, например:

```
#!/bin/bash
```

> ***Примечание***
> Такую конструкцию можно поместить только в первую строку. Во всех остальных случаях первый же символ # считается началом комментария, заканчивающегося символом конца строки.

Модифицируем наш файл `hello.sh`:

```
#!/bin/ksh
echo Hello world!  # This is comment
# This is comment too
```

Переменные и параметры сценариев

Ярким и типичным примером переменных, используемых интерпретатором shell, являются переменные системного окружения. Аналогичным способом мы можем задавать собственные переменные для наших сценариев или модернизировать существующие. Имя переменной shell выбирается так же, как это принято в большинстве языков программирования, т. е. это может быть последовательность букв, цифр и символов подчеркивания, начинающаяся с буквы или символа подчеркивания.

> ***Примечание***
> Обычно имена переменных для shell формируют из последовательности заглавных букв в сочетании с символами подчеркивания, но это просто необязательная традиция.

Переменные в shell задаются сразу со значением (можно с пустым). При объявлении переменной и она, и ее значение должны быть записаны *без пробелов* относительно символа равенства (=).

Примечание

Пробелы возле символа `=` — типичная ошибка начинающих shell-программистов.

Например, создадим переменную `MY_VAR` со значением `aaaa`:

```
MY_VAR=aaaa
```

или переменную `MY_VAR1` без значения:

```
MY_VAR1=
```

В отличие от обычных языков программирования, значение shell-переменной — всегда строка *символов*. То есть команда:

```
MY_VAR2=123
```

создает переменную `MY_VAR2`, значением которой является строка ASCII-символов "123", но никак *не число* 123!

Если для переменной задается значение, содержащее пробелы, то нужно заключить его в кавычки:

```
MY_VAR3="QNX Neutrino RTOS v.6.3.0"
```

Чтобы получить доступ к значению переменной, надо перед ее именем поставить символ `$`. Например, чтобы с помощью команды **`echo`** вывести на экран содержимое переменной `MY_VAR3`, надо выполнить такую команду:

```
echo $MY_VAR3
```

Помните, мы говорили, что результат выполнения одной команды можно передать в качестве аргумента другой команде? Точно так же результат выполнения команды можно присвоить переменной в качестве значения, например:

```
DATE=`date`
```

При этом сначала выполнится команда **`date`**, а результат ее выполнения вместо стандартного вывода приписывается в качестве значения переменной `DATE`.

Если нужно, чтобы имя переменной не сливалось с другими символами, то имя переменной заключают в фигурные скобки. Так команда:

```
echo ${MY_VAR3}beta1
```

выведет сообщение:

```
QNX Neutrino RTOS v.6.3.0beta1
```

Если требуется, чтобы сценарий запрашивал ввод значений переменных, используется команда **`read`**. Она обеспечивает работу со сценарием в диалоговом режиме, например:

read A

По этой команде сценарий остановится, ожидая ввода. Пользователь должен ввести какое-либо значение и нажать клавишу <Enter>. Введенное значение будет присвоено переменной A и сценарий продолжит выполнение. Таким способом можно присвоить значения сразу нескольким переменным, например:

read A B C

Для того чтобы попрактиковаться, напишем сценарий script1.sh:

```
#!/bin/ksh
echo –n Input variables A, B, and C:
read A B C
echo A=$A
echo B=$B
echo C=$C
```

Затем сделаем наш сценарий исполняемым и запустим:

```
chmod a+x script1.sh
./script1.sh
```

На запрос сценария Input variables A, B, and C: введем строку **111 222 333**, не забыв после этого нажать клавишу <Enter> (то, что для выполнения команды или ввода данных всегда надо нажимать клавишу <Enter>, больше повторять не буду ☺).

На экране мы увидим следующее:

```
Input variables A, B, and C:111 222 333
```

Теперь, эксперимента ради, снова запустите сценарий, но введите строку **111 222 333 444 555** и посмотрите, как изменится результат.

> ***Примечание***
> Для того чтобы можно было использовать служебные символы в составе строковых переменных, применяется экранирование — обратный слэш (\).

Если, скажем, переменной A надо присвоить в качестве значения строку $B, то это делается так:

```
A='$B'
```

Тогда команда **echo A=$A** выведет на экран: A=$B.

Мы уже говорили о том, что переменные в интерпретаторе shell имеют строковый тип, однако shell предоставляет некоторые средства,

позволяющие обрабатывать переменные как целые числа. Для этого предназначена команда **expr**. Напишем сценарий `script2.sh`:

```
#!/bin/ksh
x=50 y=4
a=`expr $x + $y`; echo a=$a
b=`expr $x - $y`; echo b=$b
c=`expr $x / $y`; echo c=$c
d=`expr $x "*"  $y`; echo d=$d
e=`expr $x % $y`; echo e=$e
```

Запустим этот сценарий. На экране получим:

```
a=54
b=46
c=12
d=200
e=2
```

Заметьте, что в результате деления получаем только целую часть результата. Остаток от деления получается с помощью операции **%**. Заметьте также, что символ умножения (звездчка — *) взят в кавычки, т. к. * — служебный символ shell, обозначающий произвольную строку.

> ***Примечание***
> Не забудьте, что переменные и знаки операций *обязательно разделяются пробелами*. Отсутствие таких пробелов — очень распространенная ошибка в shell-программировании.

Все переменные доступны только в том процессе **ksh**, в котором они были объявлены (вспомните назначение команды "точка"). Для того чтобы к переменным можно было обратиться из процессов, дочерних по отношению к данному процессу **ksh**, переменные следует "экспортировать". Это можно сделать двумя способами. Первый — создать переменную, а затем в любое время экспортировать ее:

```
VAR1=myvar
...
export VAR1
```

Второй способ — экспортировать переменную сразу при создании:

```
export VAR1=myvar
```

Посмотреть, какие из переменных данного интерпретатора экспортированы, можно, выполнив команду **export** без аргументов.

Примечание

Все переменные интерпретатора shell (экспортированные и не экспортированные) выводятся на экран командой **`set`** без аргументов.

Внутренние переменные shell

Интерпретатор shell поддерживает некоторые переменные, которым он присваивает значения самостоятельно (табл. 2.1).

Таблица 2.1. *Внутренние переменные интерпретатора*

Переменная	Значение
`?`	Код завершения последней команды (0 — нормальное завершение)
`$`	PID текущего процесса **`ksh`**
`!`	PID последнего процесса, запущенного в фоновом режиме
`#`	Число параметров, переданных в shell
`*`	Перечень параметров shell как одна строка
`@`	Перечень параметров shell как совокупность слов
`-`	Флаги, передаваемые в shell

Примечание

Различие между `$*` и `$@` заключается в том, что если `$*` имеет значение, например `"111 222 333"`, то `$@` будет иметь значение `"111" "222" "333"`.

Параметры сценариев

Как и любая программа, сценарий может принимать аргументы. Чуть ранее, говоря о внутренних (встроенных) переменных shell, я упомянул о переменных, содержащих число и перечень параметров. Как же прочитать эти параметры? Для этой цели shell поддерживает специальные переменные, имена которых состоят из одной цифры в диапазоне от 0 до 9. При этом переменная `0` содержит имя самого сценария, переменная `1` — первый параметр, переменная `2` — второй параметр и т. д. Общее число параметров, как вы помните, содержится в переменной #.

Замечательно, но если вдруг нужно передать сценарию, скажем, 30 аргументов? Передать-то, понятно, передадим — дело немудреное,

но как получить к ним доступ? А для этого есть встроенная команда **shift**. Чтобы понять принцип ее использования, вспомните, как вы просматриваете длинный документ Word: в один момент времени можно видеть только фрагмент текста, соответствующий рабочему пространству окна Word; чтобы посмотреть другие части документа, нужно перемещать рабочее пространство окна вверх или вниз по документу ("прокручивать" документ). Аналогично, переменные `1`–`9` являются этим самым "окном" для просмотра аргументов. Перемещается это окно с помощью команды **shift** на нужное число позиций. Переменная `0` всегда содержит имя сценария. Для наглядности рассмотрим пример `script3.sh`:

```
#!/bin/ksh
echo "Parameters =" ${#}
echo ------------------
echo "Param0 =" $0
echo "Param1 =" $1
echo "Param2 =" $2
echo "Param3 =" $3
echo ------------------
shift 2
echo "Param0 =" $0
echo "Param1 =" $1
echo "Param2 =" $2
echo "Param3 =" $3
```

Запустим его, задав несколько аргументов, и посмотрим, что получилось:

```
chmod a+x script3.sh
./script3.sh  AAA BBB CCC DDD EEE
```

Отладка сценариев

Для запуска сценария в подобии отладочного режима предназначена команда **set**. Есть два режима, представляющих интерес с точки зрения отладки.

- Первый режим — устанавливается командой **set -v**, указывающей интерпретатору выводить все строки, которые он считывает; отменяется командой **set +v**.

- Второй режим (при отладке используется чаще) — устанавливается командой **`set -x`**, указывающей интерпретатору выводить на экран команды перед их выполнением; отменяется командой **`set +x`**.

Операторы языка программирования Korn Shell

Как же можно написать полноценную программу, не используя циклы и проверку условий? Язык программирования Korn Shell поддерживает ряд специальных операторов (команд), основные из которых:

```
test
if
case
for
while
until
```

Оператор *test,* или *[ ]*

Оператор `test` проверяет выполнение условия. Я не встречал этот оператор в сценариях сам по себе, зато в сочетании с оператором `if` он очень распространен. Результат выполнения оператора имеет значение "истина" или "ложь". Это значит, что если требуется использовать оператор `test` в интерактивном режиме, то его результат будет содержаться в переменной `?` (0 — "истина", 1 — "ложь"). У оператора два равнозначных формата, причем в равной степени в сценариях используются оба:

`test` *`условие`*

или

`[` *`условие`* `]`

> ***Примечание***
> Между квадратными скобками и условием *обязательно должны быть пробелы*. Отсутствие этих пробелов — очень распространенная ошибка.

Условия могут быть нескольких, если можно так выразиться, типов:

- проверка файлов;
- сравнение строк;
- сравнение целых чисел.

Некоторые условия проверки файлов:

- `test -f myfile.txt` — "истина", если `myfile.txt` — обычный файл;

- `test -d mydir` — "истина", если `mydir` — каталог;
- `test -c myfile` —"истина", если `myfile` — символьное устройство;
- `test -r myfile.txt` — "истина", если `myfile.txt` доступен для чтения;
- `test -w myfile.txt` — "истина", если `myfile.txt` доступен для записи;
- `test -s myfile.txt` — "истина", если `myfile.txt` не пустой.

Некоторые условия сравнения строк:

- `test aaa = bbb` — "истина", если строка `aaa` совпадает со строкой `bbb`;
- `test aaa != bbb` — "истина", если строка `aaa` не совпадает со строкой `bbb`;
- `test -n $A` — "истина", если переменная `A` задана (не пустая);
- `test -z $A` — "истина", если `A` — пустая (т. е. не имеет значения).

Обратите внимание, что оператор `test` возвращает значение "истина", если *`условие`* — непустая строка.

POSIX определяет два стандартных "условия", которые можно использовать вместо оператора `test`, — `true` и `false`. Чему равны эти "условия", догадайтесь сами ☺.

Некоторые условия сравнения целых чисел:

- `test x -eq y` — "истина", если `x=y`;
- `test x -ne y` — "истина", если `x!=y`;
- `test x -gt y` — "истина", если `x>y`;
- `test x -ge y` — "истина", если `x>=y`;
- `test x -lt y` — "истина", если `x<y`;
- `test x -le y` — "истина", если `x<=y`.

В одном операторе `test` можно задать несколько условий, комбинируя их с помощью логических операций:

- `!` — НЕ
- `-o` — ИЛИ
- `-a` — И

Оператор *if*

В общем случае этот оператор условия имеет такой синтаксис:

```
if условие
 then список_операторов
 [elif условие
       then  список_операторов]
      [else  список_операторов]
 fi
```

где *условие* — любой оператор, возвращающий значение `true` или `false` (обычно используется оператор `test`); *список_операторов* — операторы, разделенные точкой с запятой (`;`).

Обратите внимание на оператор `elif` — это сокращение от "else if". В квадратные скобки взяты необязательные элементы конструкции. Заметим, что конструкция оператора `if` заканчивается служебным словом `fi` ("if" наоборот).

Проиллюстрируем сказанное простейшим примером — сценарием `script4.sh`:

```
#!/bin/ksh
if [ -f /tmp/myfile ]
      then cksum /tmp/myfile
      else ls /etc > /tmp/myfile; cksum /tmp/myfile
fi
```

Оператор *case*

Оператор выбора `case` имеет такой синтаксис:

```
case строка in
     шаблон) список_операторов;;
     шаблон) список_операторов;;
         ...
esac
```

где *строка* — переменная или последовательность символов, которая последовательно сравнивается с шаблоном *шаблон*. Как только *строка* и *шаблон* совпадут, выполняется соответствующий *список_операторов*. Как, вероятно, вы уже поняли, `in` и `isac` — это служебные слова, причем завершает всю конструкцию `esec` ("case" наоборот).

Примечание

Список операторов для каждого шаблона заканчивается двойной точкой с запятой `;;` (в аналогичной конструкции языка Си второй точке с запятой соответствует служебное слово `break`).

Проиллюстрируем сказанное простейшим примером — сценарием `script5.sh`:

```
#!/bin/ksh
echo -n "Input command: "
read Command
case $Command in
      a) echo Hello!;;
      b) ls –l /usr/bin;;
      c) cd /usr; pwd;;
      *) echo "Wrong command! Usage a, b, or c";;
esac
```

Как вы понимаете, шаблону `*` строка будет соответствовать всегда (в языке Си для этой цели используется служебное слово `default`).

Оператор *for*

Оператор цикла `for` имеет такой формат:

```
for имя [in список_значений]
      do
          список_операторов
      done
```

Оператор работает так. На каждой итерации цикла переменной *имя* последовательно присваиваются значения из списка *список_значений* и выполняются команды из списка *список_операторов*; *список_операторов* заключен между служебными словами `do` и `done`.

Приведем христоматийный прием использования оператора `for` — сценарий `script6.sh`:

```
#!/bin/ksh
LIST=`ls /etc`
for i in $LIST
    do
      if [ ! –d $i ]
      then cksum /etc/$i
      fi
```

```
done
```

В результате выполнения этого сценария на экран будет выведена информация с размером и контрольными суммами содержимого каталога `/etc` (без учета содержащихся в нем подкаталогов).

Если в операторе `for` нет конструкции `in` *`список_значений`*, то *`список_значений`* будет равен внутренней переменной `@` (см. ранее).

Оператор `for` не подходит для организации бесконечных циклов, т. к. он работает, перебирая элементы вполне конечного списка.

Операторы *while* и *until*

В отличие от оператора `for`, оператор `while` позволяет организовывать бесконечные циклы, т. к. следующая итерация инициируется всегда, если только условие истинно:

```
while условие
      do
          список_операторов
      done
```

В этом операторе можно задавать счетчики циклов, параметры выхода из цикла (служебное слово `break`) и т. д. Говоря кратко, оператор `while` соответствует традиционным представлениям о нем. Если *`список_операторов`* оператора `while` выполняется тогда, когда *`условие`* истинно, то в операторе `until` напротив — когда *`условие`* станет истинным, цикл завершится:

```
until условие
      do
          список_операторов
      done
```

Разумеется, цикл можно принудительно прервать командой `break`.

Пустой оператор

Иногда по правилам синтаксиса в сценарии на каком-то месте должен обязательно стоять оператор, а никаких действий предпринимать не хочется. В таком случае используется пустой оператор "двоеточие" (`:`).

Функции в сценариях

Как и в любом языке программирования, в интерпретаторе shell можно описывать функции и затем вызывать их из любого места сценария. Определить функцию очень просто:

```
имя ()
```

```
{
    список_операторов
}
```

В любом месте программы, где нужно вызвать функцию, просто указывается ее имя как команда. Функции можно передавать аргументы так же, как аргументы передаются сценарию. Мало того, доступ к аргументам выполняется в функции так же, как в сценарии, т. е. через переменные `0`–`9`. Другими словами, внутри функции переменные `0`–`9` рассматриваются как аргументы функции, а в остальных местах сценария — как аргументы сценария. Разумеется, при вызове функции новый процесс не создается. Если функция должна вернуть код завершения, то используется оператор `return`.

Обработка сигналов

Для обработки сигналов в shell предусмотрен оператор `trap`. Его синтаксис:

`trap "`*`список_операторов`*`" `*`список_сигналов`*

По сути дела, *`список_операторов`* — это своего рода обработчик сигналов *`список_сигналов`*.

Например, создадим обработчик для сигнала SIGUSR1 (с номером 16). Пусть при получении этого сигнала сценарий уничтожит содержимое каталога `/tmp`, выведет некоторое сообщение на экран и завершится с кодом 100:

```
#!/bin/ksh
echo My PID = $$
trap "rm /tmp/*; echo Good By!; exit 100" 16
while true
      do
      :
      done
```

Для проверки запустите этот сценарий. Он выведет на экран PID интерпретатора, в котором он выполняется (пусть будет 123456). Затем из другого интерпретатора (с другой виртуальной консоли или из другого псевдотерминала) выполните команду:

```
kill -s  SIGUSR1 123456
```

Посмотрите результат на терминале исходного интерпретатора.

Глава 3. Базовые механизмы

"Все дело в архитектуре".
(Слоган, придуманный маркетологами для стенда о QNX)
"Угу, вы бы еще написали — «архитектуру не пропьешь!»"
(Комментарий технарей)

В этой главе мы обсудим, как устроена ОСРВ QNX Neutrino и как можно использовать особенности ее устройства в своих программах. Для этого рассмотрим следующие темы:

- архитектура QNX Neutrino;
- потоки POSIX в Neutrino;
- синхронный механизм IPC, реализованный в микроядре;
- "импульсы" и события.

3.1. Архитектура QNX Neutrino

3.1.1. «Внутри он другой!»

Архитектура — это то, чем операционные системы семейства QNX, несмотря на большое внешнее сходство, отличаются от операционных систем семейства UNIX.

Центральным понятием в QNX является *микроядро* (microkernel). Грубо говоря, микроядро почти ничего само не делает, являясь своего рода коммутирующим элементом, к которому с помощью дополнительных программных модулей добавляется та или иная функциональность. Кстати, раньше название "Neutrino" распространялось только на микроядро, теперь — на всю ОС.

Кроме микроядра в ОСРВ QNX есть еще один важный элемент — *Администратор процессов* (Process Manager). Микроядро QNX Neutrino скомпоновано с Администратором процессов в единый модуль **`procnto`** — главный (и единственный безусловно необходимый для работы системы) компонент. Если нам надо, чтобы система реально делала какую-то работу, мы должны запустить соответствующий процесс, выполняющий эту работу. Программы, реализующие сервисные функции, называют *администраторами ресурсов* (Resource Manager). Есть администраторы ресурсов, обеспечивающие доступ к

дискам, сети и т. д. Все эти программы связаны воедино микроядром и слаженно взаимодействуют с помощью механизма сообщений.

Следует заметить, что существуют различные дополнительные варианты модуля **procnto** (не говоря о версиях для разных процессоров):

- **procnto-smp** — вариант модуля **procnto** с поддержкой симметричной многопроцессорности (входит в SMP TDK);
- **procnto-instr** — вариант модуля **procnto**, оборудованный средствами трассировки событий.
- **procnto-smp-instr** — сами догадайтесь, что это за модуль и для чего он предназначен ☺.

На самом деле функциональность ОС можно расширять не только с помощью процессов, но и с помощью библиотек динамической компоновки DLL (Dynamic Link Library). Правильнее будет сказать, что как функциональность ОС расширяется с помощью процессов, так функциональность процессов расширяется с помощью DLL. Более того, каждый администратор ресурсов может быть реализован или как программа, или как DLL.

Таким образом, в общем случае QNX Neutrino представляет собой группу процессов, работающих в изолированных адресных пространствах, взаимодействующих через микроядро и включающих:

- Администратор процессов, скомпонованный с микроядром;
- администраторы ресурсов;
- прикладные программы.

Достоинства такой архитектуры очевидны: достаточно высокая надежность (т. к. при сбое любого из системных процессов, кроме **procnto**, остальные процессы продолжают работать), простота реконфигурирования системы и добавления новых сервисов (и наоборот — простота урезания системы, создания встраиваемых конфигураций с требуемой функциональностью), простота отладки системных сервисов (т. к. это обычные программы). С другой стороны очевидны и недостатки, отсутствующие у монолитных систем: частое переключение контекста (и, как следствие, высокие накладные расходы на выполнение ряда операций), необходимость копирования данных между процессами, решающими одну задачу.

Чтобы продолжить обсуждение архитектуры QNX Neutrino, нам следует ввести понятие "потока", или "потока управления" (thread — "нить").

В предыдущей главе мы уже обсуждали процессы. Вы помните, что процесс — это выполняющаяся программа. Он включает код и данные программы, а также различную дополнительную информацию — переменные системного окружения и т. п.

Поток управления — это фрагмент процесса, содержащий непрерывную последовательность команд, которые могут выполняться параллельно с другими потоками (потоками управления) того же процесса.

Процесс является, по сути дела, контейнером потоков и содержит как минимум один поток.

Для чего вообще нужны потоки? Они весьма полезны в ряде случаев, поэтому поддержка потоков — обязательное свойство POSIX-совместимых ОС. Обычно потоки используют:

- для распараллеливания задачи на многопроцессорных ЭВМ;
- для более эффективного использования процессора (например, пока один поток ожидает пользовательский ввод, другой может выполнять расчеты);
- для облегчения совместного использования данных (все потоки процесса имеют свободный доступ к данным процесса).

Микроядро работает только с потоками и "ничего не знает" о процессах (ну, или почти ничего ☺). За процессы отвечает Администратор процессов.

На этапе выполнения поток может находиться в одном из трех состояний:

- исполнение на процессоре (RUNNING);
- ожидание процессора, или готовность к исполнению (READY);
- блокировка в ожидании освобождения некоторого ресурса (название блокированного состояния зависит от того, в ожидании какого ресурса заблокирован поток).

Есть еще состояние блокировки DEAD (в UNIX его называют ZOMBIE — "зомби"), когда физически поток уничтожен, но Администратор процессов еще сохраняет некоторые структуры данных с информацией о нем, чтобы передать родительскому потоку код завершения "умершего" потока.

3.1.2. Механизмы микроядра

Итак, микроядро— главный и обязательный компонент ОСРВ QNX Neutrino. Что же представляет из себя микроядро? Это, по сути, библиотека, содержащая несколько функций, доступ к которым прикладная программа получает с помощью специальных функций — kernel calls (давайте будем называть их по-русски, например "вызовы ядра").

> ***Комментарий***
> В монолитных системах есть термин "системный вызов", родственный в какой-то степени "вызову ядра" в QNX. Разные термины использованы сознательно, т. к. для выполнения, например, функции *write()* (в ОС UNIX это системный вызов) система QNX Neutrino на самом деле отрабатывает несколько вызовов ядра. Хотя порой вызовы ядра и называют "системными вызовами", разницу между этими понятиями всегда следует иметь в виду.

Микроядро не ведет себя, как процесс, оно не может самостоятельно захватить процессор. Другими словами, выполнение микроядра заключается в выполнении соответствующей процедуры, которая может быть вызвана одним из способов:

- какой-то процесс выполнил вызов ядра;
- возникло прерывание;
- произошла ошибка при выполнении инструкции процессора.

Обратите внимание, что Администратор процессов ведет себя как вполне обычный процесс.

Вызовы ядра в QNX Neutrino вытесняемы, т. е. если вызов ядра выполняется в интересах одного потока и более приоритетный поток тоже вызвал ядро, то "старый" низкоприоритетный вызов уступит процессор "новому". Следовательно, в QNX Neutrino (по сравнению с операционными системами, не поддерживающими вытеснение вызовов ядра):

- меньше время реакции на события, например на прерывания от аппаратуры;
- затрачиваются дополнительные ресурсы процессора на сохранение/восстановление контекста вызова ядра.

Справедливости ради надо сказать, что код вызовов ядра не однороден и, с точки зрения "вытесняемости", его можно разделить на четыре типа.

К *первому типу* отнесем моменты входа в вызов ядра и выхода из него. При этих операциях выполняется сохранение или, соответственно, восстановление контекста. Сие есть атомарная операция и в это время программные прерывания (а классический образец программного прерывания — это тот же вызов ядра), разумеется, отключены. Аппаратные прерывания никуда не денутся, но могут быть обработаны только после сохранения/восстановления контекста. Но не переживайте — это всего несколько десятков тактов процессора.

Второй тип кода вызова ядра составляет полностью вытесняемый реентерабельный код.

К *третьему типу* отнесем невытесняемый код, при выполнении которого программные прерывания встают в очередь.

Наконец, *четвертый тип* — это нереентерабельный, полностью вытесняемый код. К такому коду можно отнести проверку потока-получателя перед отправкой сообщения — нет никаких гарантий, что этот поток еще существует, когда вызов ядра, отправляющий сообщение, продолжает выполнение после вытеснения обработчиком прерывания от мыши.

Эти части могут присутствовать в вызове ядра не все сразу, они могут и перемежаться. Например, вызов ядра, выполняющийся для захвата семафора, целиком относится к третьему типу.

Микроядро выполняет следующие функции (т. е. состоит из вызовов ядра, выполняющих следующую работу):

- создание и уничтожение потоков;
- диспетчеризация потоков;
- синхронизация потоков;
- базовые механизмы IPC;
- поддержка механизма обработки прерываний;
- поддержка часов, таймеров и таймаутов.

Больше микроядро не делает ничего. Вся остальная функциональность QNX Neutrino обеспечивается Администратором процессов и администраторами ресурсов.

Диспетчеризация потоков

QNX — многозадачная ОС. Это значит, что в системе может существовать достаточно много процессов (и потоков). Причем несколько из них могут одновременно оказаться в состоянии готовности

к исполнению. Как определить, какому из этих потоков предоставить свободный процессор? Для этого микроядро использует приоритет, назначенный каждому потоку. Всего в QNX Neutrino версии 6.3 имеется 256 уровней приоритета (в Neutrino версий до 6.2.1 было 64 уровня приоритета). Самый низкий приоритет — 0, самый высокий — 255. Нулевой приоритет имеет специальный поток Администратора процессов под названием idle (что в переводе с английского означает "ленивец"), на русском техническом жаргоне его называют "холодильником". Этот поток всегда находится в состоянии готовности к исполнению READY. Не вздумайте задать своему потоку нулевой приоритет — он никогда не получит процессор!

Диспетчеризация выполняется микроядром в трех случаях:

- исполняющийся на процессоре поток перешел в блокированное состояние;
- поток с более высоким, чем у исполняющегося потока, приоритетом перешел в состояние готовности, т. е. происходит вытеснение потока (это свойство ОС называют *вытесняющей многозадачностью*);
- исполняющийся поток сам передает право исполнения на процессоре другому потоку (вызывает функцию *sched_yield()*).

Если несколько *готовых к исполнению* потоков имеют *равные* приоритеты, то повлиять на их взаимоотношения можно с помощью такого параметра потока, как *дисциплина диспетчеризации* (Scheduling Policy). Neutrino позволяет задавать для каждого потока одну из следующих дисциплин диспетчеризации:

- FIFO (First In First Out — "первый вошел — первый вышел"). Если потоку назначена дисциплина диспетчеризации FIFO, то другой поток с таким же приоритетом получит управление, только если исполняющийся поток заблокируется или сам уступит право исполнения;
- "карусельная" (Round Robin) диспетчеризация — поток исполняется в течение *кванта времени* и передает управление следующему потоку с таким же приоритетом. В QNX 6.3 квант времени (timeslice) по умолчанию равен 4 мс (для процессоров с частотой выше 40 МГц);
- спорадическая диспетчеризация — предназначена для установления лимита использования потоком процессора в течение определенного периода времени. Этот механизм заменил имевшуюся в прежних

версиях QNX адаптивную диспетчеризацию и введен в порядке эксперимента — пользуйтесь им осторожно.

При спорадической дисциплине потоку задается несколько параметров:

- нормальный приоритет (N);
- нижний приоритет (L);
- начальный бюджет (C);
- период восстановления (T);
- максимальное число пропусков восстановлений.

Когда поток переходит в состояние READY, его приоритет имеет значение N в течение интервала времени C, после чего приоритет потока снижается до значения L. Но время пребывания потока в состоянии с приоритетом L ограничено. Через интервал времени T (считая вместе с C) приоритет потока снова станет равным N. Поскольку поток при приоритете L может вообще не получить управление, задается ограничение максимального количества отложенных восстановлений, при достижении которого потоку будет принудительно возвращен приоритет N. Таким образом, спорадическая диспетчеризация гарантирует, что поток "не съест" больше C / T процессорного времени (если не он один имеет приоритет, равный N).

Может сложиться такая ситуация. Когда поток блокируется до истечения начального бюджета, то после того как поток вновь получит управление, его начальный бюджет восстанавливается. Получив управление, поток может немного поработать и заблокироваться, вновь не исчерпав начальный бюджет. И так до бесконечности. Получится, что поток использует 100% отведенного ему времени, а не C / T. Чтобы избежать такой ситуации, используется параметр *максимальное число отложенных восстановлений*. Что сие нам дает? Каждый раз, когда поток получает управление, ядро отсчитывает начальный бюджет и считает, что должно было произойти восстановление (но помните, что восстановление не происходит — ведь бюджет из-за блокировок не израсходован). Как только число таких "окончаний начального бюджета" достигнет максимального числа отложенных восстановлений, приоритет потока будет снижен до значения L — до тех пор, пока не закончится период восстановления.

В дополнение к дисциплинам диспетчеризации Neutrino поддерживает механизм клиент-управляемых приоритетов. Идея его заключается в том, что приоритет потока-сервера устанавливается равным максимальному из приоритетов потоков-клиентов, заблокированных в

ожидании освобождения этого потока-сервера. Действительно, было бы не очень здорово, если бы поток с приоритетом 100 ждал, пока закончит работу поток с приоритетом 20 и поток-сервер, наконец, получит управление, чтобы закончить обработку предыдущего запроса от потока с приоритетом, например, 5.

Существует такое понятие, как *масштаб диспетчеризации*. POSIX определяет два масштаба — процесса и системы, в QNX Neutrino реализован только масштаб системы. При диспетчеризации в масштабе процесса ресурсы процессора делятся между процессами, а уже внутри процесса "разыгрываются" между потоками. Предположим, есть 2 процесса — А и Б, А с одним потоком, а Б с пятью потоками, все шесть потоков готовы к исполнению. При диспетчеризации в масштабе процесса процессорное время было бы поровну поделено между процессами, а далее выделенное процессу время делилось бы между потоками процесса. То есть поток процесса А получил бы 50% процессорного времени, а каждый поток процесса Б — 10% процессорного времени. При диспетчеризации в масштабе системы (т. е. при единственно возможном в Neutrino способе) каждый поток получит 1/6 часть процессорного времени.

3.1.3. Администратор процессов

Итак, микроядру известны только потоки и то, как с ними обращаться, а поддержку процессов в QNX Neutrino обеспечивает Администратор процессов, скомпонованный с микроядром в единый программный модуль **`procnto`**.

Администратор процессов выполняет важные функции:

- управление процессами;
- управление памятью;
- управление пространством путевых имен.

Вопрос управления процессами мы с вами уже рассмотрели в *разд. 2.2*. Для закрепления вспомним, что процесс — это выполняющаяся программа, которая является контейнером потоков. Всю реальную работу делают именно потоки, поэтому любой процесс состоит не меньше чем из одного потока.

Процесс имеет ряд атрибутов (например, PID, EUID, UMASK и т. п.) и его жизненный цикл включает четыре этапа:

- создание — процесс может быть создан только другим (родительским) процессом, при этом Администратор процессов создает у себя необходимые структуры данных;
- загрузка кода и данных процесса в ОЗУ;
- выполнение потоков;
- завершение, которое выполняется в две стадии.

Управление памятью

Управление памятью заключается в:

- управлении защитой памяти;
- управлении разделяемой памятью (об этом механизме мы говорили в *главе 2*).

Администратор процессов QNX обеспечивает поддержку полной защиты памяти (так называемую "виртуальную память") процессов.

Каждому процессу предоставляется 4 гигабайта адресного пространства, из них код и данные процесса могут занимать пространство от 0 до 3,5 Гбайт. Диапазон адресов от 3,5 до 4 Гбайт принадлежит модулю **`procnto`** (эти значения относятся к ЭВМ на базе процессора x86, в реализациях QNX Neutrino для других аппаратных платформ они могут отличаться).

Для отображения виртуальных адресов на физическую память используются аппаратные *блоки управления памятью* (MMU, Memory Management Unit).

Управление пространством путевых имен

Пространство путевых имен является одной из особенностей ОС QNX. Дело в том, что управление ресурсами ввода/вывода не встроено в микроядро, а реализуется посредством дополнительных процессов — администраторов ресурсов. Например, записью файлов на диск управляет Администратор файловой системы, отправкой данных по сети — Администратор сети. Как же интегрировать услуги администраторов ресурсов в операционную систему? Для этого Администратор процессов предоставляет механизм пространства путевых имен.

Пространство путевых имен представляет собой дерево каталогов и файлов, в вершине которого находится корневой каталог `/` ("root"). При запуске каждый администратор ресурсов регистрирует

у Администратора процессов свою зону ответственности, или "префикс" (соответствующий ветви единого дерева). Сам модуль **procnto** при загрузке регистрирует в пространстве путевых имен несколько префиксов:

- `/` — корень ("root") файловой системы, к которому монтируются все остальные префиксы;
- `/proc/` — каталог, в который отображается информация о запущенных процессах, представленных их идентификаторами (PID);
- `/proc/boot/` — каталог, в который в виде "плоской" файловой системы отображаются файлы, входящие в состав загрузочного образа QNX;
- `/dev/zero` — устройство, которое при чтении из него всегда возвращает ноль. Используется, например, для того, чтобы заполнить нолями страницы памяти;
- `/dev/mem` — устройство, представляющее всю физическую память.

Префикс, к которому администратор ресурсов запрашивает присоединение своего поддерева, называется *точкой монтирования* (mountpoint). Точки монтирования могут перегружаться, т. е. к одной точке могут подключить свои поддеревья несколько администраторов ресурсов. Кроме того, Администратор процессов позволяет создавать символические префиксы, т. е. регистрировать в пространстве путевых имен ссылки на существующий файл:

`ln -Ps` *`существующий_файл имя_ссылки`*

Механизм пространства путевых имен позволяет Администратору процессов определять, какой из администраторов ресурсов должен выполнить поступивший запрос на ввод/вывод данных. Алгоритм будет таким:

1. Процесс выполняет системный вызов *open()* для того, чтобы открыть файл. При этом указывается полное (путевое) имя файла. Библиотечная функция *open()* посылает Администратору процессов QNX-сообщение определенной структуры, спрашивая что-то вроде: "Какой из администраторов ресурсов отвечает за этот файл?".
2. Администратор процессов сопоставляет путевое имя файла с деревом префиксов и возвращает функции *open()* сообщение-ответ, содержащее идентификатор администратора ресурсов.

3. Функция *open()* напрямую обращается к администратору ресурса с запросом на открытие файла.
4. Администратор ресурса создает у себя структуру данных, называемую *блоком управления открытым контекстом* (OCB, Open Control Block) и возвращает функции *open()* файловый дескриптор.

Теперь прикладной процесс может напрямую запрашивать у администратора ресурсов операции чтения/записи, используя полученный файловый дескриптор. Администратор ресурсов, в свою очередь, использует OCB для того, чтобы различать запросы разных процессов. OCB включает:

- файловый дескриптор, уникальный внутри одного процесса;
- идентификатор процесса, уникальный для узла сети;
- информацию о файле (например, информацию о текущем файловом указателе).

Новый OCB создается при каждом вызове функции *open()*.

Администратор процессов хранит пространство путевых имен в виде таблицы, где для каждого зарегистрированного префикса хранится 5 параметров:

- идентификатор узла сети, на котором выполняется администратор ресурса;
- номер процесса администратора ресурса;
- номер канала, через который администратор ресурса принимает клиентские запросы (подробнее каналы описаны в *разд. 3.3*);
- идентификатор префикса (используется, если администратор ресурсов зарегистрировал несколько префиксов);
- тип префикса (этот параметр во вводе/выводе не используется).

Префиксы в QNX Neutrino могут перегружаться, т. е. несколько администраторов ресурсов могут зарегистрировать одинаковые префиксы, которые Администратор процессов поместит в список. Какой же администратор ресурса получит наш запрос *open()* для обработки? Тот, который стоит первым в списке. Если этот администратор ресурса не сможет обработать запрос, т. е. вернет ошибку, то запрос будет передан следующему в списке администратору ресурса. И так далее, пока запрос не будет успешно обработан. Кто будет "перебирать" список? Это умеет делать клиентская функция *open()*. Она обращается к Администратору процессов один раз, сразу получает *весь* список

подходящих администраторов ресурсов и начинает их последовательно "обзванивать".

> ***Примечание***
> Положение своего префикса в списке администратор ресурса может регулировать флагом, имеющим одно значение из двух — "в начало списка" или "в конец списка". По умолчанию префикс помещается в начало списка.

В QNX 4.*xx* была замечательная, но абсолютно не-POSIX-овая утилита **`prefix`**, позволявшая задавать/уничтожать псевдонимы префиксов и просматривать дерево префиксов. Хорошо это или плохо, но в QNX Neutrino такую утилиту создавать не стали, однако ее функциональность вполне доступна:

- псевдонимы префиксов можно создавать с помощью утилиты **`ln`**, задав ей опции **`-Ps`**;
- удаление псевдонимов префиксов выполняется утилитой **`rm`**;
- дерево префиксов можно посмотреть, выполнив команду
  `ls /proc/mount`

Обратите внимание, что каталог `/proc/mount` нельзя увидеть в окне файлового менеджера (конечно, если не задать его явно в специальной строке в левой верхней части этого окна) или, например, с помощью команды **`ls /proc`**. Непосредственно в каталоге `/proc/mount` находятся несколько "файлов", имена которых состоят из пяти цифр, перечисленных через запятую. Это параметры префиксов, зарегистрированных с именем "`/`" (большинство из них принадлежит самому Администратору процессов). Что касается папок, имеющихся в этом "каталоге", то тут все просто — каталог `/proc/mount/dev/ser1` содержит "файл", имя которого состоит из параметров префикса `/dev/ser1`. Так можно определить параметры всех префиксов, зарегистрированных в системе.

3.2. Потоки POSIX в Neutrino

Современная операционная система немыслима без поддержки многопоточности. Потоки позволяют распараллелить выполнение независимых фрагментов программы по разным процессорам (конечно, если ЭВМ — многопроцессорная). С помощью потоков можно обеспечить более быструю реакцию графического приложения на пользовательский ввод. В операционной системе QNX4 тоже в каком-то

виде были реализованы потоки (вспомним функцию *tfork()*), но в Neutrino потоки соответствуют спецификации POSIX.

3.2.1. Управление потоками

Управление потоком заключается в создании потока, изменении его параметров в процессе выполнения и в завершении потока.

Помните, что процесс состоит минимум из одного потока? Этот первый (и, возможно, единственный поток) начинается с функции *main()* и включает все те функции, которые будут вызваны. Остальные потоки создаются явно с помощью функции *pthread_create()*. Рассмотрим параметры, которые требуются для управления потоком:

```
#include<pthead.h>
pthread_t tid;
pthread_attr_t attr;
void arg;
```

Здесь: `tid` — идентификатор потока *внутри* данного процесса; `attr` — атрибуты потока; `arg` — аргументы функции-точки входа в поток.

Для создания потока используется функция *pthread_create()*:

```
pthread_create (&tid, &attr, &func, &arg);
```

При этом обязательным параметром является только *`func`* — точка входа в поток, т. е. имя функции, с которой начнет выполнение поток, своего рода "потоковая функция *main()*". Например, если написать так:

```
pthread_create (NULL, NULL, &my_function, NULL);
```

то будет создан поток, который начнет выполняться с функции *my_function()*, вызванной без аргументов. Правда, мы не узнаем его идентификатор — ведь мы не задали указатель на `tid`, куда должен был бы записаться *идентификатор потока* (Thread ID, TID). Вообще, знать TID бывает полезно, например, чтобы уничтожить созданный поток:

```
pthread_cancel(tid);
```

Что касается структуры атрибутов `attr`, то она содержит ряд весьма полезных полей. Если вместо этой структуры при создании потока указать `NULL`, то поток будет создан с атрибутами по умолчанию (при этом параметры диспетчеризации будут унаследованы от родительского потока), но изменить мы их после создания потока не сможем. Поэтому даже если мы не собираемся сразу задавать потоку какие-то особые атрибуты, структуру атрибутов все-таки создать полезно:

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

После инициализации структуры можно читать/модифицировать ее поля, используя набор функций:

- *pthread_attr_get*()* — для того, чтобы узнать значение атрибута;
- *pthread_attr_set*()* — для того, чтобы задать значение атрибута.

Вместо звездочки (*) подставляйте значения, приведенные в первом столбце табл. 3.1.

Таблица 3.1. *Часто используемые атрибуты потока*

Поле структуры *attr*	**Назначение**	**Значения**	
		По умолчанию	**Альтернатива**
`detachstate`	Степень свободы потока	`PTHREAD_CREATE_JOINABLE`	`PTHREAD_CREATE_DETACHED`
`stacksize`	Размер стека создаваемого потока	4 Кбайт	Нужный размер
`exitfunc`	Функция, вызываемая при завершении потока, — деструктор потока	`NULL`	Указатель на функцию
`inheritsched`	Способ задания параметров диспетчеризации	`PTHREAD_INHERIT_SCHED` (Унаследовать от потока-родителя)	`PTHREAD_EXPLICIT_SCHED` (Задать явно)
`schedpolicy`	Дисциплина диспетчеризации	Наследуется от потока-родителя	`SCHED_FIFO` — FIFO; `SCHED_RR` — "карусельная"; `SCHED_OTHER` — спорадическая. (Использование возможно, только если установлен способ задания параметров `PTHREAD_EXPLICIT_SCH`

Поле структуры *attr*	Назначение	Значения	
		По умолчанию	Альтернатива
			ED)
schedparam	Значение параметров диспетчеризации, задаваемое через поля структуры типа sched_param (об этом см. ниже)	Наследуется от потока-родителя	Нужные значения структуры sched_param. (Использование возможно, только если установлен способ задания параметров PTHREAD_EXPLICIT_SCHED)
fags	Об этом поле поговорим позже		

Думаю, нужно прокомментировать эту таблицу (обратите внимание: в ней перечислены наиболее используемые по моему мнению атрибуты — остальные ищите в документации).

Атрибут `detachstate`, имеющий по умолчанию значение `PTHREAD_CREATE_JOINABLE`, указывает, что поток должен быть JOINABLE ("присоединяемый"). *JOINABLE-потоки* при завершении умеют возвращать указатель на заданную структуру данных с помощью функции *pthread_exit(value_ptr)*, где `value_ptr` — это указатель типа `void`. Какой-либо другой поток приложения может получить этот указатель, выполнив "присоединение" JOINABLE-потока:

```
pthread_join(tid, rval);
```

где `tid` — идентификатор "присоединяемого" потока, `rval` — указатель на указатель типа `void` (т. е. `void**`).

Как вы понимаете, "присоединяющий поток" может вызвать функцию *pthread_join()* либо до, либо после того, как "присоединяемый" поток вызвал *pthread_exit()*. В первом случае "присоединяющий" поток будет заблокирован до того момента, пока "присоединяемый" поток не изволит вызвать *pthread_exit()*. Такое заблокированное состояние называется *JOIN-блокированием*. Второй случай полностью аналогичен второй стадии завершения процесса — после вызова *pthread_exit()* ресурсы, занятые "присоединяемым" потоком, будут уничтожены, но

останется некоторая структура данных с возвращаемым указателем на структуру данных, который сохраняется, пока какой-либо поток не вызовет *pthread_join()*. А до того момента состояние "присоединяемого" потока будет называться *DEAD-блокированным* ("зомби").

> ***Примечание***
> При завершении родительского потока все его JOINABLE-потоки будут уничтожены, независимо от того, в каком они пребывают состоянии.

Если поток был создан как DETACHED с помощью флага `PTHREAD_CREATE_DETACHED`, то он, с одной стороны, не сможет возвращать данные (по крайней мере, столь откровенно), с другой стороны, он не будет уничтожен ядром при завершении его родителя. JOINABLE-поток ("присоединяемый") можно сделать DETACHED ("отсоединенным"), выполнив действие:

```
pthread_detach(tid);
```

но обратной дороги нет — DETACHED-поток не может стать JOINABLE-потоком.

> ***Примечание***
> Функцию *pthread_join()* можно использовать для синхронизации потоков, так как известно, что поток, вызвавший эту функцию, будет заблокирован в состоянии JOIN до завершения "присоединяемого" потока.

Вопрос: кстати, а как потоку "отсоединить" себя? Ответ: вызвать соответствующую функцию, указав ей свой TID. Вопрос: ну, а как узнать свой TID? Ответ: выполнить следующее:

```
pthread_t my_tid;
my_tid = pthread_self();
```

Кстати, как сравнить две переменные TID — это же не целые числа, а структуры? Используйте такие операторы:

```
int x;
x = pthread_equal(tid1, tid2);
```

Если значение `x` равно 0, то идентификаторы потоков разные.

Следующий атрибут потока, который мне хотелось бы обсудить, — `inheritsched`. По умолчанию он имеет значение `PTHREAD_INHERIT_SCHED`, что задает жесткое наследование параметров и дисциплины диспетчеризации создаваемого потока от его родителя. Если дисциплина диспетчеризации задается как конкретное значение, то параметры диспетчеризации задаются в виде структуры типа

sched_param. Эта структура состоит из поля sched_priority (приоритет потока), поля текущего приоритета потока (текущий приоритет может меняться — вспомните про клиент-управляемый приоритет) и из поля-структуры, игнорируемого при FIFO и "карусельной" дисциплинах диспетчеризации, но нужного для спорадической дисциплины диспетчеризации. К содержимому этой структуры можно обращаться по таким именам:

- sched_ss_low_priority — нижний приоритет (верхний приоритет — поле sched_priority);
- sched_ss_max_repl — максимальное число отложенных восстановлений;
- sched_ss_repl_period — период восстановления;
- sched_ss_init_budget — начальный бюджет.

Проиллюстрируем, как создать поток со спорадической дисциплиной диспетчеризации и со следующими значениями параметров диспетчеризации: верхний приоритет — 50, нижний — 30, число отложенных восстановлений — 12, начальный бюджет — 10 с, период восстановления — 5 с. Итак:

```
pthread_attr_t attr;
struct sched_param param;

pthread_attr_init(&attr);
pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
param.sched_priority = 50;
param.sched_ss_low_priority = 30;
param.sched_ss_max_repl = 12;
param.sched_ss_init_budget = { 5, 0 };
param.sched_ss_repl_period = { 10, 0 }
pthread_attr_setschedparam (&attr, &param);
pthread_attr_setschedpolicy (&attr, SCHED_OTHER);
pthread_create (NULL, &attr, my_func, NULL);
```

Теперь обсудим поле flags структуры атрибутов. Эти флаги — QNX-расширение атрибутов потоков, несколько выходящее за рамки POSIX.

Начнем с флага, определяющего порядок обработки сигнала завершения, полученного потоком. В соответствии с POSIX все потоки процесса должны быть уничтожены — именно это происходит в Neutrino по умолчанию. Однако мы можем задать значение флага

PTHREAD_MULTISIG_DISALLOW, и тогда будет уничтожен только поток, принявший сигнал:

```
attr.flags |= PTHREAD_MULTISIG_DISALLOW;
```

Или вернуться к значению по умолчанию (поведению, определенному POSIX):

```
attr.flags |= PTHREAD_MULTISIG_ALLOW;
```

Остальные флаги предусмотрены POSIX и позволяют изменять поведение потока при завершении, поэтому их называют еще *флагами управляемого завершения*.

Поясню, о чем идет речь. Поток в процессе работы может выполнять какую-либо операцию, во время которой его очень нежелательно уничтожать, т. е. *транзакцию*. Кроме того, поток мог, например, захватить семафор и если уничтожить этот поток, то семафор все равно останется занятым, т. к. поток должен его освободить явно. В этих случаях нужно задать для потока флаг запрета завершения:

```
attr.flags |= PTHREAD_CANCEL_DISABLE
```

Теперь, если попытаться уничтожить поток с помощью функции *pthread_cancel()*, ничего не получится — запрос на уничтожение встанет в очередь, пока мы не вернем значение флага по умолчанию:

```
attr.flags |= PTHREAD_CANCEL_ENABLE
```

Когда завершение разрешено, анализируется еще один флаг, определяющий тип завершения — *отстроченное* (по умолчанию) или *асинхронное*. При асинхронном завершении поток будет уничтожен на первой же операции, которую он попытается выполнить. Зададим этот тип завершения:

```
attr.flags |= PTHREAD_CANCEL_ASYNCHRONOUS
```

При отстроченном завершении (а это, напомню, — значение по умолчанию) поток будет выполняться, пока не достигнет ближайшей *точки завершения* (cancellation point). Точками завершения являются некоторые, как правило, блокирующие, вызовы ядра и, соответственно, высокоуровневые функции, которые базируются на таких вызовах. В Library Reference, в конце каждой статьи, посвященной описанию функции, есть небольшая табличка (табл. 3.2).

Таблица 3.2. *Допустимость использования функции*

Safety:	
Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Значение в правом столбце в строке Cancellation point (точка завершения) как раз и говорит, является данная функция такой точкой, или нет. Надо заметить, что у некоторых функций, являющихся точками завершения, имеются функции-"близнецы", которые не являются точками завершения, например, пара *MsgSend()–MsgSendnc()*, где постфикс *nc* означает "Non Cancellation point" (не точка завершения).

Остальные строки табл. 3.2 означают:

- Interrupt handler — допустимость использования функции в обработчиках прерываний;
- Signal handler — допустимость использования функции в обработчиках сигналов;
- Thread — допустимость использования функции в многопоточных приложениях.

Флаг типа завершения потока возвращается в значение по умолчанию (т. е. в значение "отложенное") так:

```
attr.flags |= PTHREAD_CANCEL_DEFERRED
```

Все это замечательно, но мы не обсудили ситуацию, когда поток захватил семафор и, тем не менее, должен быть уничтожен. Для таких случаев поток может зарегистрировать специальную функцию, которая будет вызываться, когда он должен срочно завершиться:

```
pthread_cleanup_push( &my_terminator, &his_args);
```

где `my_terminator` — имя функции-завершителя, `his_args` — ее аргументы (могут быть равны `NULL`).

Эта функция будет вызвана в трех случаях:

- поток вызвал *pthread_exit()*;
- поток уничтожается;
- поток вызвал функцию *pthread_cleanup_pop()* с аргументом, не равным 0 (обычно ее вызывают в последней строке потока — контрольный, так сказать, выстрел).

Примечание

Используйте пару *pthread_cleanup_push()* и *pthread_cleanup_pop()* в одном С-файле проекта, т. к. это вовсе не функции, а макросы, объявленные в файле `pthread.h`.

И последний штрих. При использовании многопроцессорной ЭВМ (разумеется, с модулем **`procnto-smp`** вместо **`procnto`**) количество порождаемых для решения задачи потоков можно привязать к количеству процессоров, которое хранится в системной странице:

```
#include <sys/syspage.h>
struct syspage_entry *_syspage_ptr
int cpu;
cpu = _syspage_ptr -> num_cpu;
```

Это позволит оптимизировать использование ресурсов ЭВМ.

3.2.2. Синхронизация потоков

В QNX Neutrino реализовано несколько способов синхронизации потоков. Причем только два из них (mutex и condvar) реализованы непосредственно в микроядре, остальные — надстройки над этими двумя:

- взаимоисключающая блокировка (**mut**ual **ex**clusion lock — mutex, "мутекс") — этот механизм обеспечивает исключительный доступ потоков к разделяемым данным. Только один поток в один момент времени может владеть мутексом. Если другие потоки попытаются захватить мутекс, они становятся мутекс-заблокированными.

  Примечание

  Если в состоянии мутекс-блокировки находится поток с приоритетом выше, чем у потока-владельца блокировки, то значение эффективного приоритета этого потока автоматически повышается до уровня высокоприоритетного заблокированного потока.

- условная переменная (**cond**ition **var**iable, или condvar) — предназначена для блокирования потока до тех пор, пока соблюдается некоторое условие. Это условие может быть сложным. Условные переменные всегда используются с мутекс-блокировкой для определения момента снятия мутекс-блокировки;
- барьер — устанавливает точку для нескольких взаимодействующих потоков, на которой они должны остановиться и дождаться "отставших" потоков. Как только все потоки из контролируемой

группы достигли барьера, они разблокируются и могут продолжить исполнение;

- ждущая блокировка — упрощенная форма совместного использования условной переменной с мутексом. В отличие от прямого применения mutex + condvar имеет некоторые ограничения;
- блокировка чтения/записи (rwlock) — простая и удобная в использовании "надстройка" над условными переменными и мутексами;
- семафор — про семафор мы уже говорили в *главе 2* — это, можно сказать, мутекс со счетчиком. Вернее, мутекс является семафором со счетчиком, равным единице.

Большинство этих механизмов работает только в пределах одного процесса, но это ограничение преодолевается путем использования механизма разделяемой памяти.

Кроме перечисленных способов синхронизацию можно осуществлять с помощью FIFO-диспетчеризации, "родных" QNX-сообщений и атомарных операций.

3.2.3. Механизмы IPC

В микроядре Neutrino реализована поддержка нескольких механизмов IPC:

- синхронные сообщения QNX — наряду с архитектурой микроядра являются фундаментальным принципом QNX. Это самый быстрый способ обмена данными произвольного размера в QNX;
- Pulses (по-русски этот тип сообщений называют "импульсами") — это фиксированные сообщения, имеющие размер 40 бит (8-битный код импульса и 32 бита данных), не блокирующие отправителя;
- сигналы POSIX (как простые, так и реального времени).

По просьбе заказчиков фирма QSS в порядке эксперимента ввела такое новшество, как *асинхронные сообщения*. Этот механизм, как и механизм импульсов, использует буфер для хранения сообщений. Функции асинхронных сообщений имеют такие же имена, как функции обычных сообщений, но с префиксом *asyncmsg_* (например, *asyncmsg_MsgSend()*).

Прошу не забывать, что помимо этих механизмов ОСРВ QNX поддерживает несколько дополнительных способов IPC (о них мы уже говорили в *главе 2*):

- очереди сообщений POSIX (реализованы в администраторе очередей **mqueue**);
- разделяемая память (реализована в Администраторе процессов);
- именованные каналы (реализованы в администраторе файловой системы QNX4);
- неименованные каналы (реализованы в администраторе каналов **pipe**).

Пожалуй, следует сказать пару слов о сигналах. *Сигналы* являются не блокирующим отправителя способом IPC. Сигналы имеют структуру, подобную импульсу (1 байт кода + 4 байта данных), и тоже могут ставиться в очередь. Для отправки процессу сигнала администратор может использовать две утилиты — стандартную UNIX-утилиту **kill** и более гибкую QNX-утилиту **slay**. По умолчанию обе эти утилиты посылают сигнал SIGINTR, при получении которого процесс обычно уничтожается. Строго говоря, процесс может определить три варианта поведения при получении сигнала:

- *игнорировать сигнал* — для этого следует задать соответствующее значение такому атрибуту потока, как сигнальная маска (поэтому обычно говорят не "игнорировать", а "маскировать" сигнал). Надо сказать, что три сигнала не могут быть проигнорированы: SIGSTOP, SIGCONT и SIGTERM;
- *использовать обработчик по умолчанию* — т. е. ничего не предпринимать. Обычно обработчиком по умолчанию для сигналов является уничтожение процесса;
- *зарегистрировать собственный обработчик* — т. е. собственную функцию, которая будет вызвана при получении сигнала.

Поток может вызвать функцию ожидания того или иного сигнала (сигналов). При этом поток станет SIGNAL-блокированным.

Кстати, об утилите **slay**. Она имеет чрезвычайно полезную опцию **-P**. Для чего — не скажу, загляните в электронную документацию, поставляемую в составе дистрибутива QNX Momentics ☺.

Спецификация POSIX определяет порядок обработки сигналов только для процессов. Поэтому при работе с многопоточными процессами в QNX необходимо учитывать следующие правила.

- Действие сигнала распространяется на весь процесс.
- Сигнальная маска распространяется только на поток.

- Неигнорируемый сигнал, предназначенный для какого-либо потока, доставляется только этому потоку.
- Неигнорируемый сигнал, предназначенный для процесса, передается первому не SIGNAL-блокированному потоку.

Сигналы и сообщения типа "импульс" позволяют реализовать в QNX событийно-управляемую модель поведения системы. Под *событиями* понимаются QNX-импульсы, прерывания, сигналы и различные типы неблокирущих сообщений. Впрочем, об этом мы поговорим позже.

3.3. Синхронный механизм IPC, реализованный в микроядре

В этом разделе речь пойдет о базовом механизме синхронного обмена сообщениями. Эти сообщения часто называют "родными" сообщениями QNX из-за того, что данный механизм является отличительным свойством, присущим всем операционным системам семейства QNX. По сути, механизм обмена сообщениями реализует ту самую программную "шину", которая совместно с механизмом пространства имен путей связывает набор разношерстных программ и библиотек в нечто единое, называемое операционной системой QNX Neutrino.

3.3.1 Концепция механизма сообщений

В синхронном обмене сообщениями QNX участвуют две стороны — клиент и сервер. Поскольку рассматриваемый механизм реализован непосредственно в микроядре, в качестве клиентов и серверов выступают потоки. То есть, нет принципиальной разницы, между какими именно потоками происходит обмен QNX-сообщениями — внутри процесса, между разными процессами или даже между процессами, работающими на разных узлах локальной вычислительной сети.

Идея заключается в следующем.

1. Поток-сервер выполняет вызов ядра по приему сообщений от клиентов, указав адрес и размер буфера, предназначенного для записи туда сообщения (сервер при этом блокируется по приему — становится Receive-блокированным).
2. Поток-клиент посылает сообщение серверу — вернее, сообщает микроядру посредством вызова адрес и размер буфера, содержимое которого следует скопировать в приемный буфер сервера, а также

адрес и размер буфера, в который следует поместить ответ сервера. При этом клиент блокируется по ответу (становится Reply-блокированным). В этот же момент сервер разблокируется.

3. Поток-сервер обрабатывает сообщение клиента и посылает сообщение-ответ, т. е. указывает ядру адрес и размер буфера, который следует скопировать в буфер, приготовленный клиентом для получения ответа. При этом клиент разблокируется, сервер блокироваться уже не будет.

Назовем этот алгоритм "Вариант 1" и изобразим на картинке (рис. 3.1).

Рис. 3.1. Передача QNX-сообщения. Вариант 1

Очевидно, что может возникнуть ситуация, когда клиент попытается отправить сообщение раньше, чем сервер заблокируется по приему. Это и есть "Вариант 2". Ничего страшного не произойдет: клиент заблокируется по передаче (Send-блокирован), сервер, как только выполнит вызов приема, сразу, не блокируясь, получит сообщение, а клиент станет Reply-блокированным. Далее без изменений.

> ***Примечание***
> Вместо сообщения-ответа сервер может разблокировать Reply-блокированного клиента, послав некоторое число-статус (его еще называют "кодом ошибки"). На самом деле, статус сервер возвращает всегда — это то самое число, которое возвращает клиентская функция отправки сообщения после того, как клиент разблокируется.

Теперь немного усложним схему. А перед этим вспомним, как организовано классическое клиент-серверное взаимодействие.

> ***Примечание***
> Напомню, что под понятиями *клиент* и *сервер* всегда имеют в виду *программы*, ЭВМ называют такими же терминами в зависимости от того, какие программы на них преимущественно выполняются.

Для примера рассмотрим (в упрощенном виде), как клиент сети TCP/IP находит нужный сервер. Программа-сервер указывает своей операционной системе, порт с каким номером она (программа) желает "слушать". Если запрошенный порт свободен, то ОС позволяет программе его занять. Клиент для установления соединения указывает ЭВМ (имя хоста или непосредственно IP-адрес) и номер порта (клиентские программы обычно имеют некоторый номер порта,

заданный по умолчанию, у веб-обозревателя — 80). Операционная система серверной ЭВМ, получив сообщение из сети на какой-то порт, знает, какая программа "слушает" этот порт, и передает запрос ей на обработку. Таким образом, для организации клиент-серверного взаимодействия в сети TCP/IP клиент должен знать адрес серверной ЭВМ и номер порта, который "слушает" программа-сервер (кстати, те же параметры использует сервер для доступа к клиенту).

Теперь рассмотрим более близкую схему — обмен "родными" сообщениями в ОСРВ QNX 4.*xx*. Субъектами обмена сообщениями в ней были не потоки, а процессы, поэтому для отправки сообщения клиент должен был указать серверную ЭВМ (NID — логический идентификатор узла) и PID процесса-сервера. Таким образом, для организации клиент-серверного взаимодействия в QNX 4.*xx* клиент должен был знать идентификатор NID серверной ЭВМ и PID программы-сервера.

Казалось бы, что коль скоро обмен сообщениями в QNX Neutrino выполняется между потоками, то можно было бы просто добавить еще один параметр — TID серверного потока. Однако используется несколько иной подход.

Не случайно мы говорили о клиент-серверном взаимодействии в сети TCP/IP. Дело в том, что в TCP/IP используется один прием, позволяющий повысить скорость обработки клиентских запросов. Для простоты рассмотрим, как этот прием используется в классической серверной программе — веб-сервере Apache. Этот сервер при запуске по умолчанию порождает 5 своих копий (это значение можно изменять произвольно). Для чего? Как только какой-либо клиент (веб-обозреватель) запросит подключение к серверу, один из процессов Apache тут же приступит к обслуживанию этого клиента. То же самое произойдет со следующим клиентским запросом. Другими словами, для того, чтобы приступить к обслуживанию некоторого количества клиентов, не требуется выполнять ресурсоемкую операцию порождения процесса — какой-то пул процессов уже ждет "своих" клиентов. Разумеется, ЭВМ не резиновая, поэтому максимальное число процессов, как выполняющих запросы, так и ожидающих таковые, ограничено (максимум ожидающих процессов Apache по умолчанию задается равным 10).

Отсюда интересное следствие: по сути дела, клиентскую программу не должно волновать, какой процесс обработает ее запрос. Ей интересно, чтобы запрос был обработан правильно и вовремя. Выходит, что клиент

в сети TCP/IP задает посредством номера порта не серверный процесс, а любой процесс, предоставляющий нужный сервис.

В QNX Neutrino реализовано такое полезное средство, как *пул потоков* (Thread pool). Назначение механизма пулов потоков аналогично назначению пула серверов того же Apache, поэтому в организации клиент-серверного взаимодействия QNX мы наблюдаем картину, аналогичную клиент-серверному взаимодействию TCP/IP: потоку-клиенту все равно, какой поток-сервер из пула Receive-блокированных потоков-близнецов получит сообщение-запрос и вернет сообщение-ответ, лишь бы сервер выполнил то, что клиенту требуется. При этом роль TCP-сокета выполняет *канал* (Channel). Номер канала, по сути дела, идентифицирует пул потоков.

Таким образом, для передачи сообщения клиенту нужно знать:

- узел сети, на котором выполняется поток-сервер (задается *дескриптором узла*);
- PID-процесса, содержащего поток-сервер;
- номер канала (Channel ID, CHID), из которого ожидает сообщения пул серверных потоков.

Удобства ради (в чем это удобство заключается, мы еще поговорим) клиент, прежде чем посылать сообщения, устанавливает соединение с нужным каналом (получив при этом идентификатор соединения COID — Connection ID), а затем посылает сообщения через соединение. Разумеется, сначала сервер должен создать такой канал.

Теперь посмотрим на туже самую картинку, но уже с каналом. А чтобы никто не мог сказать, что мы рассмотрели только один вариант передачи сообщений, рассмотрим "Вариант 2", т. е. случай, когда клиент отправил сообщение *до* того, как сервер пожелал таковое получить.

Рис. 3.2. Передача QNX-сообщения через соединение. Вариант 2.

Прокомментируем рис. 3.2 с допущением, что сервер работает на той же ЭВМ, что и клиент, т. е. примем дескриптор узла (ND, Node Descriptor) равным нулю:

1. Поток-сервер, исполняющийся в рамках процесса, создал канал для приема сообщений.

2. Поток-клиент установил соединение с каналом, указав параметры: ND=0, PID серверного процесса, CHID.
3. Поток-клиент посылает сообщение серверу через канал COID=coid1. При этом клиент блокируется по запросу (становится Send-блокированным).
4. Поток-сервер выполняет вызов ядра по приему сообщений из канала. При этом клиент переходит в состояние блокированного по ответу(становится Reply-блокированным). Сервер не блокируется.
5. Поток-сервер обрабатывает сообщение клиента и посылает сообщение-ответ. При этом клиент разблокируется.

> ***Примечание***
> Внимательный читатель заметил конечно же, что клиент для установления соединения должен каким-то образом узнать CHID. Если сервер и клиент являются потоками одного процесса то нет проблем — CHID может быть глобальной переменной процесса. Но что если потоки принадлежат разным процессам? А если они еще и выполняются на разных узлах сети, то как определить ND? Об этом мы поговорим в *разд. 3.3.2*.

Пожалуй, самое время рассмотреть пример кода, демонстрирующего обмен сообщениями. Вот текст файла `msg_simple.c`, в котором проиллюстрирован обмен сообщениями между клиентом и сервером, выполняющимися в рамках одного процесса:

```
#include<pthread.h>
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include<sys/neutrino.h>

//------- Здесь находится определение функций-------
//----- void *server(void) и void *client(void)-----

int main()
{
      // Объявляем структуры для хранения идентификаторов
      // потоков сервера и клиента
   pthread_t  server_tid, client_tid;
      // Запускаем потоки сервера и клиента
   pthread_create(&server_tid, NULL, server, NULL);
```

```
   pthread_create(&client_tid, NULL, client, NULL);
      // Подождем с завершением процесса, пока работают потоки
   pthread_join(client_tid, NULL);
      // Завершаем все JOINABLE-потоки процесса
   return EXIT_SUCCESS;
}
```

Рассмотрим код сервера:

```
void  *server()
{
/* Объявляем переменные для хранения идентификаторов */
   int chid, rcvid;
/* Создаем приемный буфер и буфер для ответа */
   char receive_buf[25], reply_buf[25];
/* Начинаем выполнять I этап - создадим канал. Поскольку это
   первый вызов создания каналов и опции не заданы,
   номер канала будет 1. Какие опции можно задавать при
   создании канала — смотрите в документации. Скажу лишь, что
   опции бывают достаточно интересными */
   chid = ChannelCreate(0);
/* Немного "поспим" для того, чтобы у нас все-таки получился
   "Вариант 2". Пока сервер отдыхает, клиент должен успеть
   выполнить шаги 2 (создание канала) и 3 (отправить
   сообщение) */
   sleep(1);
/* Принимаем сообщение из канала chid, получив при этом его
   идентификатор rcvid — это позволяет легко и просто посылать
   ответ. Сообщение, разумеется, следует искать в буфере
   receive_buf, имеющем размер sizeof(receive_buf). Последний
   аргумент NULL означает, что нам не интересны параметры
   сообщения. В принципе, мы можем в любой момент узнать их с
   помощью функции MsgInfo(), указав ей rcvid и адрес
   соответствующей структуры для записи этих самых параметров.
   Что собой представляют параметры сообщения — читайте
   в документации. */
   rcvid = MsgReceive(chid, &receive_buf,
                      sizeof(receive_buf), NULL);
/* Напечатаем текст принятого сообщения */
   printf(" Server thread: message <%s>
               has recived.\n", &receive_buf);
```

```
/* Запишем в буфер ответа сообщение для клиента */
   strcpy(reply_buf, "Our strong answer");
/* Отправим ответ, указав в качестве статуса произвольное
   число — 15052002. Это число мы могли бы послать клиенту
   и без самого сообщения с помощью функции MsgError(). */
   MsgReply(rcvid, 15052002, &reply_buf,
            sizeof(reply_buf));
/* Уничтожаем канал и завершаем поток-сервер. */
   ChannelDestroy(chid);
   pthread_exit(NULL);
}
```

Теперь рассмотрим код потока-клиента:

```
void  *client()
{
/* Объявляем переменные для хранения идентификаторов */
   int coid, status;
   pid_t PID;
/* Создаем буфер для сообщения и буфер для ответа от сервера*/
   char send_buf[25], reply_buf[25];
/* Определяем PID процесса, в котором содержится сервер.
   В нашем случае PID серверного процесса равен
   PID клиентского, поэтому можно поступить так: */
   PID=getpid();
/* Шаг 2 по рис. 3.2: устанавливаем соединение, указав
   параметры: 0 — ND (подробнее о ND смотрите в разд. 5.2.1),
   PID, 1 — номер канала (Мы знали! Мы знали!), 0 — так
   называемый индекс (дело в том, что по умолчанию
   идентификатор соединения берется из того же множества,
   что и файловые дескрипторы, а если задать индекс
   _NTO_SIDE_CHANNEL, то значение COID будет больше,
   чем любой возможный файловый дескриптор), 0 — флаги
   (подробности ищите в штатной документации) */
   coid = ConnectAttach(0, PID, 1 , 0, 0);
/* Запишем в буфер текст сообщения для сервера */
   strcpy(send_buf, "It's very simple example");
/* Шаг 3 по рисунку 3.2: отправляем сообщение и принимаем
   ответ со статусом. */
   status = MsgSend(coid, &send_buf, sizeof(send_buf),
                    &reply_buf, sizeof(reply_buf));
```

```
/* Закрываем соединение и завершаем поток-клиент. */
   ConnectDetach(coid);
   pthread_exit(NULL);
}
```

Вот и все — мы познакомились с главным механизмом QNX Neutrino, на котором основана эта операционная система. Что бы мы ни делали — выводили картинки на экран, записывали данные в файлы — на нижнем уровне шлются эти самые сообщения: от графического приложения через графическое ядро видеодрайверу, от текстового редактора драйверу контроллера диска и т. д.

Каковы достоинства и недостатки такой архитектуры? Достоинства — потрясающие гибкость, масштабируемость и прозрачность системы и сетей Qnet, простота отладки сложнейших системных сервисов, включая драйверы и поддержку протоколов. Плата за все эти "вкусности" (а за все приходится платить!) — дублирование данных и большое количество переключений контекста.

Теперь рассмотрим некоторые нужные "навороты" — ведь не будем же мы слать сообщения только в пределах одного процесса!

3.3.2. Установление соединения с каналом: механизм префиксов

Послать сообщение по установленному соединению — дело нехитрое, сложнее узнать PID и CHID. Из всех способов удобнее всего два, основанные на механизме пространства путевых имен Администратора процессов.

Первый способ, настоятельно рекомендуемый компанией QNX Software Systems, — оформить, так сказать, сервер в виде администратора ресурсов.

Второй способ широко использовался программистами QNX4 и понятно, почему: он чрезвычайно прост, а значит — надежен и эффективен.

У администратора ресурсов, тем не менее, есть два жирных достоинства:

- код клиента можно написать так, чтобы он был на 100% POSIX-совместимым;

- с администратором ресурсов могут взаимодействовать клиентские программы, выполняющиеся в других ОС, лишь бы они каким-то образом получили доступ к файловой системе на стороне сервера.

Концепции администраторов ресурсов — не в плане программирования, а вообще — посвящен *разд. 4.1*. Мы же теперь рассмотрим самый простой, QNX4-подобный способ.

Идея заключается в том, что сервер регистрирует у Администратора процессов некоторое имя, называемое по историческим причинам *префиксом*, при этом создается и сам канал. Администратор процессов сопровождает таблицу, где хранит соответствие между именем префикса и параметрами, нужными для установления соединения с этим каналом.

В качестве иллюстрации рассмотрим две маленькие программки, сделанные из `msg_sample.c`, — `msg_srv.c` и `msg_client.c`. Обе эти программки вместе делают то же самое и в той же последовательности, что и поток-сервер и поток-клиент предыдущего примера. Для компиляции и запуска примера нужно выполнить команды:

```
make msg_prefix
msg_srv &
msg_client
```

Сначала рассмотрим код сервера (файл `msg_srv.c`):

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

int main()
{
   name_attach_t      *my_prefix;
   dispatch_t         *dpp;
   int                rcvid;
   char            receive_buf[25], reply_buf[25];

/* Создадим контекст диспетчеризации (речь идет
   о диспетчеризации входящих сообщений). Нам не требуется
   знать ничего об этой структуре кроме того, что она
   содержит номер создаваемого канала. Учтите, что сам
   канал сейчас не создается — он будет создан функцией
   регистрации префикса. */
```

```
   dpp = dispatch_create();
/* Зарегистрируем префикс. 0 означает, что префикс локальный
   для данной ЭВМ; он может быть глобальным, т. е. сетевым,
   но об этом читайте в разд. 5.2.2.
   При регистрации префикса создается канал, номер которого
   помещается в структуру диспетчеризации. */
   my_prefix = name_attach( dpp, "my_prefix", 0);
/* Поспим 10 секунд. Это требуется опять же для
   воспроизведения "Варианта 2". За это время, я надеюсь,
   вы успеете запустить клиентскую программу. Если
   не успеете — то получите "Вариант 1". */
   sleep(10);
/* Далее выполняем обычную работу, с той лишь разницей,
   что CHID извлекается из структуры my_prefix,
   возвращаемой функцией регистрации префикса. */
   rcvid = MsgReceive(my_prefix->chid, &receive_buf,
                      sizeof(receive_buf), NULL);
   strcpy(reply_buf, "Our strong answer");
   MsgReply(rcvid, 15052002, &reply_buf,
            sizeof(reply_buf));
/* Уничтожаем префикс и канал. */
   name_detach( my_prefix, 0);
/* Завершаем поток и процесс. Проще было бы выполнить
   "return EXIT_SUCCESS;" но я хотел еще раз подчеркнуть,
   что обмен происходит не между процессами, а между
   потоками. */
   pthread_exit(NULL);
}
```

Теперь рассмотрим код клиента — файл `msg_client.c` (то, что печатается на экране, опустим — к сути дела это не относится):

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>
#include <stdlib.h>
int main()
{
/* Создаем необходимые переменные и буферы. Обратите внимание,
   что находить PID уже не требуется */
      int coid, status;
      char send_buf[25], reply_buf[25];
```

```
/* Создаем канад. Для этого нам нужно знать только имя
   префикса и его тип (0 — локальный). */
      coid = name_open("my_prefix", 0);
/* "По-старому" готовим сообщение и посылаем его через
   соединение. */
      strcpy(send_buf, "It's very simple example");
      status = MsgSend(coid, &send_buf, sizeof(send_buf),
                       &reply_buf, sizeof(reply_buf));
/* Закрываем соединение и завершаем работу. */
      name_close(coid);
      return EXIT_SUCCESS
}
```

Вот так — дешево и сердито.

3.3.3. Пулы потоков

Итак, вспомним, что нам уже известно о пулах потоков:

- они состоят из нескольких серверных потоков-близнецов;
- они слушают один и тот же канал;
- они нужны для быстрого реагирования на запросы клиентов.

Потоки в пуле могут быть либо в состоянии ожидания либо в состоянии обработки. То есть поток в пуле либо ждет запрос клиента, либо обрабатывает такой запрос. Исходя из этого у пула есть атрибуты:

- *нижняя ватерлиния* — минимальное число ожидающих потоков. Если количество потоков в пуле меньше нижней ватерлинии, то система создаст дополнительно инкремент потоков;
- *инкремент* — число потоков, создаваемых, когда число ожидающих потоков стало меньше, чем нижняя ватерлиния;
- *верхняя ватерлиния* — максимальное число ожидающих потоков. Лишние потоки немедленно уничтожаются;
- *максимум* — максимальное число и ожидающих, и обрабатывающих потоков. Например, если число *обрабатывающих* потоков равно максимуму, то *ожидающих* не будет вообще.

Чтобы создать пул, необходимо сначала заполнить структуру, имеющую тип `thread_pool_attr_t`. Помимо уже рассмотренных атрибутов пула эта структура содержит ряд других полей, смысл

которых будет понятен тому, кто изучит вопросы создания администраторов ресурсов.

> ***Примечание***
> Разработка администраторов ресурсов не рассматривается в этой книжке, поскольку желающие всегда могут изучить эту тему как по книгам Роба Кртена (Rob Krten), так и по электронной документации, входящей в состав комплекта разработчика QNX Momentics.

3.3.4. Векторные сообщения

На практике часто требуется передать за одну транзакцию не просто содержимое буфера, а структуру, состоящую из разнообразных данных. Конечно, можно создать один большой буфер, скопировать в него все нужные данные и передать его в виде сообщения. Однако такой подход требует как дублирования данных, так и лишних операций копирования. Для упрощения жизни программистов в QNX Neutrino есть механизм *векторных сообщений*.

При передаче векторного сообщения вместо указателя на буфер и его размера задается указатель на массив структур IOV и число элементов в этом массиве. *Структура IOV* (Input-Output Vector) содержит два поля — адрес буфера и размер буфера. Таким образом, для передачи трех переменных в одном сообщении следует создать массив из трех структур IOV, а полям этих структур присвоить соответствующие адреса переменных и их размеры. Посмотрим, как это можно сделать, используя специальный макрос *SETIOV()*:

```
/* Задаем переменные */
char a[10];
int b;
double c;
/* Задаем вектор из трех элементов — переменных-то три. */
iov_t iovs[3];
/* Заполним переменные какой-либо информацией, которую нужно
   передать серверу за одну транзакцию */
...
/* Проинициализируем вектор, т. е. соответствующим полям
   присвоим соответствующие значения */
SETIOV(&iovs[0],      a, sizeof(a));
SETIOV(&iovs[1],      &b, sizeof(b));
SETIOV(&iovs[2],      &c, sizeof(c));
```

Теперь векторное сообщение готово к отправке. Для этой операции можно воспользоваться двумя функциями:

- *MsgSendv()* — в случае, когда и запрос и ответ являются векторными сообщениями;
- *MsgSendvs()* — в случае, когда вектор содержится только в запросе, а в ответе — одиночное сообщение.

Кроме того, есть функция *MsgSendsv()* — для случая, когда посылается одиночное сообщение, а в ответ приходит вектор.

Соответственно, сервер может создать для приема сообщений свой вектор, который сможет принять с помощью функции *MsgReceivev()*. Вектор-ответ посылается, как вы догадались, с помощью функции *MsgReplyv()*.

Итак, вот как выглядит отправка векторного сообщения:

```
MsgSendvs(coid, iovs, 3, &reply_buf, sizeof(reply_buf));
```

...и получение его сервером:

```
MsgRecievev(chid, server_iovs, 3, NULL);
```

Конечно, на стороне сервера надо заготовить переменные, в которые следует "расфасовать" содержимое сообщения, создать массив структур `iov_t` (мы назвали его `server_iovs`) и проинициализировать этот массив адресами и размерами переменных.

Хочу обратить ваше внимание на то, что микроядру глубоко безразлично, что вы посылаете, — одиночное сообщение или векторное. Для микроядра сообщение — бессмысленный набор из определенного количества байтов, которые нужно передать от одного потока другому. То есть ответственность за интерпретацию структуры сообщения целиком ложится на программиста. Отсюда следствие: вы можете отправить одну структуру, а на стороне сервера рассматривать сообщение как совершенно другую структуру, но такого же суммарного размера.

3.3.5. Многошаговый обмен сообщениями

Вы помните, что механизм сообщений — основа всего, что делается в QNX. Например, если приложение записывает в файл порцию данных, то на самом деле оно попросту шлет сообщение *определенного формата* серверу, функцию которого выполняют средства файловой системы *(см. главу 4)*. Что значит — сообщение "определенного формата"? А то, что компоненты операционной системы QNX Neutrino

знают, как интерпретировать содержимое некоторых сообщений. Для этого в сообщениях, посылаемых компонентами операционной системы, имеется заголовок длиной 12 байт, который содержит сведения о том, что находится внутри этого сообщения. То есть компоненты операционной системы должны уметь сначала прочитать заголовок, а затем, в зависимости от полученной из заголовка информации, принять решение о том, как поступить с остальной частью сообщения (может, вообще стоит ее проигнорировать). Значит, сервер должен уметь читать сообщения в несколько шагов. Как это делается?

В QNX Neutrino реализовано несколько дополнительных функций серверных функций:

- *MsgRead()* и *MsgReadv()*;
- *MsgWrite()* и *MsgWritev()*.

В дальнейшем мы не будем упоминать суффикс *v*, но не забудем, что все функции обмена сообщениями имеют векторные версии. Упомянутые функции имеют дополнительный аргумент `offset` (смещение). Используются они так:

1. Сначала сервер принимает сообщение с помощью *MsgRecieve()*, при этом в качестве приемного буфера указываем только структуру заголовка (средства операционной системы заказывают для заголовка сообщения, как это можно догадаться, буфер размером 12 байт).
2. Клиент переходит в Reply-блокированное состояние, а мы тем временем анализируем, что за сообщение нами получено. Из структуры *struct _msg_info * info* (мы ее получаем как последний аргумент функции *MsgRecieve()* или через вызов функции *MsgInfo()*) нам известен полный размер переданного сообщения (включая заголовок), из заголовка нам известна логическая структура сообщения.
3. Сервер спокойно начинает считывать содержимое сообщения (читайте — передающего буфера клиента) с помощью функции *MsgRead()*. В качестве аргументов функции *MsgRead()* указывают идентификатор полученного сообщения (возвращается функцией *MsgRecieve()*), адрес и размер буфера, в который следует записать полученные данные, и смещение, с которого надо начинать считывать данные из буфера клиента. Клиент остается Reply-блокированным!

4. Сервер подготавливает ответное сообщение. Он может его отправить сразу, вызвав *MsgReply()* или *MsgError()*, или записать в Reply-буфер клиента порции данных, используя *MsgWrite()*. Функция *MsgWrite()* позволяет записывать любые порции данных в любую часть Reply-буфера клиента. Но, записав данные, сервер должен для разблокирования клиента обязательно вызвать *MsgReply()* или *MsgError()*.

Как видите, механизм передачи сообщений QNX Neutrino — чрезвычайно гибкий инструмент.

3.3.6. Анализ конфигурации каналов и соединений

Для того чтобы увидеть конфигурацию каналов и установленных с ними соединений, удобно использовать QNX Momentics IDE.

> ***Примечание***
> Если инструментальная и целевая системы — разные ЭВМ, то на целевой системе должен быть запущен целевой агент (QNX Target Agent) — программа **qconn**. Эта программа предоставляет для IDE необходимую информацию с целевой системы.

Запускаем IDE, открываем окно перспективы QNX System Information (**Window > Open Perspective > QNX System Information**). Затем создаем target-проект[1] (**File > New > Project**). Запустится мастер создания проекта. В левой части окна мастера щелкаем на элементе **QNX** (этот проект специфичен для QNX), в правой — на наименовании типа проекта **QNX Target System Project** (рис. 3.3).

Рис. 3.3. Запуск мастера для создания target-проекта

Нажимаем кнопку **Next** и задаем в предложенной мастером форме (рис. 3.4) имя проекта и параметры целевой системы (имя проекта — target_1, IP-адрес целевой системы — 192.168.0.187, номер порта целевого агента QNX Target Agent — 8000):

---

[1] Target-проект — это особый тип проекта, создаваемый в QNX IDE для хранения параметров целевой системы. Такими параметрами являются, например, IP-адрес целевой системы и номер порта, который "слушает" запущенная на целевой системе программа **qconn**.

Рис. 3.4. Задание параметров target-проекта

Нажимаем кнопку **Finish**. IDE создаст target-проект с именем target_1. Вернемся к окну перспективы QNX System Information. В левой верхней части окна находится представление **Target Navigator**, в котором показан список открытых проектов QNX Target System Project. В нашем случае там будет единственный проект с именем target_1 (рядом в скобках указано имя узла целевой системы QNX Neutrino). Нажимаем на значок + возле имени проекта, чтобы развернуть список процессов (рис. 3.5).

Рис. 3.5. Представление **Target Navigator** с открытым списком процессов

В списке указаны имена процессов, в скобках рядом с ними — идентификаторы процессов (PID). В правой части заголовка представления есть треугольная стрелка, щелкнув на которой можно открыть меню с настройками параметров представления.

В правой части окна перспективы расположены несколько представлений, сгруппированных на вкладках, ярлычки которых по умолчанию помещены внизу. На экране можно увидеть представления **System Summary**, **Process Information**, **Thread Information**, **Memory Information**, **Malloc Information**.

Представление **System Summary** разделено на две части (рис. 3.6).

Рис. 3.6. Представление **System Summary**

В верхней части представления отображается общая информация о целевой системе. Нижняя часть содержит три вкладки: **All Processes** (информация обо всех процессах), **Server Processes** (те процессы из указанных на вкладке **All Processes**, у которых есть потоки, создавшие каналы для приема сообщений или импульсов), **Application Processes** (те процессы из указанных на вкладке **All Processes**, которые не вошли в категорию **Server Processes**).

Представление **Process Information** является надмножеством представления **Thread Information**. Для того чтобы в нем что-либо отобразилось, надо выделить какой-нибудь процесс в представлении

Target Navigator. Для примера выделим администратор неименованных программных каналов **pipe** (рис. 3.7).

Рис. 3.7. Представление Process Information

Обратите внимание, что для каждого потока показаны: значение приоритета (столбец **Priority Name**) с дисциплиной диспетчеризации (**10o** означает приоритет 10, дисциплина other, т. е. спорадическая диспетчеризация, диспетчеризация FIFO отображалась бы как **f**, "карусельная" диспетчеризация — как **r**), состояние потока (столбец **State**) и, для Receive-блокированного потока, номер канала, который поток "слушает" (столбец **Bocked on**).

Представление **Memory Information** позволяет анализировать память, занимаемую процессом. Обратите внимание на элемент **Liblary** (я его специально развернул) — с его помощью мы можем увидеть, какие разделяемые библиотеки использует данное приложение (рис. 3.8).

Рис. 3.8. Представление **Memory Information**

Представление **Malloc Information** позволяет анализировать, как процессу выделялась оперативная память (рис. 3.9).

Рис. 3.9. Представление **Memory Information**

Это представление показывает, сколько раз запрашивалось выделение, освобождение и перевыделение памяти. Подробная информация представлена в следующих полях таблицы **Distribution**.

- **Byte Range** — значение в этом поле показывает, какого примерно размера фрагмент ОЗУ запрошен процессом (точный размер из этого представления узнать нельзя — это привело бы к значительному увеличению потока информации от **qconn** и "захламило" бы представление).
- **Total mallocs** — значение в этом поле показывает, сколько раз запрашивался фрагмент данного диапазона размеров **Byte Range**.
- **Total frees** — значение в этом поле показывает, сколько этих фрагментов освобождено процессом.

- **Allocated** — значение в этом поле равно разности значений полей **Total mallocs** и **Total frees**.
- **% Returned** — значение в этом поле определяется по формуле **Total frees** × 100 / **Total mallocs** (с округлением до десятков процентов).
- **Usage (min/max)** — диапазон, в пределах которого находится количество используемых в настоящее время фрагментов данного размера **Byte Range** — от <нижняя граница **Byte Range**> × **Allocated** до <верхняя граница **Byte Range**> × **Allocated**.

Кроме описанных представлений, открывающихся по умолчанию при создании target-проекта, доступно еще несколько. Для того, чтобы их открыть, выберите элемент меню **Window > Show View**. Откроется окно, в котором можно выбрать нужное представление, которое требуется дополнительно разместить в окне текущей перспективы. Представления для удобства поиска сгруппированы в папки. На рис. 3.10 показано такое окно с открытыми папками, которые содержат представления, используемые для работы с целевыми системами.

Рис. 3.10. Открытие дополнительных представлений

Выберем, например, представление **System Blocking Graph** (рис. 3.11).

Рис. 3.11. Окно представления **System Blocking Graph**

Это представление содержит граф и таблицу. В таблице отображена информация по всем блокировкам анализируемой целевой системы, смысл ее полей очевиден для тех, кто знаком с механизмом сообщений QNX. Обратите внимание: в верхней части рис. 3.11 хорошо виден пул потоков, Receive-блокированных на канале. Случай, когда с каналом установлено соединение со стороны клиента, проиллюстрирован на рис. 3.12.

Рис. 3.12. Отображение установленного соединения

Можно увидеть, как поток 1 (**Thread 1**) процесса с PID=7 (это **`devc-con`** — драйвер консоли) блокирован по приему сообщения из канала 1 (**Channel 1**), а три потока, принадлежащих разным процессам **`login`**

(с идентификаторами 176145, 176146, 454675 — каждый из потоков-клиентов в своем процессе имеет идентификатор TID=1), установили соединение с этим каналом. Для того чтобы подробнее узнать о данном соединении, надо открыть представление **Connection Information**. Для большей наглядности вместо соединений процесса **`login`** рассмотрим таблицу соединений процесса **`io-net`** (рис. 3.13).

Рис. 3.13. Представление **Connection Information** — информация о соединениях

В левой части таблицы соединений процесса находится столбец **File Descriptors**. Мы уже говорили: что бы ни делал пользователь, в QNX Neutrino на самом деле работает механизм сообщений. Поэтому для операционной системы открытие процессом файла означает, что поток процесса установил соединение с каналом администратора ресурса (драйвер — это частный случай администратора ресурса). Если соединение устанавливается с помощью обычной POSIX-функции *open()*, то клиент получает некоторый файловый дескриптор (на рис. 3.13 это **0**, **1**, **2**, **3** и **4**). Посылать сообщения через такое соединение можно только с помощью функций вроде *read()* и *write()*. Когда же соединение устанавливает функция *ConnectAttach()* (или, конечно, функция *name_open()*) с флагом `_NTO_SIDE_CHANNEL`), полученные дескрипторы будут обозначаться с буквой "s" (**0s**, **1s** и т. п.). Через такое соединение сообщения посылают уже с помощью функции *MsgSend()* и ее сестер.

Во втором столбце (**Server Name**) таблицы указаны имя и (в скобках) PID процесса-хозяина канала. В третьем столбце (**IOFlags**) указываются флаги, с которыми открывался файл, в четвертом (**Seek Offset**) — значение файлового указателя, в пятом (**Resource Name**) — префикс, ассоциированный с каналом.

Напоследок еще одно представление — **File System Navigator** (рис. 3.14).

Рис. 3.14. Представление **File System Navigator —** доступ к файловой системе целевого узла QNX Neutrino

Это представление является по сути файловым менеджером, отображающим файловую систему целевого узла QNX Neutrino. Этот файловый менеджер позволяет манипулировать файлами и каталогами

целевой системы, но не позволяет организовать обмен файлами между целевой и инструментальной системами — для этого есть иные инструменты.

Остальные представления вы можете изучить самостоятельно.

3.4. "Импульсы" и события

Вторым "родным" механизмом IPC в QNX Neutrino и по очереди, и по важности является, безусловно, механизм Pulses, по-русски обычно называемый "импульсами". Пользователи QNX4 знакомы, конечно, с предшественником импульсов — механизмом "прокси".

Механизм импульсов часто используется клиентами и операционной системой для извещения сервера о наступлении какого-то события. Но бывает, что и клиенту требуется получать уведомления о событии от сервера. Для этого компания QSS рекомендует использовать унифицированную надстройку, именуемую *доставкой событий* (Event Delivery). Эта надстройка позволяет серверу единым способом уведомлять клиента о наступлении событий, используя при этом те механизмы операционной системы, которые были заданы клиентом. При этом серверу нет необходимости вникать, какой именно механизм задал клиент для извещения о наступлении события.

3.4.1. Импульсы: "родные" асинхронные сообщения Neutrino

У импульсов есть два главных отличия от сообщений QNX Neutrino:

- импульс не блокирует отправителя, т. е. является *асинхронным* механизмом IPC. Ядро хранит отправленные импульсы в некотором буфере до тех пор, пока сервер не соблаговолит вызвать функцию получения импульса;
- импульс имеет фиксированный размер 40 бит (8 бит кода импульса + 32 бита данных).

Клиент, посылающий импульс, задает приоритет доставки (шкала приоритетов импульсов такая же, как шкала приоритетов потоков).

Импульс отправляется с помощью функции *MsgSendPulse()*, а приниматься может как специально обученной функцией *MsgReceivePulse()*, так и обычной *MsgReceive()*.

Примечание

Как же мы узнаем, что принято функцией *MsgReceive()* — сообщение или импульс? Вспомним, что функция *MsgReceive()* возвращает некоторый идентификатор, используемый для отправки ответного сообщения. На импульс же ответить нельзя, поэтому при получении импульса *MsgReceive()* возвращает 0.

Для отправки импульса клиент должен создать обычное соединение с каналом сервера и выполнить вызов вроде следующего:

```
status = MsgSendPulse(coid, 10, 123, 333);
```

где `status` — код возврата (`int`); `coid` — идентификатор соединения; `10` — значение приоритета (`int`); `123` — код импульса (имеет тип `int`, но помните, что под него выделено только 8 бит, поэтому он должен иметь значение в диапазоне от `_PULSE_CODE_MINAVAIL` до `_PULSE_CODE_MAXAVAIL`, заданных в заголовочном файле `sys/neutrino.h`); `333` — данные импульса (`int`).

Примечание

Код импульса может иметь и отрицательное значение, только эти значения зарезервированы разработчиками QNX Neutrino и используются для внутренних задач операционной системы.

Для приема импульса сервер должен вызвать:

```
rcvid = MsgReceivePulse(chid, &Pulse, sizeof(Pulse), NULL);
```

или

```
rcvid = MsgReceive(chid, &Pulse, sizeof(Pulse), NULL);
```

Что такое `rcvid` и `chid`, вам, как уже матерым QNX-девелоперам, известно. Структура `Pulse` должна быть заблаговременно создана так:

```
struct _pulse Pulse;
```

Получив импульс, можно с ним что-нибудь сделать — например, вывести на терминал его код и данные:

```
printf("Code=%d, Data=%d\n", Pulse.code,
                                    Pulse.value.sival_int);
```

В заключение остается добавить, что такой механизм, как *асинхронные сообщения*, появился в QNX Neutrino, начиная с версии 6.3.0. Этот механизм новый, внедрен в порядке эксперимента, посему пользоваться им нужно весьма осмотрительно.

Примечание

Напомню, что функции асинхронных сообщений имеют такие же имена, как функции обычных сообщений, но с префиксом *asyncmsg_* (например, *asyncmsg_MsgSend()*).

3.4.2. Уведомление о событиях

Для обеспечения универсальности механизма доставки событий используется структура, которую можно создать, например, так:

```
struct sigevent my_event;
```

В этой структуре есть поле `sigev_notify`. С помощью значения этого поля клиент указывает способ (т. е. механизм ОС), с помощью которого клиент желает получать извещения о событии. Можно задавать следующие способы:

- *Извещение сигналом*. На стороне клиента, разумеется, надо принять меры по обработке такого извещения. Есть три варианта такого извещения:
  - `SIGEV_SIGNAL` — обычный POSIX-сигнал;
  - `SIGEV_SIGNAL_CODE` — сигнал реального времени (т. е. тот, который может содержать 4 байта кода);
  - `SIGEV_SIGNAL_THREAD` — доставлять сигнал конкретному потоку, т. е. используя функцию *pthread_kill()*.
- *Извещение импульсом* — `SIGEV_PULSE`. На стороне клиента надо будет создать канал для приема этих импульсов и все, о чем говорилось в *разд. 3.4.1*: задать значения кода импульса, данных импульса, приоритета доставки импульса.
- *Извещение с помощью прерывания* — `SIGEV_INTR`. Клиент должен "ловить" такие события, используя функцию *InterruptWait()*.
- *Извещение путем разблокирования потока* — `SIGEV_UNBLOCK`. Этот способ используется в тайм-аутах ядра.

Остальные поля структуры `my_event` заполняются в зависимости от значения поля `sigev_notify`. Для удобства можно заполнять структуру с помощью макросов *SIGEV_SIGNAL_INIT()*, *SIGEV_PULSE_INIT()*и т. п.

Итак, сначала клиент создает и заполняет так, как ему нравится, структуру `my_event`, затем эта структура отправляется с помощью функции *MsgSend()* серверу — т. е. по сути клиент делает заявку на

извещение каким-то способом о наступлении какого-то события. Сервер сохраняет как саму структуру, так и rcvid сообщения-заявки, которым структура была получена. Сервер с помощью *MsgReply()* уведомляет клиента о том, что он принял заявку на уведомление о событиях.

При возникновении интересующего события сервер уведомляет клиента с помощью функции *MsgDeliverEvent()*, которая получает в качестве аргументов два параметра — сохраненное значение `rcvid` и указатель на сохраненную структуру `&my_event`. Клиенту посылается импульс, сигнал или что он еще там хотел, при этом клиент сам знает, как "отловить" событие.

Примечание

На самом деле это достаточно удобно ☺.

Глава 4. Управление ресурсами ЭВМ

— Старшина, быстро отправляй курс на плац и, через барабан, в клуб!
(Пример управления ресурсами в ПВУРЭ ПВО)

Управление ресурсами ЭВМ — одна из главных функций любой операционной системы. К основным ресурсам, которыми управляет ОС QNX, относятся:

- файловые системы;
- символьные устройства ввода/вывода (последовательные и параллельные порты, сетевые карты и т. д.);
- виртуальные устройства ("нуль"-устройство, псевдотерминалы, генератор случайных чисел и т. п.).

В QNX Neutrino поддержка ресурсов не встроена в микроядро и организована с помощью специальных программ и динамически присоединяемых библиотек, называемых *администраторами ресурсов*. Взаимодействие между администраторами ресурсов и другими программами реализовано через четко определенный, хорошо документированный POSIX-интерфейс файлового ввода/вывода. Хотя, конечно, на самом деле весь обмен построен на базовом механизме QNX-сообщений.

4.1. Администраторы ресурсов

Администратор ресурсов — это прикладная серверная программа, выполняющая сервисные услуги для программ-клиентов и предоставляющая клиентам POSIX-интерфейс ввода/вывода. То есть для того, чтобы написать клиентскую программу для взаимодействия с грамотно написанным Администратором ресурсов, прикладному программисту не надо знать особенности QNX Neutrino. Разумеется, клиентский API, соответствующий POSIX, является надстройкой над QNX-механизмами IPC, поэтому реально действия Администратора ресурсов заключаются в приеме QNX-сообщений от клиентов и, при необходимости, взаимодействии с аппаратурой. Для связи между

программой-клиентом и Администратором ресурсов используется механизм пространства имен Администратора процессов. Администратор ресурсов работает так:

1. Выполняется инициализация интерфейса сообщений, при этом создается канал, по которому клиенты могут посылать свои сообщения Администратору ресурсов.
2. Регистрируется путевое имя (т. е. зона ответственности) в пространстве имен Администратора процессов.
3. Вызывается функция приема сообщений от клиентов.
4. С помощью операторов `switch/case` выполняется переключение на нужный обработчик в зависимости от типа сообщения.
5. Возврат к п. 3.

По такой схеме работают все администраторы ресурсов. Можно, разумеется, "наворотить" логику Администратора ресурсов, но описанный костяк сохраняется.

4.2. Файлы и каталоги

Все данные в операционной системе хранятся в виде файлов. Одной из важнейших функций любой ОС является способность манипулировать файлами, размещенными на различных физических носителях (магнитных дисках, микросхемах ПЗУ и т. п.). В QNX Neutrino, как и во всех UNIX-подобных системах, для доступа к аппаратным устройствам также используется файловый интерфейс, поэтому управление ресурсами целесообразно начать с рассмотрения понятия файла в QNX Neutrino.

4.2.1. Типы файлов

Файл — это набор байтов, имеющий общие атрибуты:

- имя файла;
- идентификатор владельца и идентификатор группы;
- атрибуты доступа (для владельца, для членов группы и для остальных пользователей);
- метки времени (время создания файла, время последней модификации файла, время последнего доступа к файлу, время последней записи в файл);

- тип файла;
- счетчик ссылок;
- другие атрибуты.

Имя файла с добавлением *пути* — списка разделенных символом "/" имен вложенных каталогов, содержащих файл (начиная с корневого каталога), называется *полным* или *путевым именем файла*. Например, `/usr/photon/bin/slideviewer`. Здесь файл с именем `slideviewer` имеет путь `/usr/photon/bin`.

QNX обеспечивает поддержку следующих типов файлов:

- обычные (regular) файлы;
- каталоги;
- жесткие ссылки;
- символические ссылки;
- именованные программные каналы (FIFO);
- блок-ориентированные специальные файлы;
- байт-ориентированные специальные файлы;
- именованные специальные устройства (Named Special Device).

Расширенную информацию о файлах можно посмотреть, например, выполнив команду **`ls -l`**. При этом самый первый символ строки, соответствующей каждому файлу, обозначает тип файла. Для просмотра той же информации в файловом менеджере нужно щелкнуть правой кнопкой мыши на имени файла и выбрать элемент **Inspect** в появившемся меню. Откроется окно, показанное на рис. 4.1.

Рис. 4.1. Окно **File Information**

Это окно позволяет изменять некоторые атрибуты.

4.2.2. Обычные файлы

Файл этого типа — последовательность байтов, не имеющая (с точки зрения QNX) предопределенной структуры. За интерпретацию содержимого обычных файлов отвечают конкретные приложения. В сложившейся практике приложение "узнает" свой файл по расширению его имени (т. е. по части имени файла, идущей после точки). Например: `myfile.c` — это исходный текст программы на

языке C, `myfile.pdf` — это документ в формате PDF (Portable Document Format) фирмы Adobe. В отличие от операционных систем семейства Windows, QNX узнает исполняемые файлы не по расширению (вроде `exe`), а по специальному атрибуту. Для того чтобы при двойном щелчке мышью на имени файла в окне файлового менеджера Photon (**`pfm`**) автоматически запускалось нужное приложение, необходимо добавить соответствующую ассоциацию *(см. разд. 1.2.4)*.

> ***Примечание***
> При выполнении команды **`ls -l`** обычный файл будет обозначен символом **`-`**.

4.2.3. Каталоги

Каталоги — это по сути обычные файлы, имеющие определенную структуру. Каталог представляет собой набор записей определенного формата, называемых *элементами каталога*. Первым элементом каталога всегда является запись о файле с именем "`.`". Этот элемент является ссылкой "на самого себя", указывая на соответствующую самому себе запись в родительском каталоге. Вторым элементом каталога всегда является запись о файле с именем "`..`". Этот элемент ссылается на родительский каталог, указывая на его первый элемент. Таким образом, первые два элемента каталога всегда существуют и имеют известное содержимое. Если каталог является корневым для данного физического раздела, то оба эти элемента ссылаются на собственный каталог.

> ***Примечание***
> На всякий случай напомню, что штатные средства просмотра файловой системы по умолчанию не отображают файлы (а каталог, как мы уже выяснили, это просто тип файла), имена которых начинаются с точки (`.`). Такие файлы называют "скрытыми" и обычно в них содержится системная и конфигурационная информация, а "спрятаны" они для повышения "дуракоустойчивости" операционной системы.

Каждый элемент каталога связывает имя некоторого файла со служебной информацией о нем, включающей ссылку на место физического хранения данных. Содержимое элемента каталога можно представить так:

- 16 байт для имени файла;
- размер файла;

- информация о физическом размещении содержимого файла на диске;
- метки времени;
- атрибуты доступа;
- счетчик ссылок на физические данные;
- тип файла;
- статус ("закрыт" или "открыт").

Вы, конечно же, заметили, что для имени файла в элементе каталога отведено всего 16 байт. Что же делать, если требуется задать для файла более длинное имя? В файловой системе QNX (о которой мы поговорим далее) есть служебный файл `/.inodes` (Information Nodes — "информационные узлы", эти "узлы" — дальние родственники индексных узлов UNIX). Если длина имени какого-нибудь файла превысит 16 символов (т. е. 16 байт), то в файле `/.inodes` будет создана запись для этого файла, в которую переместится вся информация о файле, кроме имени. В элементе каталога, относящемся к данному файлу, останутся имя файла (теперь его длина может достигать 48 символов) и, разумеется, ссылка на запись в файле `/.inodes`.

С широким распространением платформы Java 2 Micro Edition (Sun Microsystems) возникла необходимость поддержки во встраиваемых системах еще более длинных имен файлов. Разработчики QSS добавили в версии 6.2.1 очень простое решение: "лишняя" часть имен файлов помещается в файл `.longfilenames`, аналогичный файлу `.inodes` Это позволяет задавать имена длиной до 505 символов. Если файловую систему с такими длиннющими именами смонтировать в прежней версии QNX Neutrino, которая еще не знала о существовании подобных имен, то "урезание" будет выполняться тем же способом, каким DOS урезает длинные имена Windows.

> ***Примечание***
> При выполнении команды **`ls -l`** каталог будет обозначен символом **`d`**.

4.2.4. Жесткие ссылки

Хорошо, ну а если для одних и тех же физических данных файла создать еще один элемент каталога? Да хоть дюжину! Такие дополнительные элементы каталога (т. е. имена) называют *жесткими ссылками* (или "жесткими связями"). При создании жесткой ссылки, во-первых, информация о физическом размещении данных выносится в

файл `/.inodes`, во-вторых, счетчик ссылок (атрибут файла) увеличивается на единицу. При удалении одной из жестких ссылок реально будет удален только соответствующий элемент каталога, а счетчик ссылок на inode-запись будет уменьшен на единицу. Как только счетчик достигнет значения "ноль", и inode-запись, и физические данные файла будут уничтожены.

> ***Примечание***
> Справедливости ради заметим, что для уничтожения файла есть еще одно обязательное условие — статус файла должен иметь значение "закрыт".

Вы можете спросить, удаляется ли inode-запись если счетчик стал равен единице или если длина имени файла уменьшилась, скажем, до десяти символов? Нет, не удаляется. Созданная inode-запись сохраняется, пока существует файл.

> ***Примечание***
> Нельзя создавать жесткие связи для каталогов, кроме уже существующих — "`.`" и "`..`".

> ***Примечание***
> При выполнении команды **`ls -l`** жесткая ссылка обозначается так же, как файл, на который она ссылается, при этом счетчик ссылок будет иметь значение больше 1.

4.2.5. Символические ссылки

Еще один весьма полезный тип файлов — *символические ссылки* (в UNIX-системах их обычно называют "мягкими" ссылками). Это, по сути дела, текстовый файл, содержащий имя другого файла или каталога, к которому перенаправляются все запросы ввода/вывода.

У символической ссылки есть важное достоинство — она может указывать на файл или каталог, находящийся на другом физическом носителе (например, в другом разделе диска или на другом узле сети).

Возможность создания символических ссылок для каталогов создает опасность бесконечных циклов. Поэтому число переходов по символическим ссылкам ограничено значением переменной `SYMLOOP_MAX`, определенным в заголовочном файле `<limits.h>`.

Примечание

При выполнении команды **ls -l** символическая ссылка обозначается символом **l**, при этом к имени файла добавляется стрелка с именем того файла, на который сделана ссылка.

4.2.6. Именованные программные каналы (FIFO)

Именованные программные каналы (FIFO) предназначены для организации взаимодействия между двумя или более процессами: один процесс пишет в программный канал, другой читает из программного канала.

Скажем прямо, FIFO — далеко не самый быстрый способ межзадачного взаимодействия. Но он не лишен достоинств: во-первых, данные, записанные в FIFO, сохраняются при отключении питания, во-вторых, взаимодействующие процессы не должны передавать друг другу никакие дескрипторы или идентификаторы (но им должно быть известно имя FIFO-файла).

Примечание

При выполнении команды **ls -l** FIFO-файл будет обозначен символом **f**.

4.2.7. Блок-ориентированные специальные файлы

Блок-ориентированные специальные файлы (очень часто их называют "блочными устройствами") — файлы, предназначенные для того, чтобы скрыть от приложений физические характеристики аппаратуры. Слово "блочный" говорит о том, что обмен данными осуществляется блоками по несколько байт (например, при работе с жестким диском обычный размер блока — 512 байт). В QNX блок-ориентированные специальные файлы создаются не на диске, а в оперативной памяти. Когда же они создаются? При старте соответствующих драйверов. Например, драйвер EIDE создает блок-ориентированные файлы: `/dev/hd0` — для доступа к первому жесткому диску, `/dev/hd1` — для доступа ко второму жесткому диску и т. д.

Примечание

При выполнении команды **ls -l** блочное устройство будет обозначено символом **b**.

4.2.8. Байт-ориентированные специальные файлы

Байт-ориентированные специальные файлы (очень часто их называют "символьными устройствами") — это файлы, аналогичные блочным устройствам, с той разницей, что символьные устройства обеспечивают интерфейс к аппаратуре, осуществляющей посимвольный ввод/вывод. К такой аппаратуре относятся, например, последовательный порт, сетевая карта и т. п. Так же как блочные устройства, байт-ориентированные специальные файлы создаются драйверами при запуске. Например, драйвер Ethernet-карты создает файл `/dev/en0`.

> ***Примечание***
> При выполнении команды **`ls -l`** символьное устройство будет обозначено символом **`c`**.

4.2.9. Именованные специальные устройства (Named Special Device)

Этот тип специфичен для QNX. Дело в том, что благодаря своей универсальности, байт- и блок-ориентированные специальные файлы могут использоваться приложениями не только для обмена данными с драйверами устройств, но и для взаимодействия с другими программами. В этих случаях приложения, создающие специальные файлы, будут являться как бы программными устройствами. Поскольку при обмене не всегда удобно представлять данные в виде символов и блоков, потребовалось ввести специальный дополнительный тип файлов — Named Special Device. Разумеется, обмен посредством этого типа файлов требует знания формата данных от всех участников обмена. Например, для взаимодействия компонентов графической системы используется специальный файл `/dev/photon`.

> ***Примечание***
> При выполнении команды **`ls -l`** именованные специальные устройства будут обозначены символом **`n`**.

4.3. Администраторы файловых систем

Файловая система в QNX Neutrino имеет иерархическую древовидную структуру. Вершиной иерархии является *корневой* каталог ("root"), обозначаемый символом "`/`". Поскольку QNX Neutrino предназначена

для работы на таких ЭВМ, в которых, например, могут отсутствовать как жесткие, так и гибкие магнитные диски, структура файловой системы не должна зависеть от физического размещения файлов. Для решения этой задачи в QNX реализован механизм *пространства путевых имен*. Подробно этот механизм обсуждается в *главе 3*, посвященной архитектуре QNX. Пока же достаточно представлять, что фрагменты единого дерева файловой системы могут размещаться на разных физических носителях. Вам уже известно, что драйверы устройств при старте создают байт- и блок-ориентированные специальные файлы, служащие для "сырого" доступа к устройствам. Некоторые такие файлы могут "содержать" файловые системы какого-либо типа, например файл `/dev/hd0t77` содержит "родную" файловую систему QNX Neutrino — раздел QNX4. Как же получить доступ к данным, размещенным на этих носителях? Для этого, во-первых, нужен Администратор ресурсов, "знающий" структуру данного типа раздела диска. Такой Администратор ресурсов называется *Администратором файловой системы*. Во-вторых, необходимо, чтобы Администратор файловой системы присоединил каталоги и файлы, содержащиеся на "своем" физическом носителе, к единой файловой иерархии, т. е. зарегистрировал свой префикс в пространстве путевых имен Администратора процессов. Эта операция называется *монтированием* файловой системы. Имя зарегистрированного префикса (т. е. корневого каталога монтируемого раздела), называется *точкой монтирования*. Для монтирования и демонтирования файловых систем предназначены утилиты **`mount`** и **`umount`** (обе утилиты может запускать только пользователь `root`). Следующая команда монтирует содержимое раздела, представленного блок-ориентированным специальным файлом `/dev/hd0t77`, в каталог `/fs/QNX4`:

```
mount /dev/hd0t77 /fs/QNX4
```

А такая команда отмонтирует раздел от пространства путевых имен:

```
umount /fs/QNX4
```

Точку монтирования можно перегружать, т. е. монтировать в одну точку несколько разделов.

Файловые системы, поддерживаемые в QNX, можно классифицировать по следующим типам.

- 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 его достаточно 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, где не требуется хранить данные на энергонезависимом носителе и 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-qnx4.so`**) — высокопроизводительная файловая система, сохранившая формат и структуру дисков ОС QNX4, но 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 его расширений (Rock Ridge, Joliet, Kodak Photo CD и аудио).
  - Файловая система Ext2 (**`fs-ext2.so`**) обеспечивает прозрачный доступ из среды QNX к Linux-разделам жесткого диска версии как 0, так и 1.
- 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 клиентской рабочей станции прозрачный доступ через сеть к файлам независимо от операционных систем, используемых файл-серверами. NFS использует механизм удаленного вызова процедур (RPC, Remote Procedure Call) и работает поверх TCP/IP. Реализована с помощью администраторов ресурсов **`fs-nfs2`** и **`fs-nfs3`**.
  - Файловая система CIFS (Common Internet File System) обеспечивает клиентским станциям прозрачный доступ к сетям Windows, а также к UNIX-системам с запущенным сервером SMB (Server Message Block) Работает поверх TCP/IP. Реализована с помощью Администратора ресурсов **`fs-cifs`**[1]..
- Virtual — особые файловые системы, обеспечивающие специфические функциональные возможности при работе с другими файловыми системами.
  - Пакетная файловая система — обеспечивает привычное для пользователя представление файлов и каталогов, хранящихся в каталогах, называемых *пакетами*. Реализована с помощью Администратора ресурсов **`fs-pkg`**.
  - Inflator — обеспечивает динамическое разжатие при открытии файлов, сжатых утилитой **`deflate`** и содержащихся в заданном каталоге. Реализована с помощью Администратора ресурсов **`inflator`**.

Поскольку у некоторых файловых систем, работающих в ОС QNX, много общих черт, 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

---

[1] Администратор **`fs-cifs`** удобно использовать, когда нужно получить прозрачный доступ к разделенным папкам Windows. Для доступа к принтерам Windows и для предоставления Windows-клиентам доступа к папкам QNX используют свободно-распространяемый пакет Samba (SMB, произносится "самба").

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 специальных файлов устройств. По умолчанию они обозначаются hd*n* (для жесткого диска) и cd*n* (для CD-ROM), где *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/hd0t11.1.

Модуль **io-blk.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 файловых систем, также реализованные в виде динамически присоединяемых библиотек. Поддержка блочных файловых систем реализована в модулях администраторов ресурсов:

- **fs-qnx4.so** — файловая система QNX4;
- **fs-ext2.so** — файловая система Ext2;
- **fs-dos.so** — файловая система FAT32;
- **fs-cd.so** — файловая система 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 — в точку `/fs/hd0-dos`.

Для того чтобы программа **`diskboot`** могла использовать раздел для поиска базового образа файловой системы QNX, необходимо существование 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`**.

4.4. Администраторы символьных устройств 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 устройства;
- параллельные устройства;
- псевдотерминалы (ptys).

Для повторного использования кода управления символьными устройствами соответствующие администраторы ресурсов компонуются со статической библиотекой `io-char`. Эта библиотека 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 никакое редактирование принимаемых данных;
- редактируемый (edited) — режим, при котором `io-char` может выполнять операции редактирования с каждым символом строки. По окончании редактирования строки она становится доступной для обработки прикладным процессом (обычно — после ввода символа возврата каретки `CR`). Такой режим часто называют *каноническим*.

4.4.1. 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 сеансов 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.

Кстати, в QNX4 количество виртуальных консолей указывается опциями утилиты инициализации консоли **`tinit`**. В Neutrino число виртуальных консолей задается непосредственно с помощью опций **`devc-con`**.

> ***Примечание***
> Поскольку в стандартной системе **`devc-con`** запускается процессом **`diskboot`**, количество виртуальных консолей указывается через опции утилиты **`diskboot`** при ее запуске из стартового сценария. Например:
>
> ```
> diskboot –b1 – D1 –odevc-con,–n4
> ```
>
> В приведенной команде видно заданное количество виртуальных консолей — 4. Подробнее стартовый сценарий описан в *главе 6*.

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.

4.4.2. Последовательные устройства

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

4.4.3. Параллельные устройства

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

4.4.4. Псевдотерминалы (ptys)

Псевдотерминалы управляются драйверным процессом **`devc-pty`**. С помощью аргумента при запуске драйвера определяется количество псевдотерминалов.

4.4.5. USB-устройства

В QNX Neutrino реализована поддержка стека USB версии 2.0. Основным программным модулем является сервер **`io-usb`**, который может загружать три драйвера контроллеров USB в любой комбинации:

- `devu-ohci.so`
- `devu-ehci.so`
- `devu-uhci.so`

Какие USB-устройства имеются в вашей ЭВМ, можно посмотреть специально обученной утилитой **`usb`**.

Например, запустим **`io-usb`** с двумя из трех перечисленных драйверов:

```
io-usb -dehci -duhci
```

Для использования USB-устройств ввода (например, мыши) нужно после запуска USB-сервера указать администратору этих устройств **`io-hid`** загрузить драйвер `devh-usb.so`:

```
io-hid -dusb
```

Что бы использовать USB-принтер, нужно запускать драйвер **`devu-prn`**, а для использования USB-устройств Mass Storage (флэш-память) — драйвер **`devb-umass`**.

Подробностей не ждите — очень не хочется переписывать книгу "QNX Neutrino Realtime Operating System: Utilities Reference", которая в HTML-формате поставляется в составе QNX Momentics.

4.5. Файловая система QNX4

"Родной" раздел QNX Neutrino называется *файловой системой QNX4*. Почему 4? Потому что структура и организация данных в файловой системе QNX Neutrino не отличается от QNX4.*xx* (хотя программная реализация работы с этими структурами данных, разумеется, отличается).

Файловая система QNX4 соответствует требованиям стандарта POSIX, поэтому в документации QNX она часто называется *файловой системой POSIX*. Она поддерживает многопоточную обработку запросов с клиент-управляемым приоритетом.

4.5.1. Создание раздела QNX4

Новый раздел QNX создается утилитой **`fdisk`**. Назначение этой утилиты то же, что у одноименных утилит других операционных систем.

> ***Примечание***
> Для изменения физической структуры диска нужно обладать правами `root` или, как минимум, иметь доступ по записи к соответствующим файлам устройств.
>
> Если на жестком диске предполагается создавать разделы не только QNX, то рекомендуется создать раздел QNX в последнюю очередь (хотя, пожалуй, средства редактирования физической структуры дисков в Linux достаточно гибки, так что можно не беспокоиться за загрузочные секции, созданные QNX).

Для использования утилиты **`fdisk`** нужно, чтобы существовал соответствующий блок-ориентированный специальный файл диска `/dev/hd`*X*, где *X* — порядковый номер диска. Эти файлы (читайте — префиксы) создаются (конечно же — регистрируются) Администратором буферного кэша **`io-blk.so`**, который загружается драйвером контроллера (например, драйвером EIDE — **`devb-eide`**).

Вообще, в QNX принято такое соглашение об именовании разделов жесткого диска:

- первое число (после hd) — это порядковый номер жесткого диска;
- второе число (после t) означает тип раздела диска (табл. 4.1).

Таблица 4.1. *Соглашения о наименовании*

Тип	**Файловая система**
11	FAT32 — раздел Windows
77	QNX — раздел QNX4
78	QNY — раздел QNX4
79	QNZ — раздел QNX4
131	Ext2 — раздел Linux

Соответственно, если на единственном диске есть два раздела FAT32, то имена их блок-ориентированных специальных файлов будут примерно такими — /dev/hd0t11 и /dev/hd0t11.1.

Поскольку QNX Neutrino 6.3.0 устанавливается в раздел 79 (QNZ), мы можем создать еще раздел 77 (QNX) или 78 (QNY). Давайте сделаем это (надеюсь, на диске есть свободное место и существует не более трех первичных разделов):

```
fdisk /dev/hd0 add qny half
```

В результате создан раздел 78, использующий половину свободного пространства, пока еще не имеющий структуры раздела QNX4. Теперь нужно перечитать таблицу разделов диска hd0:

```
mount –e /dev/hd0
```

В каталоге /dev появится файл hd0t78. Свежеиспеченный раздел QNY необходимо инициализировать с помощью утилиты **dinit**:

```
dinit -h /dev/hd0t78
```

В результате будет создан раздел QNX4, содержимое которого можно посмотреть так:

```
mount /dev/hd0t78 /mnt
ls /mnt
```

4.5.2. Ключевые компоненты раздела QNX4

Итак, только что созданный раздел содержит следующие ключевые компоненты:

- блок загрузчика;
- корневой блок;
- битовая матрица (или битовая карта);
- корневой каталог.

Блок загрузчика

Блок загрузчика, или загрузочный блок, — это первый физический блок в разделе диска. Этот блок содержит так называемый IPL (Initial Program Loader) — начальный загрузчик. Назначение IPL мы рассмотрим чуть подробнее в *главе 6*. Сейчас нам достаточно знать, что IPL — это код, загружающий образ QNX Neutrino. В случае, если диск не разбит на разделы, блок загрузчика будет первым физическим блоком на диске.

Корневой блок

Корневой блок имеет структуру обычного, содержащего записи следующих файлов:

- корневой ("root") каталог данного раздела QNX4. Жесткие связи `"."` и `".."` этого каталога ссылаются на него же;
- файл `/.inodes` (его назначение мы обсуждали в начале этой главы);
- файл `/.boot` — содержит загружаемый образ операционной системы QNX (как раз его и загружает IPL);
- файл `/.altboot` — содержит резервный загружаемый образ операционной системы QNX. Если этот файл не пуст, то в начале загрузки система предложит загрузить `/.altboot` в качестве опции (по умолчанию будет загружен образ, хранящийся в файле `/.boot`, *см. главу 6*);
- файл `/.longfilenames` — служит для поддержки имен файлов длиной от 48 до 505 символов.

Битовая матрица

Битовая матрица, или битовая карта (bitmap) — файл, содержащий столько битов, сколько блоков в разделе диска. Каждому физическому блоку раздела соответствует один бит. Если бит установлен в значение 1, значит, соответствующий ему блок занят. Битовая матрица

используется для выделения физических блоков на диске и содержится в файле `/.bitmap`.

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 (т. е. отмечены в битовой матрице как 1), но не использованы. Вернуть эти блоки можно, запустив утилиту **`chkfsys`** (см. далее).

Корневой каталог

Корневой каталог раздела ведет себя, как обычный каталог, за двумя исключениями:

- жесткие связи `"."` и `".."` корневого каталога раздела являются ссылками на этот же корневой каталог;
- корневой каталог всегда содержит записи файлов `/.bitmap`, `/.inodes`, `/.boot` и `/.altboot`.

> ***Примечание***
> Для того чтобы программа **`diskboot`** могла использовать раздел для поиска базового образа файловой системы QNX, необходимо наличие файла `/.diskroot`, а для поддержки длинных имен используется файл `.longfilenames`.

4.5.3. Диагностика файловой системы

После инициализации раздела с помощью утилиты **`dinit`** рекомендуется выполнить проверку физической целостности раздела с помощью утилиты **`dcheck`**:

```
dinit -h /dev/hd0t77
dcheck -m /dev/hd0t77
```

Утилита **`dcheck`** проверяет целостность каждого блока. Сбойные и предотказные блоки будут удалены из битовой матрицы (`/.bitmap`) и отмечены в специальном файле `/.bad_blks`.

Проверку *логической* целостности раздела QNX4 выполняет утилита **`chkfsys`**. Эта утилита анализирует каждый файл, обнаруживая и корректируя нарушения структуры метаданных. Кроме того, утилита **`chkfsys`** возвращает в систему неиспользуемые блоки, помеченные как используемые (так называемые "потерянные блоки"). Для этого **`chkfsys`** в ходе проверки файлов создает собственную битовую

матрицу. Если полученная утилитой битовая матрица отличается от файла `/.bitmap`, то оператору будет предложено сохранить ее в файле `/.bitmap`.

Утилиту **`chkfsys`** можно запустить с опцией **`-z`** ***`имя_файла`***. Тогда в файл *`имя_файла`* (а он должен размещаться *за пределами* проверяемого раздела) будет помещен список файлов, содержащих сбойные блоки. Эти файлы можно удалить с помощью утилиты **`zap`**. Поскольку утилита **`zap`** просто удаляет соответствующую файловую запись из каталога, можно впоследствии выполнить обратную операцию **`zap -u`**.

Для проверки логической целостности раздела FAT32 предназначена утилита **`chkdosfsys`**. По своим функциям она аналогична Windows-утилите Scandisk.

Для контроля использования дискового пространства предназнчены утилиты:

- **`df`** — POSIX-утилита, показывающая использование дискового пространства смонтированных разделов;
- **`du`** — POSIX-утилита, показывающая размер файлов и каталогов.

Обе эти утилиты выводят информацию в блоках, поэтому удобно запускать их с флагом **`-k`**, тогда результат будет выводиться в килобайтах.

Для поиска файлов и каталогов используют:

- в командной строке — утилиту **`find`**;
- в среде Photon — элемент меню **Launch > Utilities > Find**.

4.6. Разграничение доступа к ресурсам

По сути дела, данные могут находиться в двух "местах" — храниться в виде файлов на носителе (обычно — на жестком диске) или размещаться в ОЗУ, будучи частью какого-либо процесса. С точки зрения ЭВМ, данными оперируют не пользователи, а выполняющиеся процессы. Таким образом, задача разграничения доступа сводится к двум подзадачам:

- обеспечить механизм, позволяющий пользователям запускать только определенные программы;

- обеспечить механизм, позволяющий процессам работать только с определенными файлами.

Оба этих механизма реализованы с помощью файловых атрибутов доступа. В *главе 2* мы говорили, что каждому процессу при создании присваивается четыре идентификатора: реальный и эффективный идентификаторы пользователя (UID и EUID) и реальный и эффективный идентификаторы группы (GID и EGID). Как же происходит привязка этих идентификаторов к конкретному человеку? Доступом к файлам управляют с помощью проверки прав доступа и изменения атрибутов файла.

4.6.1. Проверка прав доступа

Каким образом регулируется возможность приложений выполнять операции чтения/записи с файлами? Каждая выполняющаяся программа (т. е. процесс) имеет четыре идентификатора:

- идентификатор владельца (UID);
- идентификатор группы (GID);
- эффективный идентификатор владельца (EUID);
- эффективный идентификатор группы (EGID).

Про эти атрибуты мы еще поговорим в *разд. 2.2.1*, посвященном процессам. Сейчас достаточно знать то, что для проверки прав доступа к файлу используются только эффективные идентификаторы.

Вам уже известно, что каждый файл имеет атрибуты доступа для трех "личностей" (см. рис. 4.1): владельца (**User**), группы (**Group**) и "остальных" (**Other**). Для каждой из этих "личностей" можно задавать права на чтение, запись и исполнение.

Итак, в разграничении доступа к файлам участвуют EUID и EGID процесса и файловые атрибуты доступа. Когда процесс пытается открыть для записи какой-либо файл, QNX сначала смотрит, совпадает ли EUID процесса с идентификатором владельца файла. Если совпадает, то проверяется, есть ли у владельца право на запись, — если есть, то файл открывается с разрешением на запись, если нет, — процесс получит уведомление "Permission Denied" (доступ запрещен). Конечно, EUID может не совпасть с идентификатором владельца файла. Тогда проверяется, совпадает ли EGID процесса с файловым идентификатором группы. Если никакие идентификаторы не совпали, то к данному процессу применяются атрибуты доступа, установленные для "остальных".

Примечание

Для процессов с EUID = 0 (т. е. для пользователя `root`) доступ к файлам предоставляется без процедуры проверки прав. Разумеется, это не означает, что пользователь `root` может запустить на исполнение неисполняемый файл ☺.

В окне **File Information** (см. рис. 4.1) есть два флажка (**Set User ID** и **Set Group ID**), соответствующих дополнительным атрибутам SUID и SGID (эти атрибуты имеют смысл только для исполняемых файлов). В выходной информации команды **`ls -l`** эти атрибуты отображаются путем замены **`x`** на **`S`** в атрибутах владельца (для SUID) и в атрибутах группы (для SGID). Если же снять доступ по исполнению при сохранении флага SUID или/и SGID, то соответствующая заглавная буква **`S`** будет заменена строчной буквой **`s`**.

Назначение атрибутов SUID и SGID заключается в следующем. При запуске процесса его идентификаторы UID, GID, EUID, EGID устанавливаются равными соответствующим идентификаторам родительского процесса. Если же для исполняемого файла установлен атрибут SUID, то идентификаторы UID и EUID нового процесса будут установлены равными идентификатору владельца файла. Атрибут SGID делает то же с групповыми идентификаторами процесса. То есть если существует исполняемый файл **`myfile`**, принадлежащий пользователю `user1`, с атрибутами, разрешающими исполнение файла всем пользователям, то у процесса, созданного при запуске утилиты **`myfile`** пользователем `user2`, значения UID и EUID будут равны не `user2`, как следовало бы ожидать, а `user1`.

Нетрудно догадаться, что такие флаги весьма небезобидны с точки зрения защиты информации. Но иногда без них не обойтись, яркий пример — утилита **`passwd`** *(см. разд. 4.6.3)*.

Изменение атрибутов файла

Изменять атрибуты файла может либо его владелец, либо, конечно, суперпользователь `root`.

Примечание

Обратите внимание: для того чтобы так "хозяйничать", необходимо иметь право записи в каталог, содержащий этот файл.

Для выполнения изменений следует воспользоваться файловым менеджером (см. рис. 4.1) или утилитами командной строки **`chown`** (для

изменения владельца и группы), **chgrp** (для изменения группы) или **chmod** (для изменения атрибутов доступа).

4.6.2. Регистрация пользователя

Пользователь регистрируется в системе посредством утилиты **login** или **phlogin**. Получив имя и пароль, утилита (пускай это будет **login**), просматривает базу данных пользователей /etc/passwd в поисках указанного имени. Файл /etc/passwd состоит из строк следующего формата:

username:*haspw*:*userid*:*group*:*comment*:*homedir*:*shell*

где *username* — имя пользователя, используемое для входа в систему; *haspw* — если это поле не пустое, то в файле /etc/shadow хранится пароль пользователя; *userid* — идентификатор пользователя (для root равен 0); *group* — идентификатор группы; *comment* — любая строка, не содержащая символ ":"; *homedir* — домашний каталог пользователя, т. е. каталог, в котором пользователь может произвольно создавать и удалять файлы; *shell* — командный интерпретатор, который вызывает утилита **login** при успешном входе пользователя в систему.

Если введенное имя пользователя существует в базе данных, то проверяется значение поля *haspw*. Поле может быть пустым, что означает отсутствие пароля у пользователя, или не пустым. Если поле не пустое, то **login** просматривает базу данных паролей /etc/shadow, разыскивая строку, соответствующую имени пользователя. Файл /etc/shadow состоит из строк следующего формата:

username:*password*:*lastch*:*minch*:*maxch*:*warn*:*inact*:*expire*:*reserved*

где *username* — имя пользователя; *password* — зашифрованный пароль пользователя; *lastch* — время последней модификации; *minch* — минимальное количество дней для модификации; *maxch* — максимальное количество дней для модификации; *warn* — количество дней для предупреждения; *inact* — максимальное количество дней между входами в систему; *expire* — дата истечения доступа; *reserved* — поле зарезервировано для последующего использования.

Если имя и пароль введены правильно, то **login** запускает командный интерпретатор, указанный в файле /etc/passwd, с идентификаторами UID и EUID, равными идентификатору пользователя, указанному в файле /etc/passwd, и идентификаторами GID и EGID, равными идентификатору группы, указанному в файле /etc/passwd.

Все остальные процессы, запускаемые из этого интерпретатора, включая Photon, наследуют эти идентификаторы.

Следует добавить, что существует файл `/etc/group`, строки которого имеют формат:

`groupname``:`*`reserved`*`:`*`group`*`:`*`member`*

где *`groupname`* — имя группы; *`reserved`* — зарезервировано для дальнейшего использования; *`group`* — числовой идентификатор группы; *`member`* — список имен пользователей, принадлежащих к данной группе.

В список имен пользователей, принадлежащих к группе, можно добавить любого существующего пользователя.

> ***Примечание***
> Утилита **`login`** (но не **`phlogin`**!) создает в домашнем каталоге пользователя файл `.lastlogin`, содержащий дату и время последнего входа в систему.

4.6.3. Смена пароля и добавление пользователей и групп

Пользователь может изменить свой пароль, выполнив команду **`passwd`**. Утилита запросит прежний пароль и дважды предложит ввести новый пароль.

Если команду **`passwd`** выполнит пользователь `root`, при этом указав в качестве аргумента команды имя пользователя, то возможно два варианта:

- если пользователь с таким именем уже существует, то пользователь `root` получит предложение изменить его пароль;
- если пользователя с таким именем нет, то утилита предложит системному администратору ввести регистрационные данные пользователя для заполнения файла `/etc/passwd`. Если при вводе данных администратор задаст идентификатор группы, которой нет в файле `/etc/group`, то утилита выведет сообщение, чтобы пользователь `root` не забыл отредактировать файл `/etc/group`.

Добавление новых групп выполняется путем простого редактирования файла `/etc/group`.

При добавлении пользователей и изменении паролей старые версии файлов `/etc/passwd` и `/etc/shadow` сохраняются, соответственно, в файлах `/etc/opasswd` и `/etc/oshadow`.

Кстати, утилита **`passwd`** представляет из себя замечательный образец использования файлового атрибута Set User ID (SUID). Дело в том, что процесс **`passwd`** редактирует файл, записывать в который имеет право только пользователь `root`. То есть процесс **`passwd`** должен всегда иметь UID и EUID, равные 0. Это и достигается использованием атрибута SUID. Разумеется, предоставлять файлам атрибуты SUID и SGID нужно очень аккуратно, чтобы не сделать брешь в защите информации.

4.6.4. Удаление пользователей и групп

Для того чтобы удалить пользователя из системы, системный администратор должен отредактировать три файла, удалив из них строки, соответствующие пользователю: `/etc/passwd`, `/etc/shadow`, `/etc/group`.

Для удаления группы достаточно отредактировать файл `/etc/group`, но обязательно убедитесь, что ни для кого из существующих пользователей эта группа не является первичной (короче говоря, проверьте, чтобы в файле `/etc/passwd` эта группа не упоминалась).

4.6.5. Изменение атрибутов процесса

Для временного изменения UID и EUID предназначена утилита **`su`**, а для временного изменения GID и EGID — утилита **`newgrp`**. Для того чтобы можно было изменить идентификатор группы, необходимо, чтобы имя пользователя присутствовало в списке членов этой группы в файле `/etc/group`.

В природе еще встречается свободно распространяемый пакет с утилитой **`sudo`** — она представляет собой утилиту **`su`** с расширенной функциональностью.

Для того чтобы регистрировать использование утилиты **`su`** пользователями, необходимо создать файл журнала с именем, указанным в конфигурационном файле `/etc/default/su`. По умолчанию там определен файл `/usr/adm/sulog`. Это наследство QNX4, где не было каталога `/var` для хранения текущей изменяемой информации, как это принято в UNIX, а использовался каталог `/usr`. Поэтому лично я предпочитаю указать утилите **`su`** вести журнал в файле

/var/log/sulog. Не забудьте, что журнальный файл должен иметь атрибуты доступа на чтение и запись только для своего владельца — пользователя root.

Глава 5. Сетевая подсистема QNX

Когда вы на самом деле захотите организовать разнородную сеть слово "совместимость" станет для вас ругательным...
(Из какой-то книги о сетевых операционных системах)

В этой главе рассмотрены вопросы:

- структура сетевой подсистемы QNX;
- "родная" QNX-сеть — Qnet;
- поддержка TCP/IP в QNX;
- защита информации в сети.

ОС QNX Neutrino — система изначально сетевая, однако сетевые механизмы, как и все остальное, реализованы в виде дополнительных администраторов ресурсов. Хотя некоторая поддержка сети есть в микроядре — способ адресации QNX-сообщений обеспечивает возможность передачи их по сети. Грубо говоря, микроядру все равно, является сообщение сетевым или локальным, что дает QNX Neutrino мощный механизм поддержки сетевых кластеров.

В комплектах QNX Momentics без Net TDK содержатся средства поддержки двух протоколов — TCP/IP, являющегося промышленным стандартом, и Qnet — "родного" протокола QNX Neutrino, реализующего концепцию "прозрачной сети". Компьютеры, объединенные в сеть Qnet, фактически представляют собой виртуальную многопроцессорную суперЭВМ.

В состав Net TDK (Extended Networking Technology Development Kit), как мы уже сказали в *разд. 2.1.2*, входят разные известные, но нетрадиционные для операционных систем реального времени полезные средства — IPSec, NAT, SNMP и т. п.

5.1. Структура сетевой подсистемы QNX

В центре реализации сетевой подсистемы QNX Neutrino находится Администратор сетевого ввода/вывода **`io-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**.

Рис. 5.1. Фрагмент окна перспективы QNX System Information: некоторые данные о процессе **io-net**

В верхней части рис. 5.1 можно увидеть, что процесс **io-net** относится к администраторам ресурсов и что он создает пул серверных потоков, ожидающих клиентские запросы по каналу 1 (**Channel 1**). В нижней части того же рисунка видно, какие разделяемые библиотеки загружены процессом **io-net**:

- Администратор стека TCP/IP — **npm-tcpip.so**;
- Администратор протокола Qnet — **npm-qnet.so**;
- драйвер сетевых адаптеров AMD PCNET — **devn-pcnet.so**.

Примечания

1. Префикс **npm** у администраторов сетевых протоколов является аббревиатурой Network Protocol Manager.

2. Можно для получения этой же информации воспользоваться утилитой **sin**:

   ```
   sin –P io-net mem
   ```

   или, конечно, ее графическим аналогом — утилитой **psin**.

Следующая команда запускает поддержку сети с драйвером сетевого адаптера NE2000 и с поддержкой протоколов TCP/IP и Qnet:

```
io-net -dne2000 -ptcpip –pqnet
```

Или, например, можно выполнить следующие 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
```

Примечание

По умолчанию в инсталлированной с компакт-диска системе поддержка Qnet не запускается. Чтобы она все-таки запускалась, необходимо создать пустой файл `/etc/system/config/useqnet` — сценарий `/etc/system/sysinit` при наличии этого файла сам выполнит указанную команду **`mount`**.

4. Загрузить Администратор протокола TCP/IP:

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

Каждая из загружаемых DLL администраторов какого-либо ресурса (протокола, драйвера, фильтра и т. п.) регистрирует в каталоге `/dev/io-net` свой префикс. Посмотрим на содержимое каталога `/dev/io-net` (табл. 5.1).

Таблица 5.1. *Префиксы в каталоге* `/dev/io-net`

Префикс	**Владелец префикса**
`/dev/io-net/en0`	Драйвер Ethernet
`/dev/io-net/qnet_en`	Конвертор Qnet-Ethernet
`/dev/io-net/ip0`	Стек TCP/IP
`/dev/io-net/ip_en`	Конвертор IP-Ethernet

Вообще-то, администратору **`io-net`** глубоко безразлично, какие драйверы и администраторы протоколов загружать — он является банальным коммутатором. Администратор сетевого ввода/вывода **`io-net`** предоставляет единый интерфейс, позволяющий разделяемым библиотекам регистрировать свои услуги. Все DLL **`io-net`** относит к нескольким следующим типам.

- *Драйвер сетевого адаптера* (теоретически все равно какого — Ethernet, Token Ring или другого). Администратор сетевого ввода/вывода **`io-net`** сообщает драйверу, когда он может забрать данные из какого-то буфера для передачи в физическую среду, и принимает от драйвера уведомления о поступлении данных из физической среды в какой-то буфер.
- *Администратор сетевого протокола* (опять же, теоретически — все равно какого). Администратор сетевого ввода/вывода **`io-net`** сообщает администратору протокола, когда можно забрать данные из какого-то буфера для передачи в приложение, и принимает от

администратора протокола уведомления о готовности данных в каком-то буфере для отправки во внешний мир.

- *Фильтр*. Фильтр может быть *восходящего* или *нисходящего* типа. Фильтр сообщает свой тип и между какими модулями он хочет хозяйничать. Администратор **`io-net`** сообщает восходящему фильтру, когда можно забрать данные из какого-то буфера драйвера, и принимает от восходящего фильтра уведомления о готовности данных в каком-то буфере для считывания администратором протокола. Нисходящий фильтр работает, как не трудно догадаться, с точностью до наоборот. Пример восходящего фильтра — firewall. Классический пример нисходящего фильтра — NAT[1].
- *Конвертор* — модуль для инкапсуляции/деинкапсуляции данных при передаче их между уровнями. Например, для добавления к IP-пакету заголовка и завершителя Ethernet при отправке IP-пакета и удаления заголовка и завершителя Ethernet при получении Ethernet-кадра.

Отключить поддержку какого-либо протокола, выгрузив соответствующую DLL, в QNX Neutrino 6.3.0 нельзя. В версии 6.2.1 можно было выгрузить, например, **`npm-qnet.so`** командой:

```
umount /dev/io-net/qnet_en
```

Однако в версии QNX Neutrino 6.3.0 можно все-таки выгружать драйвер Ethernet:

```
umount /dev/io-net/en0
```

Для получения диагностической и статистической информации о работе сетевой карты предназначена утилита **`nicinfo`** (Network Interface Card Information). По умолчанию утилита **`nicinfo`** обращается к устройству `/dev/io-net/en0`. Для адаптеров Ethernet утилита **`nicinfo`** выводит наименование модели контроллера (в примере — RealTek 8139) и информацию, представленную в табл. 5.2.

---

[1] Network Address Translation (трансляция сетевых адресов) — механизм (и реализующие его программные средства), предназначенный для замены адресов источника и получателя в IP-пакетах по определенным правилам с целью, например, сокрытия от внешнего мира структуры локальной вычислительной сети, подключенной к Интернету.

Таблица 5.2. *Информация, выводимая утилитой* **`nicinfo`**

Наименование параметра	**Пример вывода**	**Значение**
Информация об адаптере		
Physical Node ID	`00005C 000218`	Физический адрес (MAC-адрес) адаптера
Current Physical Node ID	`00005C 000218`	MAC-адрес, присвоенный адаптеру (имеет смысл для адаптеров с программируемыми MAC-адресами)
Media Rate	`100.00 Mb/s full-duplex UTP`	Скорость и способ передачи данных
Mtu	`1514`	Максимальный размер кадра, байты
Lan	`0`	Логический номер сети (используется, если к ЭВМ подключено несколько сетевых адаптеров)
I/O Port Range	`0xC000 -> 0xC0FF`	Диапазон портов ввода/вывода
Hardware Interrupt	`0x5`	Номер прерывания
Promiscuous	`Disabled`	Включен ли режим приема всех кадров
Multicast	`Enabled`	Включен ли режим групповой передачи
Информация по кадрам		
Total Packets Txd OK	`1136`	Общее число удачно отправленных кадров
Total Packets Txd Bad	`0`	Общее число кадров, отправленных с ошибками
Total Packets Rxd OK	`1934`	Общее число удачно принятых кадров
Total Rx Errors	`0`	Общее число принятых кадров, содержащих

Наименование параметра	**Пример вывода**	**Значение**
		ошибки
Количество байтов в кадрах		
Total Bytes Txd	`163 907`	Сколько всего байт отправлено
Total Bytes Rxd	`846 826`	Сколько всего байт получено
Информация об ошибках		
Tx Collision Errors	`0`	Количество коллизий при передаче
Tx Collisions Errors (aborted)	`0`	Количество коллизий при передаче (с отменой передачи кадра)
Carrier Sense Lost on Tx	`0`	Количество "потерь несущей" при передаче
FIFO Underruns During Tx	`0`	Число попыток отправить данные из пустого буфера
Tx defered	`0`	Количество кадров, передача которых была отложена
Out of Window Collisions	`0`	Число поздних коллизий
FIFO Overruns During Rx	`0`	Количество переполнений приемного буфера
Alignment errors	`0`	Число ошибок выравнивания
CRC errors	`0`	Количество ошибок несовпадения контрольной суммы

Вспомним, что для разработки собственных драйверов сетевых карт для QNX Neutrino предназначен специальный программный пакет Network Driver Development Kit (Network DDK), включающий исходные тексты

нескольких драйверов и инструкцию по написанию аппаратно-зависимого кода.

5.2. "Родная" QNX-сеть — Qnet

Задача протокола Qnet – превратить сеть из машин под управлением QNX Neutrino в некое подобие единого распределенного компьютера. В составе дистрибутива поставляется две версии администраторов Qnet:

- **`npm-qnet-compat.so`** — полнофункциональная версия Qnet, полностью совместимая с версией Qnet, использовавшейся в QNX Neutrino версии 6.2.*x*;
- **`npm-qnet-l4_lite.so`** — "облегченная" и, соответственно, более шустрая версия Qnet, появившаяся в QNX Neutrino версии 6.3.0 (l4 означает Level 4 модели ISO OSI).

Все сценарии оперируют именем **`npm-qnet.so`**, являющимся символической ссылкой[1] на один из двух указанных модулей.

5.2.1. Сеть Qnet — виртуальная супер-ЭВМ

Знаменитая сетевая прозрачность QNX достигается тем, что администраторы Qnet, работающие на разных ЭВМ, организуют как бы мост между своими микроядрами. Микроядро, когда видит, что у адресата сообщения дескриптор узла (ND)[2] не равен нулю, "не задумываясь" отдает это сообщение Администратору Qnet. Администратор Qnet засылает это сообщение Администратору Qnet указанного узла. Администратор Qnet узла на стороне получателя отправляет сообщение собственно получателю.

В QNX Neutrino узел имеет фиксированное *имя*, а дескриптор ND (Node Descriptor) назначается ему способом, напоминающим назначение файлового дескриптора при открытии файла. Обратите внимание, что ND = 0 всегда обозначает локальный узел (ядро не будет передавать

---

[1] По умолчанию это ссылка на **`npm-qnet-l4_lite.so`**, поэтому для совместной работы ЭВМ под управлением QNX Neutrino версий 6.3.0 и младше нужно "пересослаться" на **`npm-qnet-compat.so`**.

[2] Узел (node) — ЭВМ, работающая под управлением QNX. Обратите внимание, что в QNX версий 4.*xx* каждый узел имел жестко заданный идентификатор NID (Node ID). NID и ND — совсем не одно и то же.

администратору сети сообщение, содержащее ND=0, а сразу направит это сообщение потоку-адресату).

Каким же образом, зная имя узла-адресата сообщения, можно определить ND? Есть два способа. *Первый способ* заключается в использовании функции *netmgr_strtond()*,определяющей ND по имени узла. Недостаток этого способа очевиден — он платформозависимый. Поэтому чаще используют *второй способ*, основанный на использовании пространства путевых имен Администратора процессов. При загрузке и инициализации Администратор протокола Qnet **`npm-qnet.so`** регистрирует не только символьное устройство `/dev/io-net/qnet_en`, но и префикс-каталог `/net`, в котором размещаются папки, имена которых совпадают с именами узлов сети. Эти папки-узлы соответствуют корневым каталогам одноименных узлов. То есть, 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 **`ls`** и т. п. Поэтому все то, что было сказано про взаимодействие между клиентскими приложениями и администраторами ресурсов, остается в силе и тогда, когда речь идет о взаимодействии в сети Qnet. Другими словами, мы просто открываем файл с путевым именем, начинающимся с `/net`, получаем файловый дескриптор и начинаем записывать или считывать информацию как ни в чем не бывало.

Да, скажете вы, все это замечательно, но чтобы послать Ethernet-кадр, нужно знать MAC-адрес получателя. В TCP/IP используется протокол ARP, а что использует Qnet? Отвечаю. Qnet по умолчанию использует протокол NDP (Node Discovery Protocol), очень похожий на ARP.

Можно, кстати, по аналогии с протоколом FLEET операционной системы QNX4 жестко задать пару *имя_узла*–MAC-адрес. Только файл будет называться не `/etc/config/netmap`, а `/etc/qnet_hosts`, и синтаксис записей в этом файле таков:

имя_узла.имя_домена *MAC-адрес1*[, *MAC-адрес2*]

Как видно из синтаксиса, для одного узла можно задать два MAC-адреса.

Для того что бы использовать эту схему, нужно задать Администратору протокола Qnet опцию **`resolve=file`**.

Поскольку в QNX Neutrino почти все построено на основе механизма сообщений, протокол Qnet, делающий механизм сообщений сетевым,

дает нам достаточно много возможностей. Например, Qnet позволяет запускать процессы на любом узле сети.

Рассмотрим простую, но полезную утилиту **on**. Эта утилита является своего рода расширением командного интерпретатора по запуску приложений. Синтаксис утилиты **on**:

on *опции команда*

При этом команда будет выполнена в соответствии с предписаниями, заданными посредством опций. Основные опции этой утилиты:

- **-n** ***узел*** — указывает имя узла, на котором должна быть выполнена команда. При этом в качестве файловой системы будет использована файловая система узла, с которого запущен процесс;
- **-f** ***узел*** — аналогична предыдущей опции, 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** — предписывает отсоединиться 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*****]** — можно задать значение приоритета и, если надо, дисциплину диспетчеризации.

Например, команда, выполненная на узле host4:

```
on –d –n host5 –p 30f mytest
```

запустит на процессоре узла с именем host5 программу **mytest**, находящуюся на узле host4. Приоритет запущенного процесса будет иметь значение 30 с дисциплиной диспетчеризации FIFO. После запуска программы **mytest** утилита **on** сразу завершит свою работу, и интерпретатор выдаст приглашение для ввода очередной команды. Вывод утилиты **mytest** будет напечатан в текущей консоли узла host4.

Можно выполнить такую команду:

```
on –n host5 –f host3 –t con2 mytest
```

Она запустит на процессоре узла **host5** программу **mytest**, находящуюся на узле host3, вывод направит на консоль /dev/con2 узла host3.

Если нужно направить поток вывода утилиты **mytest** на другой узел, то имя консоли нужно указать целиком. То есть команда:

```
on –n host5 –f host3 -t /net/node2/dev/con1 mytest
```

выполненная с консоли узла node4, запустит на процессоре узла host5 программу **mytest**, находящуюся на узле host3, а вывод направит на консоль /dev/con2 узла host2.

Здесь уместно сказать о том, что платформозависимые файлы (т. е. исполняемые файлы и библиотеки) целесообразно хранить в "платформозависимых" каталогах, например, файлы PowerPC BigEndian помещают в каталог /ppcbe. Это позволяет брать на удаленном узле файлы, предназначенные для выполнения на аппаратуре конкретного узла сети. Например, мы хотим на нашем бездисковом компьютере на базе процессора Hitachi SH4 (LittleEndian) запустить утилиту **ls**, расположенную на узле с именем alpha (и с процессором Pentium III). Для этого нужно выполнить команду:

```
/net/alpha/shle/usr/bin/ls –l /home
```

Этот механизм позволяет использовать без конфликтов дисковое пространство всех узлов сети Qnet независимо от аппаратуры, на которой работает QNX Neutrino. В сочетании с возможностями утилиты **on** администратор получает "виртуальную суперЭВМ". Платформозависимые файлы можно отыскать в каталоге /usr/qnx630/target/qnx6.

> ***Примечание***
> Но не забывайте про "ложку дегтя" — в сети Qnet нелегко обеспечить безопасность информации!

Информацию о доступных узлах сети Qnet можно посмотреть с помощью команды **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
```

Из приведенного фрагмента видно, что выводятся: имя узла, размер доступной оперативной памяти, количество процессоров, тактовая частота и модель процессора.

> ***Примечание***
> Некоторые утилиты QNX Neutrino (например, **sin** или **slay**) имеют специальную опцию **–n** ***имя_узла***, указывающую, к какому узлу сети применить действие утилиты.

Для получения диагностической и статистической информации о работе Qnet можно в любом текстовом редакторе посмотреть файл статистики `/proc/qnetstats`.

5.2.2. Глобальные имена процессов

В *главе 3* мы рассмотрели пример применения унаследованного от QNX4 способа регистрации префиксов, использующего функцию *name_attach()*. Этот механизм уникален для QNX (т. е. не определен POSIX), поэтому компания QSS рекомендует не злоупотреблять им. Правда, программистам, воспитанным на QNX4, да и мне, воспитанному на Sun Solaris, имена процессов нравятся из-за их простоты и удобства. Однако в QNX Neutrino версий до 6.2.1 включительно было серьезное ограничение — не поддерживались в чистом виде глобальные имена процессов, имевшиеся в QNX4 (см. описание Администратора глобальных имен **`nameloc`** в документации QNX4). Что значит "в чистом виде"? Дело в том, что глобальные имена процессов при желании, немного постаравшись, все-таки можно было реализовать на прикладном уровне. В версии 6.3.0 разработчики QNX Neutrino решили облегчить жизнь прикладным программистам и написали нужный код сами.

Глобальные имена процессов поддерживаются с помощью администратора **`gns`** (GNS — Global Name Service). Эта программа может быть запущена либо в серверном (опция **`-s`**), либо в клиентском (опция **`-c`**) режиме. При этом она регистрирует префикс `/dev/name/gns_server` или `/dev/name/gns_client` соответственно. Регистрация имени выполняется прикладной программой с помощью вышеупомянутой функции *name_attach()*. Для регистрации глобального префикса в аргументах этой функции задается флаг `NAME_FLAG_ATTACH_GLOBAL`. Префикс регистрируется или в каталоге `/dev/name/global`, или в каталоге `/dev/name/local` — в зависимости от флага. Далее все без изменений — клиент получает идентификатор соединения с помощью функции *name_open()*.

5.2.3. Технология Jump Gate

Для обеспечения прозрачности в графической оболочке Photon есть механизм, получивший название Jump Gate.

Технология Jamp Gate основана на использовании серверного процесса **`phrelay`**, передающего 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 использования процесса **phrelay** в TCP/IP обычно применяется программа **inetd**. В стандартном файле `/etc/inetd.conf` уже есть (в закомментированном виде) нужная запись, поэтому достаточно просто раскомментировать ее:

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

Проверьте, что файл `/etc/services` содержит строку (вообще говоря, она там *должна* быть):

```
phrelay        4868/tcp
```

Программы-клиенты кэшируют получаемую информацию, поэтому им достаточно получать данные только об изменениях "картинки". Окно работающей программы **Phditto** показано на рис. 5.2.

Рис. 5.2. Программа **Phditto** в действии

При удаленном подключении к Photon microGUI по умолчанию создается дополнительный *сеанс* (session). Чтобы подключиться к существующему сеансу, необходимо указать имя файла нужного сеанса (т. е. указать соответствующий префикс типа "Named Special Device"). Например, подключимся к текущему сеансу Photon узла `host1`:

```
phditto -n /dev/photon host1
```

Картинку Photon в другую графическую оболочку передает, используя протокол TCP/IP[1], серверная программа **phrelay**. Для доступа к процессу **phrelay** из Windows предназначена клиентская программа **Phindows**[2] (рис. 5.3).

Рис. 5.3. Доступ к процессу **phrelay** из ОС Windows

Сервер **phrelay** может подключить клиента как к существующему сеансу Photon (тогда клиент получит полный доступ к запущенным

---

[1] Можно использовать и последовательный порт, но это — очень медленное решение.

[2] Для доступа к процессу **phrelay** из графической оболочки X Window System предназначена утилита **phinx** (продукт "Photon in X" в репозитарии 3rd-Party Software).

приложениям), так и к отдельному, специально созданному сеансу (тогда пользователи будут работать независимо в разных окружениях). Запретить или разрешить подключение к своему сеансу Photon пользователь может, установив или сбросив флажок в окне утилиты **`phrelaycfg`** (элемент меню **Remote Access**).

5.3. Поддержка 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 v1.5;
- `/lib/dll/npm-tcpip-v6.so` — профессиональный стек протоколов TCP/IP, KAME-расширение стека NetBSD v1.5.

Как вы, наверное, догадались, **`npm-tcpip.so`** — это просто ссылка на один из указанных модулей.

Конфигурация сети TCP/IP хранится в файле `/etc/net.cfg`. Для того чтобы изменения, внесенные в этот файл, 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 (мы пользовались ею при начальном конфигурировании QNX после инсталляции). Например, для запуска **`io-net`** с поддержкой TCP/IP вручную можно поступить, скажем, так:

```
io-net –d rtl –p tcpip
netmanager
```

Для получения информации о TCP/IP в QNX содержится полный набор общепринятых UNIX-утилит. Вот основные из них:

- **`arp`** — выполнение ARP-запросов;
- **`ifconfig`** — настройка параметров IP-интерфейса;
- **`if_up`** — проверка доступности IP-интерфейса;
- **`netstat`** — отображение состояния и статистики IP-сети;

- **route** — настройка статической маршрутизации;
- **nslookup** — опрос службы доменных имен DNS;
- **showmount** — получение информации о состоянии сервера NFS;
- **ping** — эхо-запрос узла IP-сети пакетами ECHO_REQUEST протокола ICMP;
- **rpcinfo** — отправка RPC-запросов RPC-серверу и отображение полученной информации об RPC;
- **traceroute** — трассировка маршрута IP-пакетов.

5.4. Защита информации в сети

Для обеспечения защиты информации, обрабатываемой в вычислительной сети, могут применяться различные методов, хорошо описанные в специализированной литературе. В этой главе мы рассмотрим некоторые из них — лежащие, так сказать, "на поверхности". К таким методам относятся:

- защита "родных" сообщений QNX от преднамеренной или непреднамеренной модификации;
- построение виртуальных частных сетей — протокол IPSec;
- фильтрация IP-пакетов и трансляция IP-адресов;
- шифрование трафика при удаленном терминальном доступе и при обмене файлами — протокол SSH.

5.4.1. Защита QNX-сообщений от модификации

Бывает, что есть два или более серверов, к которым последовательно обращается некоторый клиент. У первого сервера, к примеру, клиент запрашивает некоторую информацию, которую затем передает второму серверу, а тот, в свою очередь, на основе предоставленной ему информации выдает какие-то данные. При этом может сложиться ситуация, когда нам не хотелось бы, чтобы клиент мог модифицировать информацию, полученную от первого сервера, прежде чем передаст ее второму серверу. То есть налицо случай передачи информации от одного доверенного субъекта к другому доверенному субъекту через недоверенного субъекта-посредника.

Для защиты передаваемых данных от модификации используют функцию *MsgKeyData()*. Идея заключается в следующем: первый сервер

на основе некоторого известного только ему целого числа (*закрытого ключа*), `rcvid` и текста сообщения, подготовленного к отправке, генерирует *открытый ключ*, который вместе с ответным сообщением (вернее, в составе ответного сообщения) отправляет клиенту. Клиент пересылает это сообщение второму серверу. Второй сервер с помощью открытого ключа, входящего в состав сообщения, вычисляет, было ли это сообщение модифицировано. Вообще-то, в электронной документации комплекта разработчика QNX Momentics в описании функции *MsgKeyData()* вся эта кухня с контролем модификации документирована достаточно подробно с весьма наглядным примером, поэтому направляю вас, уважаемый читатель, к документации.

Нелишне заметить, что как полнофункциональная, так и облегченная версии Администратора протокола Qnet **`npm-qnet.so`** поддерживают две полезные опции, позволяющие повысить защищенность информации от НСД в сети Qnet:

- **`mapany=`*`map_uid`*** — идентификаторы UID и GID любого пользователя, который пытается подключиться к узлу через Qnet, будут заменены на заданные идентификаторы ***`map_uid`***;
- **`maproot=`*`map_uid`*** — идентификаторы UID и GID суперпользователя, пытающегося подключиться к узлу через Qnet, будут заменены на заданные идентификаторы ***`map_uid`***.

5.4.2. Протокол IPSec

Для обеспечения безопасности в модуле профессионального стека TCP/IP используется протокол IPSec. Этот протокол включает две основные части — *аутентификационный заголовок* (AH, Authentication Header), позволяющий определять отправителя пакета, и *инкапсуляционный протокол безопасности* (ESP, Encapsulating Security Protocol), позволяющий шифровать содержимое пакетов. Оба эти компонента позволяют использовать IPSec как для защиты соединения точка–точка, так и для создания *туннелей*, т. е. так называемых *виртуальных частных сетей* (VPN, Virtual Private Networks). Использование защиты соединения точка–точка невозможно при трансляции сетевых адресов (о трансляции адресов см. далее). Туннели применяют для доступа извне в защищенную сеть.

5.4.3. Фильтрация пакетов

Для *пакетной фильтрации* (firewall) и трансляции сетевых адресов (NAT, Network Address Translation) используется перенесенный в QNX программный пакет IP Filter версии 3.4.31, основу которого составляет модуль **lsm-ipfilter-v6.so** или **lsm-ipfilter-v4.so**, "подгружаемый" администратором **npm-tcpip-v6.so** или **npm-tcpip-v4.so** соответственно. Удобно сделать символическую ссылку `ipfilter.so` на один из модулей в зависимости от используемой версии TCP/IP. Далее по тексту мы будем для краткости использовать имя символической ссылки. В состав IP Filter также входит ряд утилит:

- **ipf** — предназначена для изменения списка правил фильтрации;
- **ipfs** — сохраняет и восстанавливает информацию для NAT и таблицы состояний;
- **ipfstat** — возвращает статистику пакетного фильтра;
- **ipmon** — монитор для сохраненных в журнале пакетов;
- **ipnat** — пользовательский интерфейс для NAT.

Посмотреть версию и текущий статус IP Filter можно с помощью команды:

```
ipf -V
```

Для загрузки модуля **ipfilter.so** следует выполнить команду:

```
mount –Ttcpip ipfilter.so
```

Возможно, для загрузки IP Filter потребуется больший размер стека у потоков TCP/IP, чем это задано по умолчанию. Больший размер стека можно задать так:

```
io-net –drtl –ptcpip stacksize=3310
netmanager
```

На имевшейся в моем распоряжении тестовой ЭВМ 3310 байт — минимальный размер стека, при котором корректно подгружается IP Filter. Администратор сетевого ввода/вывода **io-net** загрузит динамическую библиотеку, в чем можно убедится, выполнив команду

```
ipf -V
```

или

```
pidin –P io-net mem
```

После загрузки модуля необходимо задать правила фильтрации пакетов и трансляции адресов. Для этой цели правила сначала следует записать

в файлы конфигурации, а затем с помощью соответствующих административных утилит указать модулю **ipfilter.so** прочитать эти файлы.

> ***Примечание***
> Файл конфигурации представляет собой набор строк-правил.

Рассмотрим, как настраивать фильтрацию пакетов. Перед тем как загружать правила, неплохо бы "сбросить" старые правила, например:

```
ipf -Fa
```

Если файл конфигурации назвать, скажем, `/etc/ipf.conf`, то команда загрузки правил фильтрации будет такой:

```
ipf -f /etc/ipf.conf
```

Можно пользоваться *комментариями* — правила такие же, как при написании командных сценариев, т. е. комментарий начинается с символа # и продолжается до конца строки. Правило в простейшем случае имеет следующий синтаксис:

Действие Направление Отправитель-Получатель

Действия могут быть двух типов:

- `block` — отвергнуть пакет;
- `pass` — пропустить пакет.

Направление может быть одним из двух:

- `in` — входящие пакеты;
- `out` — исходящие пакеты.

Отправитель-Получатель может задаваться по-разному:

- `any` — пакет от любого отправителя любому получателю;
- `from` *Отправитель* `to` *Получатель* — пакет от отправителя, соответствующего шаблону *Отправитель*, получателю, соответствующему шаблону *Получатель*; при этом каждый из шаблонов может иметь различные значения, например:
  - `any` — любой;
  - *A.B.C.D* — с фиксированным IP-адресом (вместо IP-адреса допускается задавать имя хоста);
  - *A.B.C.D/netmask* — хосты из определенных подсетей.

Например,

```
block in all # Запретить все входящие пакеты
pass in all   # Пропускать все входящие пакеты
```

Попрошу обратить внимание, что, получив пакет, IP Filter последовательно перебирает ВСЕ правила и применяет к пакету ПОСЛЕДНЕЕ из подходящих правил. То есть если заданы правила:

```
block in all
pass in all
```

то все входящие пакеты будут пропускаться. Для того чтобы IP Filter не просматривал все правила, следует задать ключевое слово `quick`. Например, правила

```
block in quick all
pass in all
```

не пропустят ни одного входящего пакета. Почему? Потому что любой входящий пакет соответствует правилу

```
block in quick all
```

и для этого правила задано ключевое слово `quick`, запрещающее дальнейший просмотр правил. Другими словами, `quick` указывает фильтру: "Если это правило подходит к данному пакету, то применить правило немедленно".

Кроме IP-адресов отправителя/получателя пакета в правиле можно задавать имя интерфейса, через который пакет передается. Напомню, что интерфейсы для контроллеров Ethernet обозначаются `en`*`N`*, где *`N`* — порядковый номер интерфейса:

```
block in quick on en0 any
```

В правилах допускается комбинировать IP-адреса и имена интерфейсов:

```
pass in quick on en0 from any to 111.222.0.0/16
```

При желании можно включить регистрацию пакетов, удовлетворяющих правилу. Для этой цели в правило следует добавить ключевое слово `log`:

```
pass in log quick on en0 from any to 111.222.0.0/16
```

Журналом для записи регистрационной информации является устройство `/dev/ipl`. Для чтения журнала можно воспользоваться утилитой **`ipmon`**. Можно пересылать регистрационную информацию в систему ведения журналов событий syslog UNIX. Для этого в каждом правиле можно задать уровень важности регистрируемого события, используя дополнительное ключевое слово `level` с парой параметров *`источник.уровень_серьезности`*. Оба параметра хорошо известны читателям, знакомым с syslog. В качестве источника логично задать `auth` (не объясняя, почему, направлю вас к документации по syslog), а в

качестве уровня серьезности события задается одно из значений (в порядке убывания этой самой серьезности):

- `emerg`
- `alert`
- `crit`
- `err`
- `warning`
- `notice`
- `info`
- `debug`

Нарпимер:

```
pass in log level auth.info quick on en0 from any to 1.2.3.4
```

Не забудьте программе **`ipmon`** задать опцию **`-s`**, ну и настроить syslog, разумеется.

Фильтрация пакетов по IP-адресам отправителя и получателя дело хорошее, но само по себе достаточно грубое. Для управления пакетами, принадлежащим определенным протоколам, используется ключевое слово `proto`:

```
pass in quick on en0 proto tcp from any to 111.222.0.0/16
pass in quick on en0 proto udp from any to 111.222.0.0/16
pass in quick on en0 proto tcp/udp from any to 111.222.0.0/16
```

При фильтрации пакетов и дэйтаграмм TCP и UDP не обойтись, конечно же, без ключевого слова `port`. Порты задаются как в виде определенного значения, так и в виде диапазона, например

```
pass in quick proto tcp from any to 1.2.3.4 port=50
pass in quick proto udp from any to 1.2.3.4 port 1025 <> 2050
```

Номера портов, используемые стандартными TCP- и UDP-сервисами ("хорошо известные порты"), можно посмотреть в файле-справочнике `/etc/services`.

Что касается протокола ICMP, то нам, вероятнее всего, потребуется пропускать только пакеты, генерируемые утилитами **`ping`** и **`traceroute`**. Разблокировать прохождение пакетов ICMP можно так:

```
pass in quick proto icmp from any to 1.2.3.4
```

Теперь посмотрим, как можно пользоваться этими правилами. Если мы хотим запретить доступ извне ко всем сервисам нашей сети, кроме Telnet, то можно записать так:

```
block in all
pass in quick on en0 proto tcp from any to 1.2.3.4 port = 23
```

Еще IP Filter умеет посылать ответы клиентам, приславшим блокируемые пакеты. Например, при использовании протокола TCP можно послать клиенту ответный пакет, содержащий флаг RST (Reset):

```
block return-rst in proto tcp from any to 1.2.3.4 port = 23
```

> ***ВНИМАНИЕ!***
> Эта книжка — обзор возможностей операционной системы QNX Neutrino, а не учебник по защите информации, поэтому без изучения специализированной литературы вам не обойтись.

Существует ряд хорошо известных приемов взлома — "подмена собой" одного из участников обмена информацией, DoS-атаки[1]. Методы борьбы с этими и иными способами вредительства также известны и поддерживаются различными межсетевыми экранами (firewalls), в том числе, в том числе и IP Filter. Для желающих подробнее изучить эту кухню предлагаю ссылку на сайт разработчика IP Filter — **http://coombs.anu.edu.au/~avalon/ip-filter.htm**.

5.4.4. Трансляция сетевых адресов

Трансляция сетевых адресов (NAT, Network Address Translation) позволяет создавать так называемые *прокси-серверы*, т. е. ЭВМ, за которыми "прячутся" другие ЭВМ. При этом извне "виден" только прокси-сервер, что позволяет скрывать состав и конфигурацию внутренних сетей организации. Задача прокси-сервера — подменять IP-адреса источника пакета, исходящего из внутренней сети, другим IP-адресом. Ответные сообщения, разумеется, будут приходить на прокси-сервер. В этом случае он должен корректно пересылать эти пакеты реальному получателю во внутреннюю сеть.

---

[1] Самый известный, наверное, вариант DoS-атаки — это "зафлуживание", т. е. массированная посылка какой-либо ЭВМ запросов, требующих обработки. В результате ЭВМ-жертва "захлебывается" и не способна обрабатывать реальные запросы.

Примечание

Надо сказать, что IP Filter реализует не только трансляцию IP-адресов, но и трансляцию номеров портов.

Для использования QNX Neutrino в качестве межсетевого экрана необходимо разрешить передачу пакетов с одного сетевого интерфейса на другой (forwarding). Для этого при загрузке Администратора протокола TCP/IP нужно указать опцию **`forward`** или **`forward6`**.

Для активизации трансляции необходимо, по аналогии с настройкой пакетной фильтрации, задать в некоем файле соответствующие правила и указать модулю **`ipfilter.so`** прочитать этот файл.

Если мы задали файлу конфигурации имя `/etc/ipnat.conf`, то команда "сброса" старых правил трансляции и загрузки новых будет такой:

```
ipnat -CF -f /etc/ipnat.conf
```

Правило трансляции в общем случае имеет следующий синтаксис:

`map` *`Интерфейс Отправитель-Получатель`* `->` *`Маска Порты`*

`Интерфейс` — это тот сетевой интерфейс, для которого мы, собственно, задаем правила трансляции, например — `en0`;

`Отправитель-Получатель` — задается по аналогии с правилами фильтрации пакетов;

`Маска` — IP-маска или диапазон IP-адресов, из которого берется подставляемый IP-адрес;

`Порты` — правило трансляции номеров портов (указывать необязательно), содержит ключевое слово `portmap`, имя протокола (TCP, UDP или оба сразу) и диапазон номеров портов. Вместо диапазона можно задать ключевое слово `auto`, тогда IP Filter возьмет ближайший свободный непривелигированный (т. е. с номером не меньше 1024) порт.

Заметим, что есть два особых IP-адреса:

- `0/0` — если задан этот адрес, то подмена адресов выполняться не будет;
- `0/32` — вместо этого адреса подставляется реальный адрес интерфейса.

Рассмотрим несколько правил, управляющих трансляцией сетевых адресов и портов.

- Заменять IP-адреса внутренней сети на адрес 1.2.3.4:

```
map en0 192.168.1.0/24 -> 1.2.3.4/32
```

- Заменять IP-адреса внутренней сети на адрес интерфейса `en0` (такой способ удобен, если, например, IP-адрес внешнего интерфейса задается динамически с помощью протокола DHCP):

  ```
  map en0 192.168.1.0/24 -> 0/32
  ```

- То же самое, но заменять номера портов TCP и UDP на номер из диапазона от 30 000 до 32 000:

```
map en0 192.168.1.0/24 -> 0/32 portmap tcp/udp 30000:32000
map en0 192.168.1.0/24 -> 0/32
```

В приведенном здесь примере сразу два правила задано не по ошибке — первое правило будет распространяться ТОЛЬКО на протоколы TCP и UDP, поэтому для остальных протоколов (например, SNMP) также следует задать правило трансляции адресов.

Допускается задавать для подстановки не один адрес, а пул адресов, , например с помощью адреса подсети и ее маски. В этом случает IP Filter будет сам выбирать адрес для отображения на него реальных адресов отправителей пакетов. Однако здесь имеется тонкость: есть приложения, которые устанавливают несколько соединений через несколько портов, при этом некоторые приложения требуют, чтобы IP-адрес удаленного хоста для всех соединений был одним и тем же. Чтобы IP Filter для всех пакетов сеанса подставлял один и тот же IP-адрес отправителя, необходимо использовать вместо ключевого слова `map` ключевое слово `map-block`:

```
map-block en0 from 192.168.1.0/24 -> 1.2.3.0/24 ports 32
```

С помощью ключевого слова `ports` задается количество портов, которое можно использовать для каждого IP-адреса.

Можно задавать отдельные правила трансляции для пакетов не только по адресу отправителя, но и по адресу получателя:

```
map en0 from 192.168.1.0/24 to 3.4.5.0/24 -> 1.2.3.4/32
```

Или даже отдельное правило в зависимости от номеров порта отправителя и/или получателя:

```
map en0 from 192.168.1.0/24 port 2000 \
        to 3.4.5.0/24 -> 1.2.3.4/32
```

IP Fiter позволяет перенаправлять[1] пакеты. Например:

```
rdr en0 1.2.3.4/24 port 80 -> 192.168.1.1/24 port 8080
```

Это правило действует так. Любой пакет, пришедший извне на адрес 1.2.3.4/24 и порт 80, будет реально направляться на адрес 192.168.1.1/24 и порт 8080. При этом ответные пакеты будут посылаться с адреса 1.2.3.4 и порта 80.

Примечания

1. По умолчанию правило переадресации распространяется только на протокол TCP. Для перенаправления UDP-дэйтаграмм в конце правила следует указать ключевое слово udp.

2. Правила перенаправления лучше задавать ПЕРЕД правилами фильтрации пакетов, иначе можно перехитрить самих себя.

5.4.5. Протокол SSH

Протокол SSH (Secure Shell) предназначен для защиты информации в открытых сетях и используется в качестве замены таких ранее популярных средств удаленного доступа, как **`rlogin`**, **`rsh`**, **`telnet`**. В основе защитных свойств SSH — аутентификация с использованием открытых/закрытых ключей и шифрование передаваемых данных. При этом с точки зрения пользователя ничего не меняется — как аутентификация, так и шифрование трафика выполняются автоматически и прозрачно. Есть другое важное свойство SSH — через шифрованный канал можно организовать обмен данными для других TCP-сервисов.

В операционную систему QNX Neutrino перенесен пакет OpenSSH. Это свободно-распространяемый продукт, реализующий непатентованные механизмы SSH (в основном это касается алгоритмов шифрования). В дальнейшем речь идет именно об OpenSSH.

Пакет OpenSSH включает ряд программ, к которым относятся:

- **`sshd`** — программа-сервер, основные задачи которого — прием запросов от программ-клиентов, обмен ключами, шифрование, аутентификация, исполнение команд и обмен данными. Для каждого соединения порождается своя копия процесса **`sshd`**. Сервер **`sshd`**

---

[1] По идее пакеты должны перенаправляться куда угодно. Однако мне удалось "перебрасывать" пакеты между портами и интерфейсами только в пределах ЭВМ с IP Filter.

в паре с программой-клиентом **ssh** реализует защищенный вариант программ **rlogin** и **rsh**;

- **ssh** — программа-клиент, позволяющая выполнять регистрацию и выполнение команд на удаленной ЭВМ;
- **scp** — защищенный вариант команды копирования файлов с одной машины на другую **rcp**. Использует программу **ssh**;
- **ssh-keygen** — программа для генерирования открытых ключей авторизации RSA или DSA для программы **ssh**. Программа **ssh-keygen** позволяет также выполнять ряд операций над ключами;
- **ssh-agent** — агент авторизации, хранящий в памяти закрытые ключи RSA или DSA, используемые для авторизации с помощью открытых ключей. По сути, эта программа является сервером для локальных программ, выполняющих авторизацию посредством механизма открытых/закрытых ключей. Закрытые ключи добавляются только с помощью программы **ssh-add**;
- **ssh-add** — программа добавления закрытых ключей агенту авторизации **ssh-agent** (номер порта, который слушает агент, должен храниться в переменной окружения SSH_AUTH_SOCK);
- **sftp-server** — программа-сервер, расширяющая **sshd** для поддержки защищенного протокола FTP (SFTP, Secure FTP);
- **sftp** — программа-клиент защищенной передачи файлов SFTP. Использует **ssh** и **ssh-keygen**;
- **ssh-keyscan** — программа для перехвата открытых ключей SSH, передаваемых по сети.

Теперь коротко поговорим о том, как все это работает.

Есть ключи — сервера **sshd** (генерируется сервером при старте и меняется с заданной периодичностью), хостов и пользователей.

Шифрование данных начинается до выполнения аутентификации. Сначала клиент соединяется с сервером **sshd** и получает от него открытые ключи сервера **sshd** и серверного хоста, затем клиент проверяет ключ серверного хоста, генерирует случайное число, шифрует его полученными ключами и посылает серверу. Затем обе стороны используют это случайное число в качестве ключа сеанса, применяемого во всех последующих передачах данных в этом сеансе. Только после этого выполняется аутентификация клиента и начинается

собственно обмен данными по одному из алгоритмов, поддерживаемых и клиентом, и сервером.

Для использования OpenSSH в качестве сервера сначала нужно после инсталляции запустить скрипт `/etc/openssh/genhostkeys.sh`, затем уже можно запускать сервер `/opt/sbin/sshd`. Использовать OpenSSH в качестве клиента еще проще — запускаете клиентскую утилиту **`ssh`** или **`sftp`** с нужными параметрами.

Ну и, конечно, вот ссылка для изучения дополнительного материала — **http://www.openssh.com**.

Глава 6. Построение и профилирование целевых систем QNX Neutrino

Точка, точка, запятая — вышла рожица кривая.
Палка, палка, огуречик — получился человечек.
А еще добавим ножек — получился осьминожек.
(Любимая считалочка разработчика целевых систем Neutrino)

QNX Neutrino — встраиваемая операционная система. То есть архитектура ОС позволяет разработчикам ПО с помощью специального инструментария формировать специализированные дистрибутивы Neutrino, "заточенные" для решения конкретной прикладной задачи в рамках ограниченных ресурсов. Как создаются встраиваемые конфигурации Neutrino и как можно протестировать их поведение при решении критических задач, — это мы и обсудим в данной главе.

Здесь мы рассмотрим следующие темы:

- процесс начальной загрузки;
- построение загружаемого образа QNX Neutrino;
- загрузка образа на целевой ЭВМ;
- системное профилирование.

6.1. Процесс начальной загрузки

Для начала вспомним, как выполняется процесс начальной загрузки QNX Neutrino.

После включения питания мультизагрузчик сначала предлагает выбрать, какую ОС следует загрузить. Вернее, он предлагает выбрать раздел диска, с которого следует выполнять загрузку. На моей машине по умолчанию грузится QNX, это выглядит так:

```
Press F1-F4 for select drive or select partition 1,2,3,4? 1
```

(Нажмите клавишу <F1>, <F2>, <F3> или <F4> для выбора дисковода или выберите раздел 1, 2, 3 или 4? 1)

После выбора раздела QNX начинается процесс начальной загрузки ОС. Начальная загрузка QNX протекает поэтапно:

1. Выполняется *код начального загрузчика* (IPL, Initial Program Loader), который осуществляет минимальное конфигурирование аппаратуры (конфигурирует контроллер памяти, системные часы и т. п.) и затем выполняет поиск на носителе и загрузку в ОЗУ образа ОС и передачу ему управления. IPL бывают:

   - "Warm-start" — для систем с BIOS (на момент старта IPL часть оборудования уже проинициализирована BIOS);
   - "Cold-start" — для систем без BIOS (всю инициализацию оборудования должен выполнять сам IPL).

   Последнее действие IPL — запуск модуля **`startup`**.

2. Выполняется код загрузчика ОС (**`startup`**), входящий 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`**, всегда имеющемуся в составе загрузочного образа QNX.

3. Модуль **`procnto`** выплняет необходимую инициализацию и запускает так называемый *стартовый сценарий*, входящий в состав загрузочного образа.

4. Стартовый сценарий (назовем его `Boot Script`) запускает с необходимыми параметрами программы и сценарии, входящие в состав образа. Без этого система делать ничего не будет — модуль **`procnto`**, как вы помните, лишь обеспечивает выполнение других программ.

Следует заметить, что в QNX обычно используется два загрузочных образа: основной образ помещается в файл `/.boot`, а резервный — в файл `/.altboot`. Вторичный загрузчик **`startup`** предлагает нажать клавишу <Esc> для загрузки резервного образа:

```
Hit Esc for .altboot
```

Если ничего не нажимать, то загружаться будет основной образ `/.boot`. Одним из процессов, входящих в стандартный образ, является **`diskboot`**, который запускается из стартового сценария. Основное назначение этого процесса — поиск файловой системы QNX на всех

доступных дисках для монтирования и запуск командного сценария `/etc/system/sysinit`.

> ***Примечание***
> Раздел диска с файловой системой QNX может быть смонтирован как корневой, только если в нем есть файл `/.diskroot`.

Конечно, на настоящей целевой системе использовать **`diskboot`** ни к чему — мы сами знаем, какие драйверы запускать и какие разделы диска куда монтировать. Следовательно, мы можем написать свой стартовый сценарий, который гораздо быстрее загрузит ЭВМ, чем слишком умная утилита **`diskboot`**.

Последним действием утилиты **`diskboot`** является вызов сценария `/etc/system/sysinit`, запускающего ряд других сценариев из каталога `/etc/rc.d/`. В конце "стандартная" цепочка сценариев запускает утилиту инициализации терминала **`tinit`**, указав ей запустить утилиту **`login`** или, при отсутствии файла `/etc/system/config/nophoton`, графическую оболочку Photon.

> ***Примечание***
> В файле конфигурации `/etc/config/ttys` можно указать, чтобы утилита **`tinit`** запускала на том или ином терминале любую другую программу или сценарий.

Photon при необходимости запускает утилиту **`phlogin`**. Как, кстати, Photon определяет, надо запускать **`phlogin`** или нет? Очень просто — по наличию непустой переменной окружения `LOGNAME`. Дело в том, что если идентификация и аутентификация пользователя уже выполнены утилитой **`login`**, то переменная `LOGNAME` уже существует и ее значение равно имени зарегистрировавшегося пользователя. Отсюда мораль — для быстрого запуска графического интерфейса без регистрации нужно в сценариях задать требуемое значение `LOGNAME` и потом запустить Photon.

> ***Примечание***
> Кстати, переменные окружения удобно задавать через опции утилиты **`tinit`**.

В любом случае стандартная схема потребует ввести имя и пароль. Что ж, как мы видим, в своей целевой системе можно легко сделать что-нибудь подобное или, напротив, совсем непохожее.

Таким образом, для загрузки QNX Neutrino нам нужны:

- IPL — "живет" отдельно, выбирается из имеющихся в наличии или разрабатывается на основе исходных текстов, входящих в BSP;
- **`startup`**-модуль — встраивается в загружаемый образ, выбирается из имеющихся в наличии или разрабатывается на основе исходных текстов, входящих в BSP;
- **`procnto`** — встраивается в загружаемый образ, выбирается из имеющихся в наличии;
- `Boot Script` — пишется самостоятельно;
- собственно загружаемый образ, включающий **`startup`**, **`procnto`**, `Boot Script` и другие необходимые файлы.

6.2. Построение загрузочного образа QNX Neutrino

Построение загрузочного образа — это то, с чем сталкивается разработчик, формируя конфигурацию QNX Neutrino с заданными параметрами. В какой-то степени построение загрузочного образа можно рассматривать как гораздо менее сложную и более безопасную (с точки зрения, например, надежности) альтернативу процедуры перекомпиляции ядра в монолитных операционных системах.

6.2.1. Общие сведения

Итак, последнее действие IPL — запуск модуля **`startup`**. Главный результат работы **`startup`**-модуля — загружен в память образ и управление передано модулю **`procnto`**. Что же представляет из себя загружаемый образ?

Вообще, в QNX Neutrino образы бывают двух типов:

- загружаемый образ ОС;
- образ встроенной файловой системы (файловой системы Flash).

Загружаемый образ ОС — это файл, содержащий модуль **`procnto`** приложения и некоторые данные, необходимые для функционирования целевой системы. Образ ОС можно рассматривать как "маленькую файловую систему", т. к. в образе есть простое дерево каталогов, из которого **`procnto`** берет информацию о содержащихся в образе файлах, их месте размещения в образе (эта файловая система получила название Image, мы подробнее обсудим ее в этой главе ниже).

С точки зрения работающей системы, образ ОС можно рассматривать как файловую систему с доступом по чтению. Содержимое образа можно посмотреть с помощью утилиты **dumpifs**. Посмотрим содержимое образа на нашей инструментальной машине:

```
dumpifs /.boot
```

Как образ ОС, так и образ файловой системы Flash могут содержать сжатые данные — при создании образа эти данные компрессируются (сжимаются), при последующих операциях чтения — декомпрессируются (разжимаются) автоматически. Следует отметить, что использование механизма компрессии/декомпрессии замедляет доступ к данным и создает дополнительную нагрузку на CPU целевой системы. Поэтому использовать компрессию рекомендуется только для систем с действительно ограниченным объемом носителя данных (Flash, ROM).

Образ ОС создается программой **mkifs** (MaKe Image FileSystem). Данные для своей работы эта утилита берет из командной строки и из *файла построения*, или *build-файла* (buildfile).

Для того чтобы создать загрузочный образ, следует написать в текстовом редакторе или сгенерировать с помощью IDE-перспективы QNX System Builder три части build-файла:

- секцию `boot`, определяющую параметры модулей **startup** и **procnto**;
- сценарий `Boot Script`;
- остальную часть, определяющую содержимое файловой системы Image.

Чтобы не изобретать велосипед, рассмотрим build-файл нашей системы разработки — `/boot/build/qnxbasedma.build`. Основные системные элементы для образа находятся в каталоге `/boot/sys`.

6.2.2. Секция *boot*

Рассмотрим секцию `boot` (комментарии удалены):

```
[virtual=x86,bios +compress] boot = {
   startup-bios -s64k
   PATH=/proc/boot:/bin:/usr/bin
     LD_LIBRARY_PATH=/proc/boot:/lib:/usr/lib:/lib/dll
     procnto-instr
}
```

Означает это следующее:

- `virtual=x86` — собираем образ для *x*86-совместимой целевой системы с полной защитой памяти (для *x*86 других вариантов, собственно, нет);
- `+compress` — использовать сжатие образа. Этот флаг уменьшает размер образа, но увеличивает время начальной загрузки (**`startup`**-модулю придется разжимать образ);
- `startup-bios -s64k` — определяем, какой из **`startup`**-модулей будем использовать, задаем ему опции запуска;
- `PATH=`*`пути_к_программам`* `LD_LIBRARY_PATH=`*`пути_к_DLL`* `procnto-instr` (одной строкой!) — указываем, какой из модулей **`procnto`** включаем в образ. В данном случае опции не указаны, но их можно задавать. Модуль **`procnto`** будет запущен с указанными значениями переменных окружения `PATH` и `LD_LIBRARY_PATH`.

6.2.3. Файловая система Image

За секцией `boot` следует описание файловой системы Image. Утилита **`mkifs`** берет указанные файлы из системы разработчика, однако содержимое текстовых файлов можно явно определить прямо в build-файле. Яркий пример — файл сценария `Boot Script` (про него поговорим отдельно). Итак, посмотрим снова на наш build-файл. В нем перечислены программные модули:

```
libc.so
libcam.so
io-blk.so
cam-disk.so
fs-qnx4.so
fs-dos.so
fs-ext2.so
cam-cdrom.so
fs-cd.so
[data=uip]
seedres
pci-bios
devb-eide
devb-amd
devb-aha2
```

```
devb-aha4
devb-aha7
devb-aha8
devb-ncr8
diskboot
slogger
fesh
devc-con
```

Другими словами, в этой части перечисляем все, что собираемся включать в состав образа. Советую не увлекаться и обратить внимание на то, что в рассматриваемом примере бо̲льшая часть модулей образа — драйверы устройств и администраторы файловых систем.

Видно, что перед перечислением исполняемых модулей указан атрибут `[data=uip]`. Дело в том, что поскольку файловая система Image после загрузки находится непосредственно в ОЗУ, при запуске приложений можно поступить двояко: использовать уже загруженный код непосредственно из того места ОЗУ, где он находится (`uip` — Use In Place), либо сделать, так сказать, рабочую копию (`copy`). Причем поведение можно определить отдельно для сегмента кода (`code`) и для сегмента данных (`data`).

Все заданные файлы после загрузки образа на целевой системе по умолчанию помещаются в каталог `/proc/boot`. Если нужно какой-то файл поместить в другое место, то следует указать полное путевое имя этого файла таким, каким оно должно быть на целевой системе.

После перечисления файлов в нашем примере есть такой фрагмент:

```
unlink_list={
/proc/boot/devb-*
}
```

Это пример наглядно показывает, как можно включать в образ произвольные текстовые файлы. Имя файла — `unlink_list`, между фигурными скобками — содержимое. Этот текстовый файл, как легко догадаться, на целевой системе будет называться `/proc/boot/unlink_list`.

> ***Примечание***
>
> Файл `unlink_list` предназначен только для информационных целей. Читатель, конечно, понял, что драйверы в том количестве, в каком они включены в образ, реально не требуются. Поэтому программа **`diskboot`** удаляет драйверы из ОЗУ после монтирования дисковой файловой системы. Операция удаления жестко задана в программе

diskboot, а чтобы пользователь знал о том, *что* удалено, разработчики добавили в образ файл `unlink_list`.

6.2.4. Сценарий *Boot Script*

От написания сценариев не освобождает даже QNX IDE, поэтому рассмотрим их чуть подробнее. У сценария `Boot Script` есть полезные свойства:

- он выполняется после того, как Администратор процессов завершил свою инициализацию;
- на момент запуска сценария на целевой системе в нашем распоряжении будет файловая система Image со всеми компонентами, которые мы задали;
- из него можно запускать другие сценарии, содержащиеся в образе.

Посмотрим на сценарий из нашего примера:

```
[+script] startup-script = {
  procmgr_symlink ../../proc/boot/libc.so.2
    /usr/lib/ldqnx.so.2
  [pri=10o] PATH=/proc/boot diskboot -b1 -D1 -odevc-con,-n4
}
```

Сценарий назвается `startup-script`, но это, строго говоря, неважно — утилита **mkifs** узнает его по атрибуту `[+script]`. В среде разработки, как мы видим, сценарий очень прост: сначала создается ссылка `/usr/lib/ldqnx.so.2` на файл `/proc/boot/libc.so.2`, затем запускается "умная" утилита **diskboot**, которая доводит загрузку до победного конца.

> ***Примечание***
> Мы уже говорили, что загружать целевые системы утилитой **diskboot** можно, но нерационально. Эта утилита тратит много времни на всевозможные проверки и сканирование и создана специально для загрузки ЭВМ с неизвестным набором аппаратуры, поддерживаемой в QNX. По сути дела, в своей сетевой системе нам нужно вручную получить тот же результат, что и у **diskboot**. Но это на самом деле просто — спецификация целевой ЭВМ вам известна.

Конечно, возможностей у нас меньше, чем при использовании Korn Shell, т. к. не будем же мы помещать в образ разные утилиты — ОЗУ не безразмерно, — но, тем не менее, есть несколько встроенных команд, которые всегда можно использовать в стартовом сценарии:

- `display_msg` — выводит указанный текст, например:

```
display_msg Hello!
```

- `procmgr_symlink` — создаёт "виртуальную" символьную ссылку аналогично команде **`ln -P`**, за исключением того, что не требуется утилита **`ln`** (это мы уже видели в примере).
- `reopen` — связывает потоки **`stdin`**, **`stdout`** и **`stderr`** с указанным файлом. Например:

```
reopen /dev/con1
```

- `waitfor` — ожидает момента, когда вызов функции *stat()* по указанному путевому имени завершится успешно (т. е. пока не будет зарегистрирован соответствующий префикс). Например:

```
waitfor /dev/pci
```

Перед командами сценария можно помещать атрибуты. Есть два типа атрибутов:

- логический атрибут (Boolean) ("`+`" — `true`, "`-`" — `false`). Например:

```
[+session] ksh
```

  (процесс **`ksh`** будет лидером сеанса).

- атрибут-значение (Value). Например:

```
[pri=27f] ksh
```

  (процесс **`ksh`** будет запущен с приоритетом 27 и дисциплиной диспетчеризации FIFO).

Атрибуты можно комбинировать:

```
[+session pri=27f] ksh
```

> ***Примечание***
> Обратите внимание, что все скомбинированные атрибуты помещают в одни квадратные скобки.

Если необходимо использовать программы с такими же именами, как у встроенных команд, то для этих программ следует задавать аттрибут `[+external]`. Например, запустим гипотетическую утилиту **`reopen`** с опцией **`-f`**:

```
[+external] reopen –f
```

Для наглядности посмотрим на фрагмент настоящего сценария, более содержательного, чем используемый утилитой **`diskboot`**:

```
display_msg Starting serial driver
```

```
devc-ser8250 -e -b115200 &
waitfor /dev/ser1 # don't continue until /dev/ser1 exists

display_msg Starting pseudo-tty driver
devc-pty &

display_msg Setting up consoles
devc-con &
reopen /dev/con2  # set stdin, stdout and stderr to /dev/con2
    [+session pri=27r] PATH=/proc/boot fesh &
reopen /dev/con1  # set stdin, stdout and stderr to /dev/con1
    [+session pri=10r] PATH=/proc/boot fesh &
```

Для выполнения такого сценария в образе, само собой, должны быть модули **`devc-ser8250`**, **`devc-pty`**, **`devc-con`** и **`fesh`**.

6.2.5. Перспектива QNX System Builder

Внешний вид окна **QNX System Builder** одноименной перспективы в последней имеющейся у меня версии IDE показан на рис. 6.1.

Рис. 6.1. Окно перспективы QNX System Builder

Перспектива QNX System Bulder предоставляет такие возможности:

- конфигурирование секции `boot` в графическом виде;
- добавление/удаление модулей в образ в графическом режиме (при этом мы сразу видим, как будет выглядеть файловая система Image на целевой системе). Разделяемая библиотека `libc.so` добавляется автоматически с установлением символической ссылки на нее `/usr/lib/ldqnx.so`;
- мастер оптимизации System Optimizer (для его запуска надо нажать кнопку). Он может выполнять три вида оптимизации:
  - удалить неиспользуемые библиотеки, включенные нами в образ;
  - добавить необходимые библиотеки, которые мы забыли включить в образ;

- удалить из библиотек для сокращения размеров образа те функции, которые не используются (другие инструменты этого не умеют);

- информация о каждом элементе образа отображается в окне свойств **Properties** и в окне атрибутов **Item Attributes**.

Сгенерированный IDE образ имеет расширение `ifs` и помещается в каталог `$HOME/workspace/`*`имя_проекта`*`/Images`.

6.3. Загрузка образа на целевой ЭВМ

Теперь нужно как-то загрузить полученный образ на целевой ЭВМ. Для этого есть несколько способов.

- Скопировать образ в файл `/.boot` или `/.altboot` загружаемого раздела QNX4 на целевой системе. Это самый простой способ (конечно, при наличии диска у целевой ЭВМ).Поместить образ в ПЗУ целевой системы. Обычно для этой цели используют флэш-память. Такое решение — часто единственно возможное для целевых систем, работающих в жестких условиях эксплуатации (например, при тряске) и требующих очень быстрой загрузки операционной системы.

- Загрузить образ по сети с TFTP-сервера. Для этого на сетевой карте должна быть микросхема ПЗУ с расширением BIOS, поддерживающим протоколы Internet Boot Protocol и TFTP (Trivial File Transfer Protocol). Эту микросхему ПЗУ называют *bootp ROM*.

- Загрузить образ по сети с узла QNX4. Для этого на сетевой карте должна быть специальная микросхема ПЗУ, содержащая расширение BIOS, поддерживающее протокол QNX Boot Protocol. Эту микросхему ПЗУ называют *QNX boot ROM*.

- Загрузиться через последовательный канал. Для этого на сервере загрузки используется утилита **`sendnto`**, на целевой системе для получения такого образа требуется специальный IPL.

6.3.1. Загрузка целевой системы с узла QNX4

При включении питания целевой системы запускается код, содержащийся в QNX boot ROM на сетевой карте. Он шлет широковещательный Ethernet-запрос, разыскивая образ. Этот запрос содержит MAC-адрес Ethernet-карты целевой системы.

Процесс **netboot** на узле QNX4 (узел QNX4 должен находится в той же подсети, что и целевая система) просматривает каждый Ethernet-кадр на предмет наличия запроса на загрузку от QNX boot ROM.

Как только запрос получен, **netboot** ищет в файле `/etc/config/netmap` по полученному MAC-адресу *логический номер узла* (logical node number). Затем по логическому номеру узла отыскивается нужная запись в файле `/etc/config/netboot`. В соответствии с этой записью **netboot** посылает образ на QNX boot ROM. QNX boot ROM загружает образ в ОЗУ и запускает его.

> ***Примечание***
> Подробную информацию об утилите **netboot** можно найти в документации ОСРВ QNX4.*xx*.

6.3.2. Загрузка целевой системы с TFTP-сервера

Этот способ удобнее, чем загрузка с QNX4, по нескольким причинам:

- bootp ROM гораздо дешевле, чем QNX boot ROM;
- в качестве сервера загрузки можно использовать ЭВМ с разными операционными системами;
- использование TCP/IP позволяет размещать сервер и целевую систему в разных подсетях.

При включении питания целевой системы запускается код, содержащийся в bootp ROM на сетевой карте. При этом выполнение проходит в две стадии:

1. Сетевая карта шлет DHCP-запрос. Ответ от DHCP-сервера (или BOOTP-сервера — без разницы, но DHCP более гибок) содержит IP-адрес целевой системы, IP-адрес TFTP-сервера и имя образа.
2. Сетевая карта шлет TFTP-запрос. В ответ получает файл образа, который она загружает и передает ему управление.

В QNX IDE есть встроенный TFTP-сервер, его главное достоинство — он сам "знает" где лежат образы, сгенерированные IDE (обычные TFTP-серверы умеют работать только с одним каталогом, помещать образы в который следует вручную).

6.3.3. Размещение образа в ПЗУ целевой системы

Есть несколько способов переноса образа QNX Neutrino в ПЗУ целевой системы, зависящих от применяемой аппаратуры.

- *Программирование образа с помощью ППЗУ-программатора* — в случае, если флэш-память не напаяна на процессорную плату (т. е. "не on-board"). При этом может понадобиться операция преобразования образа в формат, понятный программатору, с помощью утилиты **mkrec**.
- *Перенос образа в onboard Flash-память с помощью программатора через специальную шину (например, JTAG)* — возможно, также понадобится утилита **mkrec**.
- *Использование настройки BIOS для программирования образа через устройство ввода/вывода* — этот вариант возможен для некоторых процессорных плат, BIOS Setup которых позволяет войти в сервисный режим программирования onboard-носителей (например, Fastwel CPU686E). Устройством ввода/вывода может быть любое коммуникационное устройство, например, последовательный порт.
- *Программирование образа с помощью драйвера встроенной файловой системы* — этот вариант возможен только тогда, когда имеется драйвер для соответствующей микросхемы Flash. К тому же, скорее всего, потребуется установить связь между целевой и инструментальной системами по сети или другим способом (например, через последовательный порт). В составе QNX Momentics PE поставляется ряд драйверов флэш-устройств, но если подходящего среди них все-таки нет, то его можно разработать (самостоятельно или заказать).

6.4. Системное профилирование

QNX Neutrino — 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 обязательно должны быть эффективные инструменты анализа поведения системы в целом, т. е. того, как процессы уживаются вместе, что происходит при возникновении прерываний и т. п.

Для выполнения системного профилирования в состав QNX Momentics PE входит пакет System Analysis Toolkit (SAT). В операционной системе происходят различные события, все они так или иначе проходят через микроядро, поэтому основу SAT составляет модуль **procnto-instr**[1] — Администратор процессов, с которым скомпоновано микроядро, оборудованное средствами диагностики. QNX утверждает, что производительность "инструментированного" микроядра составляет не менее 98% производительности обычного микроядра. Средства, входящие в инструментальное расширение микроядра, записывают информацию о системных событиях в специальных буферах ядра (*буферах трассировки*) в виде записей. Средства сбора данных извлекают информацию из буферов трассировки в специальный файл журнала трассировки, который для увеличения скорости этой операции имеет двоичный формат — т. е. средства сбора данных не пытаются анализировать данные. Конвертирование двоичного журнала трассировки в понятный для человека формат выполняются в любой момент средствами визуализации трассы.

Системное профилирование может выполняться тремя способами:

- С помощью утилиты **qconn**. В этом случае в качестве средства сбора данных используется утилита **qconn**, которая может на выбор сохранять данные в журнале трассировки на целевой системе или сразу отправлять их в виде потока средству визуализации — перспективе QNX System Profiler интегрированной среды разработки. Важным достоинством этого способа является возможность ограничивать перечень регистрируемых событий трассировки теми событиями, которые нас действительно интересуют — уж больно много информации собирает инструментальное ядро по умолчанию.
- С помощью утилиты **tracelogger**. Утилита сохраняет данные в журнале трассировки. Для конвертирования данных в форму, удобную для чтения, следует или воспользоваться утилитой **traceprinter**, или написать свою программу обработки журнала трассировки (в составе SAT есть специальная библиотека `libtraceparser.a` — SAT API), или воспользоваться перспективой QNX System Profiler.

---

[1] На самом деле модулей ***-instr** несколько. Например, “инструментированный”модуль **procnto** с поддержкой SMP – **procnto-smp-instr**.

- Управлять процессом трассировки из своего приложения, используя функцию *TraceEvent()*. Данные будут извлекаться из буферов трассировки в журнал с помощью утилиты **`tracelogger`**. Этот метод позволяет весьма тонко настраивать процесс трассировки.

По умолчанию и утилита **`tracelogger`**, и **`qconn`** заказывают у **`procnto-instr`** 32 буфера. Каждый из буферов состоит из 1024 слотов по 16 байт (т. е. каждый буфер имеет размер 16 Кбайт, рис. 6.2). Событие ядра, для описания которого достаточно одного слота, называют *fast*. Те события, для описания которых недостаточно одного слота, называют *wide*.

Рис. 6.2. Схема использования буферов трассировки в **`procnto-instr`**

Для управления трассировкой можно задавать фильтры:

- *Режим fast или wide*. Fast указывает, что информация о wide-событиях должна обрезаться до размера fast. Это позволяет сэкономить слоты и увеличить скорость трассировки. Режим fast задан по умолчанию.
- *Статический и динамический фильтры*. Позволяют задавать правила записи событий в буферы, например, указывать, какие именно события нас интересуют. Статические фильтры используются утилитой **`qconn`**, динамический фильтр — это программа, использующая вызовы функции *TraceEvent()*.
- *Выходной фильтр*. Не влияет на запись трассы, а только позволяет извлекать нужную информацию из журнала трассировки. Пример создания динамического фильтра есть в электронной документации по SAT ("System Analysis Toolkit User's Guide").

На рис. 6.3 показано расположение фильтров в схеме трассировки. Видно, что все фильтры напрямую влияют на системное профилирование, кроме выходного.

Рис. 6.3. Расположение фильтров в схеме трассировки

На рис. 6.4 показано, как задаются правила фильтрации при системном профилировании с помощью QNX IDE.

Рис. 6.4. Задание правил фильтрации

Окно перспективы QNX System Profiler показано на рис.6.5.

Рис. 6.5. Окно перспективы QNX System Profiler

Подробности о системном профилировании читайте в руководстве "BSPs, DDKs, and specialty tools: System Analysis Toolkit User's Guide".