Операционная система QNX4

Системная архитектура

Оглавление

Об этой книге

Книга *Системная архитектура* сопровождает операционную систему QNX и предназначена как для разработчиков приложений, так и для конечных пользователей.

В книге подробно рассматривается структура и функции QNX. В ней описаны микроядро, системные менеджеры, а также уникальный механизм связи между процессами, основанный на передаче сообщений. Прежде чем использовать QNX, рекомендуется сначала прочитать эту книгу.

За информацией об установке и использовании QNX, обратитесь к книге *Руководство пользователя* ОС QNX.

Системная архитектура содержит следующие главы:

1. **Концепция QNX**
2. **Микроядро**
3. **Менеджер процессов**
4. **Пространство имен ввода/вывода**
5. **Менеджер файловой системы**
6. **Менеджер устройств**
7. **Менеджер сети**
8. **Оконная система Photon microGUI**

Глава 1. Концепция QNX

Эта глава охватывает следующие темы:

- Что такое QNX?
- Архитектура микроядра
- Связь между процессами (IPC)
- QNX как сеть

Что такое QNX?

Главная обязанность операционной системы состоит в управлении ресурсами компьютера. Все действия в системе - диспетчеризация прикладных программ, запись файлов на диск, пересылка данных по сети и т.п. - должны выполняться совместно настолько слитно и прозрачно, насколько это возможно.

Некоторые области применения предъявляют более жесткие требования к управлению ресурсами и диспетчеризации программ, чем другие. Приложения реального времени, например, полагаются на способность операционной системы обрабатывать многочисленные события в пределах ограниченного интервала времени. Чем быстрее реагирует операционная система, тем большее пространство для маневра имеет приложение реального времени в пределах жестких временных рамок.

Операционная система QNX идеальна для приложений реального времени. Она обеспечивает все неотъемлемые составляющие системы реального времени: многозадачность, диспетчеризацию программ на основе приоритетов и быстрое переключение контекста.

QNX - удивительно гибкая система. Разработчики легко могут настроить операционную систему таким образом, чтобы она отвечала требованиям конкретных приложений. QNX позволяет вам создать систему, использующую только необходимые для решения вашей задачи ресурсы. Конфигурация системы может изменяться в широком диапазоне - от ядра с несколькими небольшими модулями до полноценной сетевой системы, обслуживающей сотни пользователей.

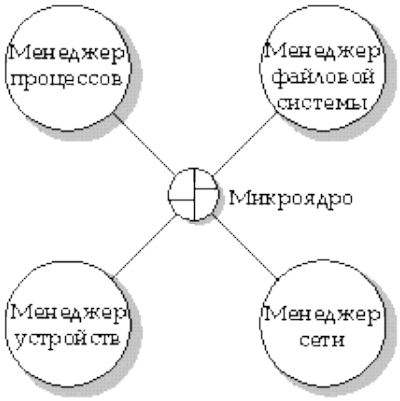
QNX достигает своего уникального уровня производительности, модульности и простоты благодаря двум фундаментальным принципам:

- архитектура на основе микроядра;
- связь между процессами на основе сообщений.

Архитектура микроядра системы QNX

QNX состоит из небольшого ядра, координирующего работу взаимодействующих процессов. Как показано на рисунке, структура больше напоминает не иерархию, а команду, в которой

несколько игроков одного уровня взаимодействуют между собой и со своим "защитником" — ядром.



Микроядро системы QNX координирует работу системных менеджеров.

Настоящее ядро

Ядро - это "сердце" любой операционной системы. В некоторых операционных системах на него возлагается так много функций, что ядро, по сути, заменяет всю операционную систему!

В QNX же Микроядро - это настоящее ядро. Во-первых, как и следует ядру реального времени, ядро QNX имеет очень маленький размер. Во-вторых, оно выполняет две важнейшие функции:

- **передача сообщений** - Микроядро обеспечивает маршрутизацию всех сообщений между всеми процессами в системе;
- **диспетчеризация** - планировщик - это часть Микроядра, и он получает управление всякий раз, когда процесс изменяет свое состояние в результате получения сообщения или прерывания.

В отличие от всех остальных процессов, ядро никогда не получает управления в результате диспетчеризации. Входящий в состав ядра код выполняется только в результате прямых вызовов из процесса или аппаратного прерывания.

Системные процессы

Все услуги операционной системы, за исключением тех, которые выполняются ядром, в QNX предоставляются через стандартные процессы. Типичная конфигурация QNX имеет следующие системные процессы:

- Менеджер процессов (`Proc`);
- Менеджер файловой системы (`Fsys`);
- Менеджер устройств (`Dev`);

- Менеджер сети (`Net`).

Системные и пользовательские процессы

Системные процессы практически ничем не отличаются от любых написанных пользователем программ - они не имеют какого-либо скрытого или особого интерфейса, недоступного пользовательским процессам.

Именно за счет такой системной архитектуры QNX обладает уникальной наращиваемостью. Так как большинство услуг операционной системы предоставляются стандартными процессами QNX, то расширение операционной системы требует всего лишь написания новой программы, обеспечивающей новую услугу!

Фактически, граница между операционной системой и прикладной программой может быть очень размыта. Единственный критерий, по которому мы можем отличить прикладные процессы и системные сервисные процессы, состоит в том, что процесс операционной системы управляет каким-либо ресурсом в интересах прикладного процесса.

Предположим, что вы написали сервер базы данных. Как же должен быть классифицирован этот процесс?

Точно так же, как сервер файловой системы принимает запросы (в QNX реализованные через механизм сообщений) на открытие файлов и запись или чтение данных, это будет делать и сервер базы данных. Хотя запросы к серверу базы данных могут быть и более сложными, сходство обоих серверов заключается в том, что оба они обеспечивают доступ к ресурсу посредством запросов. Оба они являются независимыми процессами, которые могут быть написаны пользователем и запускаться по мере необходимости.

Сервер базы данных может рассматриваться как процесс в одном случае и как приложение в другом. *Это действительно не имеет значения!* Важно то, что создание и выполнение таких процессов в QNX не требует абсолютно никаких изменений в стандартных компонентах операционной системы.

Драйверы устройств

Драйверы устройств - это процессы, которые являются посредниками между операционной системой и устройствами и избавляют операционную систему от необходимости иметь дело с особенностями конкретных устройств.

Так как драйверы запускаются как обычные процессы, добавление нового драйвера в QNX не влияет на другие части операционной системы. Таким образом, добавление нового драйвера в QNX не требует ничего, кроме непосредственно запуска этого драйвера.

После запуска и завершения процедуры инициализации, драйвер может выбрать один из двух вариантов поведения:

- стать расширением определенного системного процесса;
- продолжать выполнение как независимый процесс.

Связь между процессами (IPC)

В типичной для многозадачной системы реального времени ситуации, когда несколько процессов выполняются одновременно, операционная система должна предоставить механизмы, позволяющие им общаться друг с другом.

Связь между процессами (Interprocess communication, сокращенно IPC) является ключом к разработке приложений как совокупности процессов, в которых каждый процесс выполняет отведенную ему часть общей задачи.

QNX предоставляет простой, но мощный набор возможностей IPC, которые существенно облегчают разработку приложений, состоящих из взаимодействующих процессов.

Передача сообщений

QNX была первой коммерческой операционной системой своего класса, которая использовала передачу сообщений в качестве основного способа IPC. Именно последовательное воплощение метода передачи сообщения в масштабах всей операционной системы обусловливает мощность, простоту и элегантность QNX.

Сообщения в QNX - это последовательность байт, передаваемых от одного процесса другому. Операционная система не пытается анализировать содержание сообщения - передаваемые данные имеют смысл только для отправителя и получателя, и ни для кого более.

Передача сообщения позволяет не только обмениваться данными, но и является способом синхронизации выполнения нескольких процессов. Когда они посылают, получают или отвечают на сообщения, процессы претерпевают различные "изменения состояния", которые влияют на то, когда и как долго они могут выполняться. Зная состояния и приоритеты процессов, ядро организует их диспетчеризацию таким образом, чтобы максимально эффективно использовать ресурсы центрального процессора (ЦП).

Приложение реального времени и другие ответственные приложения по праву нуждаются в надежном механизме передачи сообщений, т.к. входящие в состав этих приложений процессы тесно взаимосвязаны. Реализованный в QNX механизм передачи сообщений способствует упорядочению и повышению надежности программ.

QNX как сеть

В простейшем случае локальная сеть обеспечивает разделяемый доступ к файлам и периферийным устройствам для нескольких соединенных между собой компьютеров. QNX идет гораздо дальше этого простейшего представления и объединяет всю сеть в единый однородный набор ресурсов.

Любой процесс на любом компьютере в составе сети может непосредственно использовать любой ресурс на любом другом компьютере. С точки зрения приложений, не существует никакой разницы между местным или удаленным ресурсом, и использование удаленных ресурсов не требует каких-либо специальных средств. Более того, чтобы определить, находится ли такой ресурс как файл или устройство на локальном компьютере или на другом узле сети, в программу потребуется включить специальный дополнительный код!

Пользователи могут иметь доступ к файлам по всей сети, использовать любое периферийное устройство, запускать программы на любом компьютере сети (при условии, что они имеют надлежащие полномочия). Связь между процессами осуществляется единообразно, независимо от их местоположения в сети. В основе такой прозрачной поддержки сети в QNX лежит всеобъемлющая концепция IPC на основе передачи сообщений.

Модель единого компьютера

QNX изначально проектировался как сетевая операционная система. В некоторых отношениях QNX сеть напоминает скорее большую ЭВМ, нежели набор мини-компьютеров. Пользователям известно, что в распоряжении любой из прикладных программ имеется большой набор ресурсов. Но в отличие от большой ЭВМ, QNX обеспечивает быструю реакцию системы, т.к. соответствующий объем вычислительных ресурсов может быть выделен на каждом узле в соответствии с потребностями каждого пользователя.

В условиях управления производством используются программируемые контроллеры и другие устройства ввода/вывода, а также комплексы программ, работающие в режиме реального времени, которым может потребоваться больше ресурсов, чем другим менее ответственным приложениям, таким как текстовый редактор. Сеть QNX достаточно "отзывчива", чтобы поддерживать *одновременно* оба этих типа приложений, QNX позволяет сфокусировать вычислительную мощность системы на производственном оборудовании (там, где это необходимо), в то же время, не жертвуя интерфейсом пользователя.

Гибкая поддержка сети

QNX сеть может быть построена с использованием различного оборудования и стандартных промышленных протоколов. В силу своей полной прозрачности для прикладных программ и пользователей, новые сетевые архитектуры могут быть внедрены в любое время, не разрушая операционной системы.

Примечание. Список поддерживаемого QNX сетевого оборудования пополняется со временем. Для получения подробной информации обратитесь к документации на сетевое оборудование, которое вы используете.

Каждому узлу QNX сети присваивается уникальный номер, который становится его идентификатором. Этот номер также единственный видимый признак того, функционирует QNX как сеть или как однопроцессорная операционная система.

Такая степень прозрачности является еще одним примером больших возможностей архитектуры QNX, основанной на передаче сообщений. Во многих операционных системах

такие важные функции как поддержка сети, IPC или даже передача сообщений выполнены в виде надстроек над операционной системой, а не интегрированы непосредственно в ее сердцевину. Результатом такого подхода является неуклюжий и неэффективный интерфейс с "двойным стандартом", когда связь между процессами - это одно дело, в то время как проникновение в скрытый интерфейс таинственного монолитного ядра - совершенно другое дело!

QNX, напротив, исходит из того, что эффективная связь является ключом к эффективной работе. Передача сообщений является, таким образом, краеугольным камнем архитектуры QNX, увеличивает эффективность *всех без исключения* транзакций между процессами в системе, независимо от того, идет ли речь о передаче данных по внутренней шине персонального компьютера или по коаксиальному кабелю на расстояние нескольких миль.

Теперь давайте перейдем к более подробному рассмотрению структуры QNX.

Глава 2. Микроядро

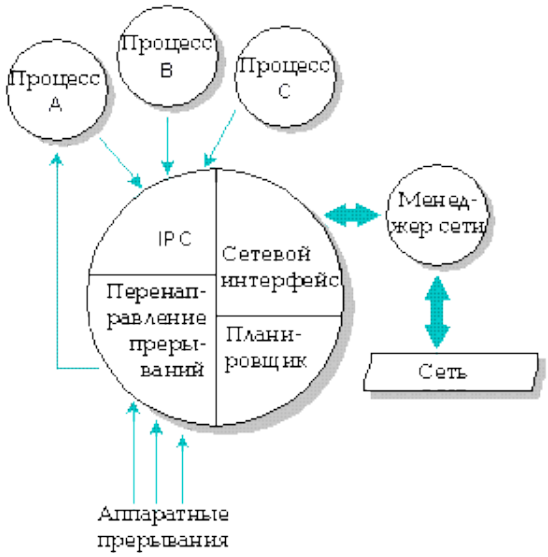
Эта глава охватывает следующие темы:

- Введение
- Связь между процессами (IPC)
- IPC посредством сообщений
- IPC посредством прокси
- IPC посредством сигналов
- IPC в сети
- IPC посредством семафоров
- Диспетчеризация процессов
- Несколько слов о реальном времени

Введение

Микроядро QNX отвечает за выполнение следующих функций:

- связь между процессами - Микроядро управляет маршрутизацией *сообщений*; оно также поддерживает две другие формы IPC - *прокси* и *сигналы*;
- сетевой интерфейс низкого уровня - Микроядро осуществляет доставку всех сообщений, предназначенных для процессов на других узлах сети;
- диспетчеризация процессов - входящий в состав Ядра *планировщик* решает, какому из запущенных процессов должно быть передано управление;
- первичная обработка прерываний - все аппаратные прерывания и исключения сначала проходят через Микроядро, а затем передаются соответствующему драйверу или системному менеджеру.



Внутри микроядра QNX

Связь между процессами

Микроядро QNX поддерживает три важнейшие формы связи между процессами: сообщения, прокси и сигналы.

- *Сообщения* - это основополагающая форма IPC в QNX. Они обеспечивают синхронную связь между взаимодействующими процессами, когда процессу, посылающему сообщение, требуется получить подтверждение того, что оно получено и, возможно, ответ.
- *Прокси* - это особый вид сообщения. Они больше всего подходят для извещения о наступлении какого-либо события, когда процессу, посылающему сообщение, не требуется вступать в диалог с получателем.
- *Сигналы* - это традиционная форма IPC. Они используются для асинхронной связи между процессами.

IPC посредством сообщений

Сообщения в QNX - это пакеты байт, которые синхронно передаются от одного процесса к другому. QNX при этом не анализирует содержание сообщения. Передаваемые данные понятны только отправителю и получателю и никому более.

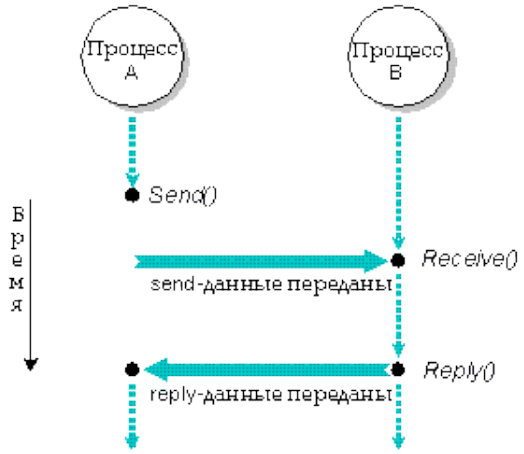
Примитивы передачи сообщений

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

Функция языка Си:	**Назначение:**
Send()	посылка сообщений
Receive()	получение сообщений
Reply()	ответ процессу, пославшему сообщение

Эти функции могут быть использованы как локально, т.е. для связи между процессами на одном компьютере, так и в пределах сети, т.е. для связи между процессами на разных узлах.

Следует заметить, однако, что далеко не всегда возникает необходимость использовать функции *Send()*, *Receive()* и *Reply()* в явном виде. Библиотека функций языка Си в QNX построена на основе использования сообщений - в результате, когда процесс использует стандартные механизмы передачи данных (такие, как, например, программный канал - pipe), он косвенным образом использует передачу сообщений.



Процесс A посылает сообщение процессу B, который получает его,обрабатывает и посылает ответ

На рисунке изображена последовательность событий, имеющих место, когда два процесса, процесс A и процесс B, используют функции *Send()*, *Receive()* и *Reply()* для связи друг с другом:

1. Процесс A посылает сообщение процессу B, вызвав функцию *Send()*, которая передает соответствующий запрос ядру. В этот момент времени процесс A переходит в SEND-блокированное состояние и остается в этом состоянии до тех пор, пока процесс B не вызовет функцию *Receive()* для получения сообщения.
2. Процесс B вызывает *Receive()* и получает сообщение от процесса A. При этом состояние процесса A изменяется на REPLY-блокирован. Процесс B при вызове функции *Receive()* в данном случае не блокируется, т.к. к этому моменту его уже ожидало сообщение от процесса A.

Заметьте, что если бы процесс B вызвал *Receive()* до того, как ему было послано сообщение, то он бы попал в состояние RECEIVE-блокирован до получения сообщения. В этом случае процесс-отправитель сообщения немедленно после посылки сообщения попал бы в состояние REPLY-блокирован.

1. Процесс B выполняет обработку полученного от процесса A сообщения и затем вызывает функцию *Reply()*. Ответное сообщение передается процессу A, который переходит в состояние готовности к выполнению. Вызов *Reply()* не блокирует процесс B, который также готов к выполнению. Какой из этих процессов будет выполняться, зависит от их приоритетов.

Синхронизация процессов

Передача сообщений не только позволяет процессам обмениваться данными, но и предоставляет механизм синхронизации выполнения нескольких взаимодействующих процессов.

Давайте снова рассмотрим приведенный выше рисунок. После того как процесс A вызвал функцию *Send()*, он не может продолжать выполнение до тех пор, пока не получит ответ на посланное сообщение. Это гарантирует, что выполняемая процессом B по запросу процесса

А обработка данных будет завершена прежде, чем процесс А продолжит выполнение. Более того, после вызова процессом В запроса на получение данных *Receive()*, он не может продолжать выполнение до тех пор, пока не получит следующее сообщение.

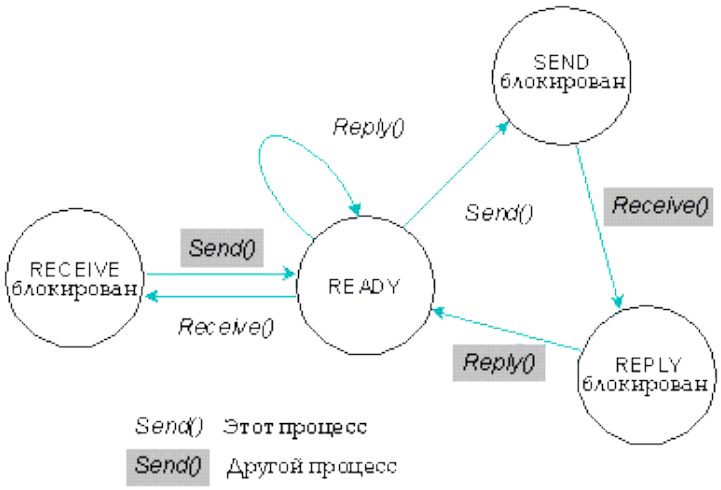
Примечание. Более подробно диспетчеризация процессов в QNX рассматривается в разделе "**Диспетчеризация процессов**" далее в этой главе.

Блокированные состояния

Когда процессу не разрешается продолжать выполнение, т.к. он должен ожидать окончания определенной стадии протокола передачи сообщения, - процесс называется *блокированным*.

Возможные блокированные состояния процессов приведены в следующей таблице:

Если процесс выдал:	**То процесс:**
Запрос *Send()*, и отправленное им сообщение еще не получено процессом-получателем	SEND-блокирован
Запрос *Send()*, и отправленное им сообщение получено процессом-получателем, но ответ еще не выдан	REPLY-блокирован
Запрос *Receive()*, но еще не получил сообщение	RECEIVE-блокирован



Изменение состояния процессов в типичном случае передачи сообщения

Примечание. Для получения информации обо всех возможных состояниях процесса смотри главу "**Менеджер процессов**".

Использование *Send()*, *Receive()* и *Reply()*

Давайте теперь более подробно рассмотрим вызовы функций *Send()*, *Receive()* и *Reply()*. Воспользуемся рассмотренным выше примером передачи сообщения от процесса А к

процессу B.

Функция *Send()*

Предположим, что процесс A выдает запрос на передачу сообщения процессу B. Это выполняется посредством вызова функции *Send()*:

```
Send( ''pid'', ''smsg'', ''rmsg'', ''smsg_len'', ''rmsg_len'' );
```

При вызове функции *Send()* используются следующие аргументы:

- *pid* – идентификатор процесса (*p*rocess *ID*), которому предназначается сообщение (т.е. процесса B); этот идентификатор используется для обращения к процессу со стороны операционной системы и других процессов;
- *smsg* – буфер сообщения (т.е. сообщение, подлежащее посылке)
- *rmsg* – буфер ответа (будет содержать ответ от процесса B)
- *smsg_len* – длина посылаемого сообщения в байтах
- *rmsg_len* – максимальная длина ответа, который может принять процесс A, в байтах

Обратите внимание, что будет передано не более *smsg_len* байт и не более чем *rmsg_len* байт будет получено в качестве ответа - это гарантирует, что не произойдет случайного переполнения буферов.

Функция *Receive()*

Вызвав запрос *Receive()*, процесс B может получить сообщение, направленное ему процессом A:

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

Вызов функции *Receive()* содержит следующие аргументы:

- *pid* – идентификатор процесса, который послал сообщение (т.е. процесс A)
- 0 – (ноль) означает, что процесс B желает принять сообщение от любого процесса
- *msg* – буфер, куда будет помещено принимаемое сообщение
- *msg_len* – максимальный размер данных, которые будут помещены в буфер приема, в байтах

Если значения аргументов *smsg_len* в вызове функции *Send()* и *msg_len* в вызове функции *Receive()* отличаются, друг от друга, то наименьшее из них определяет размер данных, которые будут переданы.

Функция *Reply()*

После успешного получения сообщения от процесса A, процесс B должен ответить процессу A, вызвав функцию *Reply()*:

```
Reply( ''pid'', ''reply'', ''reply_len'' );
```
Вызов функции *Reply()* содержит следующие аргументы:

- *pid* – идентификатор процесса, которому предназначается ответ (т.е. процесс A)
- *reply* – буфер, содержащий ответ
- *reply_len* – длина данных, передаваемых в качестве ответа, в байтах

Если значения аргументов *reply_len* при вызове функции *Reply()* и *rmsg_len* при вызове функции *Send()* отличаются друг от друга, то наименьшее из них определяет размер передаваемых данных.

Reply-управляемый обмен сообщениями

Пример обмена сообщениями, который мы только что рассмотрели, иллюстрирует наиболее распространенный случай использования сообщений - случай, когда для процесса, выполняющего функции сервера, нормальным является состояние RECEIVE-блокирован в ожидании каких-либо запросов клиента. Это называется *send-управляемый обмен сообщениями*: процесс-клиент инициирует действие, посылая сообщения, ответ сервера на сообщение завершает действие.

Существует и другая модель обмена сообщениями, не столь распространенная, как рассмотренная выше, хотя в некоторых ситуациях она даже более предпочтительна: *reply-управляемый обмен* сообщениями, при котором действие инициируется вызовом функции *Reply()*. В этом случае процесс-"работник" посылает серверу сообщение, говорящее о том, что он готов к работе. Сервер не отвечает сразу, а "запоминает", что работник готов выполнить его задание. В последствии сервер может инициировать действие, ответив ожидающему задание работнику. Процесс-работник выполнит задание и завершит действие, послав серверу сообщение, содержащее результаты своей работы.

Дополнительные сведения

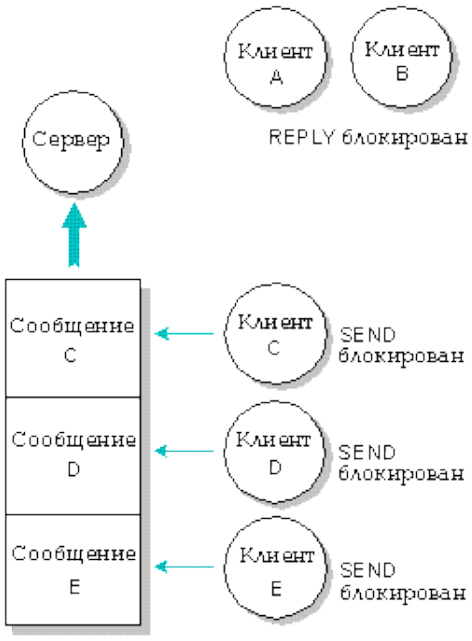
При разработке программ, использующих передачу сообщений, необходимо иметь в виду следующее:

- сообщение сохраняется в теле посылающего процесса до тех пор, пока принимающий процесс не будет готов получить его. Сообщение *не* копируется в Микроядро. Это безопасно, т.к. посылающий процесс SEND-блокирован и не может неумышленно изменить содержимое сообщения
- при вызове функции *Reply()* данные копируются из отвечающего процесса в REPLY-блокированный процесс как одна неразрывная операция. Вызов функции *Reply()* не блокирует процесс - REPLY-блокированный процесс перестает быть блокированным после того, как в него скопированы данные;
- при посылке сообщения процессу не требуется знать состояние того процесса, которому это сообщение адресовано. Если получатель не готов к приему сообщения в момент его отправки, то посылающий сообщение процесс просто становится SEND-блокирован;
- если это необходимо, то процесс может посылать сообщение нулевой длины, ответ нулевой длины, либо и то и другое;
- с точки зрения разработчика, использование *Send()* для передачи процессу-серверу запроса на получение какой-либо услуги равнозначно вызову библиотечной подпрограммы, предоставляющей ту же самую услугу. В обоих случаях ваша

программа сначала подготавливает определенные данные, а затем вызывает либо функцию *Send()*, либо библиотечную подпрограмму, после чего ожидает, когда они закончат выполнение. В обоих случаях код, реализующий запрашиваемую услугу, выполняется между двумя четко определенными точками - *Receive()* и *Reply()* для процесса-сервера, входом в подпрограмму и командой return для вызова библиотечной подпрограммы. Когда вызываемая сервисная программа завершена, ваша программа "знает", куда помещены результаты, и может приступить к проверке возвращенного кода ошибки, обработке результатов и так далее.

Несмотря на такую кажущуюся простоту, вызов функции *Send()* делает гораздо больше, чем простой вызов библиотечной подпрограммы. Функция *Send()* может прозрачно для вызывающей программы передать запрос на другой узел сети, где и будет в действительности выполняться обслуживающий запрос код. При этом также может быть задействована параллельная обработка данных без издержек на создание нового процесса. Процесс-сервер может сразу, как только станет возможным, вызвать функцию *Reply()*, позволяя тем самым запрашивавшему процессу возобновить выполнение и, в то же время, продолжить выполнение самому.

- возможна ситуация, когда сообщения от многих процессов одновременно ожидают своей обработки одним и тем же принимающим процессом. Как правило, принимающий процесс получает сообщение в том же порядке, в каком они были посланы другими процессами; однако принимающий процесс может установить очередность получения сообщений на основе приоритетов посылающих процессов.



Сервер получил сообщение от клиента A и клиента B (но еще не ответил им). Сообщения от клиентов C, D, E еще не получены

Дополнительные возможности

QNX также предоставляет следующие дополнительные возможности по передаче

сообщений:

- условный прием сообщений;
- чтение или запись части сообщений;
- составные сообщения (сообщения, состоящие из нескольких частей).

Условный прием сообщений

Обычно, когда процесс хочет принять сообщение, он вызывает функцию *Receive()* для ожидания прихода сообщения. Это обычный способ получения сообщений, который пригоден в большинстве случаев.

Однако возможна ситуация, когда процессу необходимо определить, имеются ли ожидающие приема сообщения, не попадая при этом в состояние RECEIVE-блокирован в случае их отсутствия. Например, процессу требуется опрашивать работающее с высокой скоростью устройство, которое не способно генерировать прерывание, и в то же время он должен отвечать на сообщения от других процессов. В этом случае процесс может использовать функцию *Creceive()*, которая либо прочитает ожидающее приема сообщение, либо немедленно вернет управление процессу в случае отсутствия ожидающих приема сообщений.

Примечание. Следует по возможности избегать использования функции *Creceive()*, т.к. она позволяет процессу непрерывно выполняться с неизменяющимся уровнем приоритета.

Чтение или запись части сообщения

Иногда желательно читать или записывать сообщения по частям с тем, чтобы использовать уже выделенный для сообщения буфер вместо выделения отдельного рабочего буфера.

Например, менеджер ввода/вывода может принимать данные в виде сообщений, которые состоят из заголовка фиксированной длины и следующих за ним данных переменной длины. В заголовке указывается размер данных (от 0 до 64 Кбайт). В этом случае менеджер ввода/вывода может сначала принять только заголовок сообщения, а затем использовать функцию *Readmsg()* для чтения данных переменной длины непосредственно в соответствующий буфер вывода. Если размер данных превышает размер буфера, то менеджер может неоднократно вызывать функцию *Readmsg()* по мере освобождения буфера вывода. Аналогичным образом, функция *Writemsg()* может быть использована для поэтапного копирования данных в выделенный для ответного сообщения буфер уменьшая, таким образом, потребность менеджера ввода/вывода в выделении внутренних буферов.

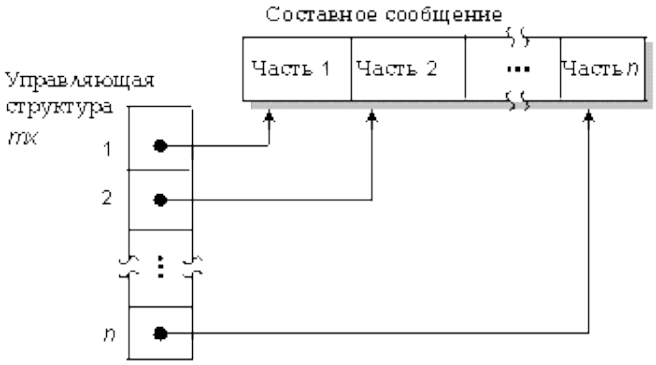
Составные сообщения

До сих пор мы рассматривали сообщения как непрерывную последовательность байт. Однако сообщения часто состоят из двух или более отдельных частей. Например, сообщение может иметь заголовок фиксированной длины, за которым следуют данные переменной длины. Для того чтобы избежать копирования частей такого сообщения во временные промежуточные буферы при передаче или приеме, может быть использовано составное сообщение, состоящее из двух или более отдельных буферов сообщений. Именно благодаря этой возможности

менеджеры ввода/вывода QNX, такие как Dev и Fsys, достигают своей высокой производительности.

Следующие функции позволяют обрабатывать составные сообщения:

- *Creceivemx()*
- *Readmsgmx();*
- *Receivemx();*
- *Replymx();*
- *Sendmx();*
- *Writemsgmx().*



Составные сообщения могут быть описаны с помощью специальной mx структуры. Микроядро объединяет части такого сообщения в единый непрерывный поток данных

Зарезервированные коды сообщений

QNX начинает все сообщения с 16-ти битного слова, называемого *кодом сообщения*. Заметим, однако, что это не является обязательным для вас требованием при написании собственных программ. QNX использует коды сообщений в следующих диапазонах:

Зарезервированный диапазон:	**Описание:**
0x0000 - 0x0000	Сообщения Менеджера процессов
0x0100 - 0x01FF	Сообщения ввода/вывода (общие для всех серверов ввода/вывода)
0x0200 - 0x02FF	Сообщения Менеджера файловой системы
0x0300 - 0x03FF	Сообщения Менеджера устройств
0x0400 - 0x04FF	Сообщения Менеджера сети
0x0500 - 0x0FFF	Зарезервированы для будущих системных процессов QNX

IPC посредством прокси

Прокси - это форма неблокирующего сообщения, особенно подходящего для извещения о наступлении события, когда посылающий процесс не нуждается в диалоге с получателем. Единственная функция прокси состоит в посылке фиксированного сообщения

определенному процессу, который является владельцем прокси. Подобно сообщениям, прокси могут быть использованы в пределах всей сети.

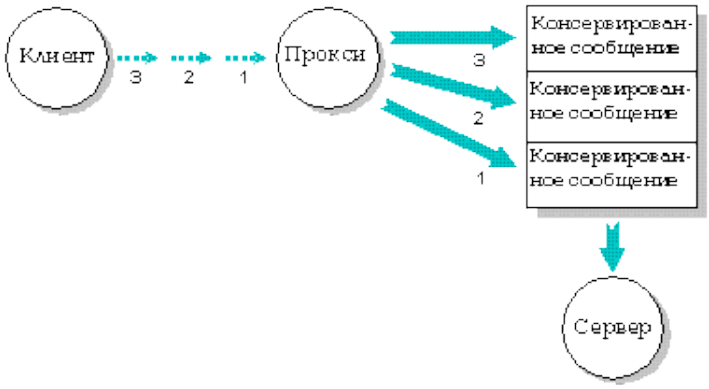
Используя прокси, процесс или обработчик прерывания может послать сообщение другому процессу, не блокируясь и не ожидая ответа.

Вот некоторые примеры использования прокси:

- процесс желает известить другой процесс о наступлении какого-либо события, но при этом не может позволить себе посылку сообщения (в этом случае он оставался бы блокированным до тех пор, пока получатель не вызовет *Receive()* и *Reply()*);
- процесс хочет послать данные другому процессу, но при этом ему не требуется ни ответа, ни какого-либо другого подтверждения того, что адресат (получатель) получил сообщение;
- обработчик прерывания хочет известить процесс о поступлении новых данных.

Для создания прокси используется функция языка Си *qnx_proxy_attach()*. Любой другой процесс или обработчик прерывания, которому известен идентификатор прокси, может воспользоваться функцией языка Си *Trigger()* для того, чтобы заставить прокси передать заранее заданное сообщение. Запрос *Trigger()* обрабатывается Микроядром.

Прокси может быть "запущено" неоднократно - каждый раз при этом оно посылает сообщение. Прокси может накапливать очередь длиной до 65535 сообщений.



Процесс-клиент запускает прокси 3 раза, в результате чего сервер получает 3 "консервированных" сообщения от прокси

IPC посредством сигналов

Сигналы являются традиционным способом асинхронной связи, которая используется в течение многих лет в различных операционных системах.

QNX поддерживает большой набор сигналов, соответствующих стандарту POSIX, кроме того, сигналы, исторически присущие некоторым UNIX-системам, и ряд сигналов, уникальных для QNX.

Как породить сигнал

Считается, что сигнал доставлен процессу тогда, когда выполняется определенное в процессе для данного сигнала действие. Процесс может посылать сигнал самому себе.

Если вы хотите:	**Используйте:**
Породить сигнал из командной строки	Утилиты: kill и slay
Породить сигнал внутри процесса	Функции Си: *kill()* и *raise()*

Получение сигналов

Процесс может принять сигнал одним из трех способов, в зависимости от того, как в процессе определена обработка сигналов:

- если процесс не определил никаких специальных мер по обработке сигнала, то выполняется предусмотренное для сигнала действие по умолчанию - обычно таким действием по умолчанию является завершение работы процесса;
- процесс может игнорировать сигнал. Если процесс игнорирует сигнал, то сигнал не оказывает на процесс никакого воздействия (учтите, что SIGCONT, SIGSTOP и SIGKILL не могут быть игнорированы в обычных обстоятельствах);
- процесс может предусмотреть *обработчик сигнала* - функцию, которая будет вызываться при приеме сигнала. Если процесс содержит обработчик для какого-либо сигнала, говорят, что процесс может "поймать" этот сигнал. Любой процесс, который "ловит" сигнал, фактически получает особый вид программного прерывания. Никакие данные с сигналом не передаются.

В промежутке времени между моментом, когда сигнал порожден, и моментом, когда он доставлен, сигнал называется ожидающим. Для процесса ожидающими одновременно могут быть несколько различных сигналов. Сигналы доставляются процессу, когда планировщик ядра делает этот процесс готовым к выполнению. Процесс не должен строить никаких предположений относительно порядка, в котором будут доставлены ожидающие сигналы.

Перечень сигналов

Сигнал:	**Описание:**
SIGABRT	Сигнал ненормального завершения, порождается функцией *abort()*
SIGALRM	Сигнал тайм-аута, порождается функцией *alarm()*
SIGBUS	Указывает на ошибку контроля четности оперативной памяти (особая для QNX интерпретация). Если во время выполнения обработчика данного сигнала произойдет вторая такая ошибка, то процесс будет завершен
SIGCHLD	Завершение порожденного процесса. Действие по умолчанию - игнорировать сигнал
SIGCONT	Если процесс в состоянии HELD, то продолжить выполнение. Действие по умолчанию - игнорировать сигнал, если процесс не в состоянии HELD

Сигнал:	**Описание:**
SIGDEV	Генерируется, когда в Менеджере устройств происходит важное и запрошенное событие
SIGFPE	Ошибочная арифметическая операция (целочисленная или с плавающей запятой), например, деление на ноль или операция, вызвавшая переполнение. Если во время выполнения обработчика данного сигнала произойдет вторая такая ошибка, то процесс будет завершен
SIGHUP	Гибель процесса, который был ведущим сеанса, или зависание управляющего терминала
SIGILL	Обнаружение недопустимой аппаратной команды. Если во время выполнения обработчика данного сигнала произойдет вторая такая ошибка, то процесс будет завершен
SIGINT	Интерактивный сигнал "внимание" (**Break**)
SIGKILL	Сигнал завершения - должен быть использован только в экстренных ситуациях. *Этот сигнал не может быть "пойман" или игнорирован*. Сервер может защитить себя от этого сигнала посредством функции языка Си *qnx_pflags()*. Для этого сервер должен иметь статус привилегированного пользователя
SIGPIPE	Попытка записи в программный канал, который не открыт для чтения
SIGPWR	Перезапуск компьютера в результате нажатия **Ctrl***Alt***Shift-Del** или вызова утилиты shutdown
SIGQUIT	Интерактивный сигнал завершения
SIGSEGV	Обнаружение недопустимой ссылки на оперативную память. Если во время выполнения обработчика данного сигнала произойдет вторая такая ошибка, то процесс будет завершен
SIGSTOP	Сигнал приостановки выполнения (HOLD) процесса. Действие по умолчанию - приостановить процесс. Сервер может защитить себя от этого сигнала посредством функции языка Си *qnx_pflags()*. Для этого сервер должен иметь статус привилегированного пользователя
SIGTERM	Сигнал завершения
SIGTSTP	Не поддерживается QNX
SIGTTIN	Не поддерживается QNX
SIGTTOU	Не поддерживается QNX
SIGUSR1	Зарезервирован как определяемый приложением сигнал 1
SIGUSR2	Зарезервирован как определяемый приложением сигнал 2
SIGWINCH	Изменился размер окна

Управление обработкой сигналов

Чтобы задать желаемый способ обработки для каждого из сигналов, вы можете использовать функции языка Си *signal()* стандарта ANSI или *sigaction()* стандарта POSIX.

Функция *sigaction()* предоставляет большие возможности по управлению обработкой сигналов.

Вы можете изменить способ обработки сигнала в любой момент времени. Если вы укажете, что сигнал данного типа должен игнорироваться, то все ждущие сигналы такого типа будут немедленно отброшены.

Обработчики сигналов

Некоторые специальные замечания касаются процессов, которые ловят сигналы посредством обработчиков сигналов.

Вызов обработчика сигнала аналогичен программному прерыванию. Он выполняется асинхронно по отношению к остальной части процесса. Таким образом, существует вероятность того, что обработчик сигнала будет вызван во время выполнения любой из имеющихся в программе функций (включая библиотечные функции).

Если в вашей программе не предусмотрен возврат из обработчика сигнала, то могут быть использованы функции *siglongjmp()* либо *longjmp()* языка Си, однако, функция *siglongjmp()* предпочтительнее. При использовании функции *longjmp()* сигнал остается блокированным.

Блокирование сигналов

Иногда может возникнуть необходимость временно запретить получение сигнала, не изменяя метод его обработки. QNX предоставляет набор функций, которые позволяют блокировать получение сигналов. Блокированный сигнал остается ожидающим; после разблокирования он будет доставлен вашей программе.

Пока ваша программа выполняет обработчик сигнала для определенного типа сигнала, QNX автоматически блокирует этот сигнал. Это означает, что нет необходимости заботиться о вложенных вызовах обработчика сигнала. Каждый вызов обработчика сигнала - это неделимая операция по отношению к доставке следующих сигналов этого типа. Если ваш процесс выполняет нормальный возврат из обработчика, сигнал автоматически разблокируется.

Примечание. Реализация обработчиков сигналов в некоторых UNIX системах отличается тем, что при получении сигналов они не блокируют его, а устанавливают действие по умолчанию. В результате некоторые UNIX приложения вызывают функцию *signal()* изнутри обработчика сигналов, чтобы подготовить обработчик к следующему вызову. Такой способ имеет два недостатка. Во-первых, если следующий сигнал поступает, когда ваша программа уже выполняет обработчик сигнала, но еще не вызвала функцию *signal()*, то программа может быть завершена. Во-вторых, если сигнал поступает сразу после вызова функции *signal()* в обработчике, то возможен рекурсивный вход в обработчик сигнала. QNX поддерживает блокирование сигналов и, таким образом, не страдает ни от одного из этих недостатков. Вам не нужно вызывать функцию *signal()* изнутри обработчика сигналов. Если вы покидаете обработчик через дальний переход (long jump), вам следует использовать функцию *siglongjmp()*.

Сигналы и сообщения

Существует важное взаимодействие между сигналами и сообщениями. Если ваш процесс

SEND-блокирован или RECEIVE-блокирован в момент получения сигнала, и вы предусмотрели обработчик сигнала, происходят следующие действия:

1. Процесс разблокируется;
2. Выполняется обработка сигнала;
3. Функция *Send()* или *Receive()* возвратит код ошибки.

Если процесс был SEND-блокирован в момент получения сигнала, то проблемы не возникает, потому что получатель еще не получил сообщение. Но если процесс был RECEIVE-блокирован, то вы не узнаете, было ли обработано посланное им сообщение и, следовательно, не будете знать, следует ли повторять *Send()*.

Процесс, играющий роль сервера (т.е. получающий сообщения), может запросить, чтобы он получал извещение всякий раз, когда его процесс-клиент получает сигнал, находясь в состоянии REPLY-блокирован. В этом случае клиент становится SIGNAL-блокированным с ожидающим сигналом. Сервер получает специальные сообщения, описывающие тип сигнала. Сервер может затем выбрать один из следующих вариантов:

- закончить выполнение запроса от клиента нормальным образом: клиент (процесс-отправитель) будет уведомлен, что его сообщение было обработано должным образом

или

- освободить используемые ресурсы и возвратить код ошибки, указывающий, что процесс был разблокирован сигналом: клиент получит четкий признак ошибки.

Когда сервер отвечает процессу, который был SIGNAL-блокирован, то сигнал выдается непосредственно после возврата из функции *Send()*, вызванной клиентом (процессом-отправителем).

IPC в сети

Виртуальные каналы

Приложение в QNX может общаться с процессом на другом компьютере сети точно так же, как если бы оно общалось с другим процессом на том же самом компьютере. Кстати говоря, с точки зрения приложения нет различия между локальным и удаленным ресурсом.

Такая замечательная степень прозрачности становится возможна благодаря *виртуальным каналам* (от английского virtual circuit, сокращенно VC). VC - это пути, которые Менеджер сети предоставляет для передачи сообщений, прокси и сигналов по сети. VC способствуют эффективному использованию QNX-сети в целом по нескольким причинам:

1. Когда создается VC, он получает возможность обрабатывать сообщения вплоть до определенного размера; это означает, что вы можете предварительно выделить ресурс для обработки сообщения. Тем не менее, если вам необходимо послать сообщение, превышающее заданный максимальный размер, VC автоматически увеличивает размер буфера.
2. Если два процесса, располагающиеся на различных узлах сети, используют для связи

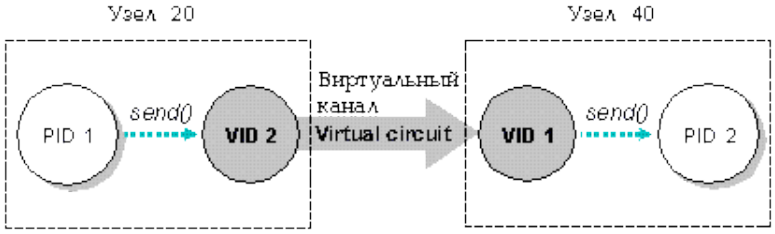
друг с другом более чем один VC, то такие VC будут разделяемыми, так как между процессами будет *реально* существовать только один виртуальный канал. Такая ситуация обычно имеет место, когда процесс использует несколько файлов на удаленной файловой системе.

3. Если процесс присоединяется к существующему разделяемому VC и запрашивает буфер большего размера, чем тот, который используется, то размер буфера автоматически увеличивается.
4. Когда процесс завершает выполнение, все связанные с ним VC автоматически освобождаются.

Виртуальные процессы

Процесс-отправитель отвечает за подготовку VC между собой и тем процессом, с которым он хочет установить связь. С этой целью процесс-отправитель обычно вызывает функцию *qnx_vc_attach()*. Кроме создания VC, эта функция также создает на каждом конце канала виртуальный идентификатор процесса, сокращенно VID. Для каждого из процессов, VID на противоположном конце виртуального канала является идентификатором удаленного процесса, с которым он устанавливает связь. Процессы поддерживают связь друг с другом посредством этих VID.

Например, на следующем рисунке виртуальный канал соединяет PID 1 и PID 2. На узле 20 - где находится PID 1 - VID представляет PID 2. На узле 40 - где находится PID 2 - VID представляет PID 1. И PID 1 и PID 2 могут обращаться к VID на своем узле так же, как к любому другому локальному процессу, посылая сообщения, принимая сообщения, поставляя сигналы, ожидая и т.д. Так, например, PID 1 может послать сообщение VID на своем конце, и этот VID передаст сообщение по сети к VID, представляющему PID 1 на другом конце. Этот VID затем отправит сообщение к PID 2.



Связь по сети реализуется посредством виртуальных каналов.Когда PID 1 посылает данные VID 2, посланный запрос передается по виртуальному каналу, в результате VID 1 направляет данные PID 2

Каждый VID поддерживает соединение, которое имеет следующие атрибуты:

- локальный *pid*;
- удаленный *pid*;
- удаленный *nid* (ID узла);
- удаленный *vid*.

Возможно, вам не придется часто иметь дело непосредственно с VC. Так, например, когда приложение хочет получить доступ к ресурсу ввода/вывода на другом узле сети, то VC

создается библиотечной функцией *open()* от имени приложения. Приложение не принимает непосредственного участия в создании или использовании VC. Аналогичным образом, когда приложение определяет местонахождение сервера с помощью функции *qnx_name_locate()*, то VC автоматически создается от имени приложения. Для приложения VC просто является PID сервера.

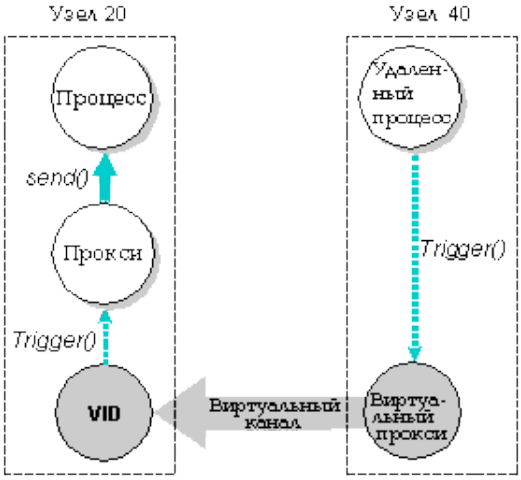
Для более подробной информации о функции *qnx_name_locate()* смотрите описание **"Символьные имена процессов"** в главе "Менеджер процессов".

Виртуальные прокси

Виртуальное прокси позволяет посылать прокси с удаленного узла, подобно тому, как виртуальный канал позволяет процессу обмениваться сообщениями с удаленным узлом.

В отличие от виртуального канала, который связывает вместе два процесса, виртуальный прокси может быть послан любым процессом на удаленном узле.

Виртуальные прокси создаются функцией *qnx_proxy_rem_attach()*, которой в качестве аргументов передаются узел (*nid_t*) и прокси (*pid_t*).



На удаленном узле создается виртуальный прокси, который ссылается на локальный прокси.

Имейте в виду, что на вызывающем узле виртуальный канал создается автоматически посредством функции *qnx_proxy_rem_attach()*.

Отключение виртуальных каналов

Процесс может утратить возможность связи по установленному VC в результате различных причин:

- произошло отключение питания компьютера, на котором выполнялся процесс;
- был отсоединен сетевой кабель компьютера;
- удаленный процесс, с которым была установлена связь, завершил выполнение.

Любая из этих причин может помешать передаче сообщений по VC. Необходимо выявлять такие ситуации с тем, чтобы приложения могли предпринять корректирующие действия, либо корректно завершить свое выполнение. Если это не будет сделано, то ценные ресурсы могут оставаться без необходимости постоянно занятыми.

Менеджер процессов на каждом узле проверяет целостность VC на своем узле. Он делает это следующим образом:

1. Каждый раз, когда происходит успешная передача по VC, обновляется временная метка, связанная с данным VC, в соответствии со временем последней операции.
2. Через определенные промежутки времени Менеджер процессов проверяет каждый VC. Если не было зафиксировано никаких операций, то Менеджер процессов посылает проверочный пакет Менеджеру процессов на другом конце VC.
3. Если ответ не получен или выявлен сбой, то VC помечается как неисправный. Затем делается определенное количество попыток восстановить соединение.
4. Если эти попытки не дают результата, то VC демонтируется; все процессы, блокированные VC, переходят в состояние READY (готов). Процесс получает код ошибки, возвращаемой ранее вызванной функцией связи.

Для установки параметров, связанных с данной проверкой целостности VC, используйте утилиту netpoll.

IPC посредством семафоров

Другой распространенной формой синхронизации процессов являются семафоры. Операции "сигнализации" (*sem_post()*) и "ожидания" (*sem_wait()*) позволяют управлять выполнением процессов, переводя их в состояние ожидания либо выводя из него. Операция сигнализации увеличивает значение семафора на единицу, а операция ожидания уменьшает его на единицу.

При положительном значении семафора, при выполнении операции ожидания, процесс не блокируется. В противном случае операция ожидания блокирует процесс до тех пор, пока какой-либо другой процесс не выполнит операцию сигнализации. Допускается выполнение операции сигнализации один или более раз перед выполнением операции ожидания - это позволит одному или нескольким процессам выполнить операцию ожидания, не блокируясь.

Важное отличие между семафорами и другими примитивами синхронизации заключается в том, что семафоры "асинхронно безопасны" и могут использоваться обработчиками сигналов. Если требуется, чтобы обработчик сигнала выводил процесс из состояния ожидания, то семафоры являются правильным выбором.

Диспетчеризация процессов

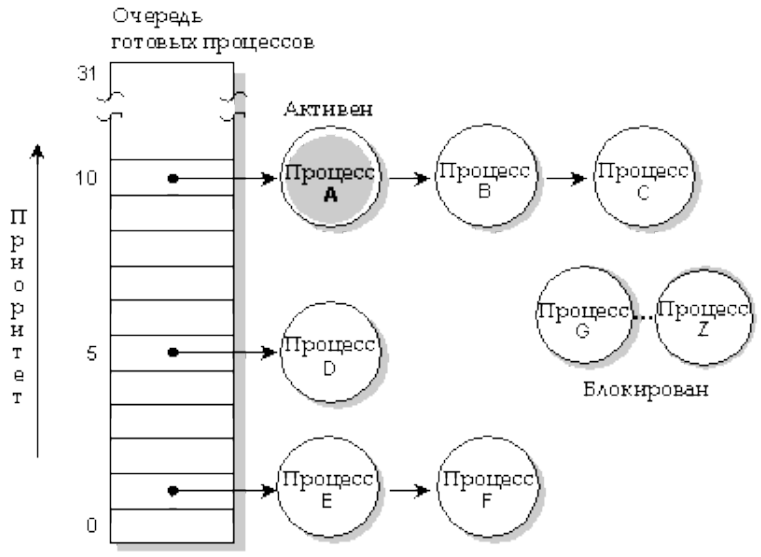
Когда принимаются решения по диспетчеризации

Планировщик Микроядра принимает решение по диспетчеризации:

- после разблокировки процесса;
- по истечении временного интервала (кванта) для выполняющегося процесса;
- когда прерывается текущий процесс.

Приоритеты процессов

В QNX каждому из процессов присваивается приоритет. Планировщик выбирает для выполнения следующий процесс, находящийся в состоянии READY, в соответствии с его приоритетом. (Программа в состоянии READY - это программа, которая способна использовать ЦП). Для выполнения выбирается процесс с наивысшим приоритетом.



В очереди шесть процессов (A-F), готовых к выполнению и находящихся в состоянии READY. Остальные процессы (G-Z) блокированы. В данный момент выполняется процесс A. Процессы A, B и C имеют наивысший приоритет, поэтому будут разделять центральный процессор в соответствии с алгоритмом диспетчеризации для выполняемого процесса

Приоритеты, присваиваемые процессам, находятся в диапазоне от 0 (наименьший) до 31 (наивысший). Уровень приоритета по умолчанию для создаваемого процесса наследуется от его родителя; для приложений, запускаемых командным процессором, приоритет обычно равен 10.

Если вы хотите:	**Используйте функцию языка СИ:**
Определить приоритет процесса	*getprio()*
Установить приоритет для процесса	*setprio()*

Методы диспетчеризации

Чтобы удовлетворить потребность различных приложений, QNX предлагает три метода диспетчеризации:

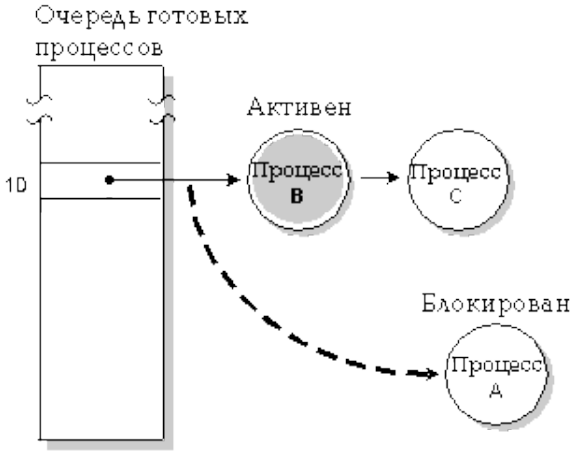
- FIFO;
- карусель;
- адаптивный.

Каждый процесс в системе может выполняться, используя любой из этих методов. Они

действуют применительно к каждому отдельному процессу, а не применительно ко всем процессам на узле.

Помните, что эти методы диспетчеризации применимы, только когда два или более процесса с одинаковым приоритетом находятся в состоянии READY (т.е. эти процессы непосредственно конкурируют друг с другом). Если процесс с более высоким приоритетом переходит в состояние READY, то он немедленно вытесняет все процессы с более низким приоритетом.

На следующей диаграмме три процесса с одинаковым приоритетом находятся в состоянии READY. Если процесс A блокируется, то выполняется процесс B.



Процесс A блокируется, процесс B выполняется

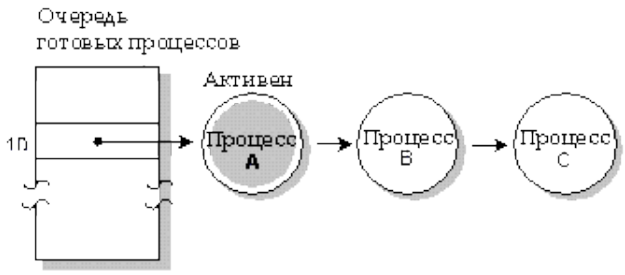
Метод диспетчеризации процесса наследуется от его родительского процесса, однако, затем этот метод может быть изменен.

Если вы хотите:	**Используйте функцию языка Си:**
Определить метод диспетчеризации для процесса	*getscheduler()*
Установить метод диспетчеризации для процесса	*setscheduler()*

FIFO диспетчеризация

При FIFO диспетчеризации процесс продолжает выполнение пока не наступит момент, когда он:

- добровольно уступает управление (т.е. выполняет *любой* вызов ядра);
- вытесняется процессом с более высоким приоритетом.



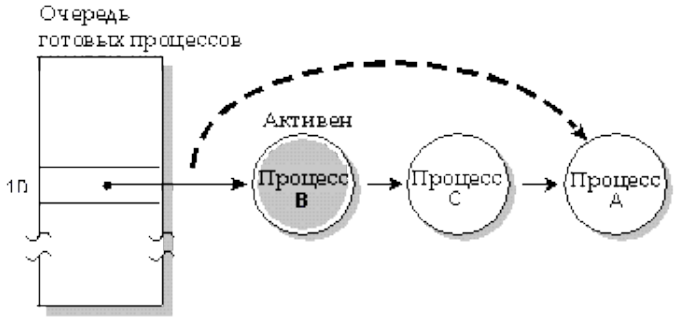
FIFO диспетчеризация. Процесс A выполняется до тех пор, пока не блокируется

Два процесса, которые выполняются с одним и тем же приоритетом, могут использовать метод FIFO для организации взаимоисключающего доступа к разделяемому (т.е. совместно используемому) ресурсу. Ни один из них не будет вытеснен другим во время своего выполнения. Так, например, если они совместно используют сегмент памяти, то каждый из этих двух процессов может обновлять данные в этом сегменте, не прибегая к использованию какого-либо способа синхронизации (например, семафора).

Карусельная диспетчеризация

При карусельной диспетчеризации процесс продолжает выполнение, пока не наступит момент, когда он:

добровольно уступает управление (т.е. блокируется);
вытесняется процессом с более высоким приоритетом;
использовал свой квант времени (*timeslice*).



Карусельная диспетчеризация. Процесс A выполняется до тех пор, пока он не использовал свой квант времени; затем выполняется следующий процесс, находящийся в состоянии READY (процесс B)

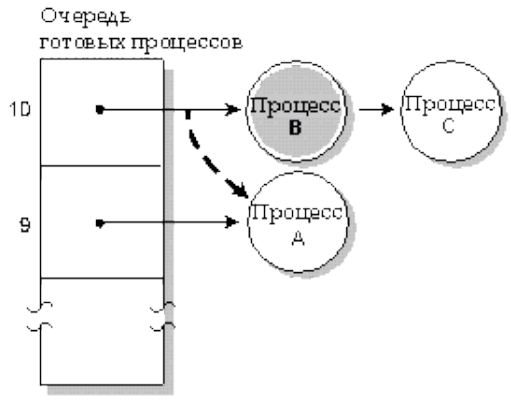
Квант времени - это интервал времени, выделяемый каждому процессу. После того, как процесс использовал свой квант времени, управление передается следующему процессу, который находится в состоянии READY и имеет такой же уровень приоритета. Квант времени равен 50 миллисекундам.

Примечание. За исключением квантования времени, карусельная диспетчеризация идентична FIFO-диспетчеризации.

Адаптивная диспетчеризация

При адаптивной диспетчеризации процесс ведет себя следующим образом:

- Если процесс использовал свой квант времени (т.е. он не блокировался), то его приоритет уменьшается на 1. Это получило название *снижение приоритета* (priority decay). Учтите, что "пониженный" процесс не будет продолжать "снижаться", даже если он использовал еще один квант времени и не блокировался - он снизится только на один уровень ниже своего исходного приоритета.
- Если процесс блокируется, то ему возвращается первоначальное значение приоритета.



Адаптивная диспетчеризация. Процесс A использовал свой квант времени; его приоритет снизился на 1. Выполняется следующий процесс в состоянии READY (процесс B)

Вы можете использовать адаптивную диспетчеризацию в тех случаях, когда процессы, производящие интенсивные вычисления, выполняются в фоновом режиме одновременно с интерактивной работой пользователей. Вы обнаружите, что адаптивная диспетчеризация дает производящим интенсивные вычисления процессам достаточный доступ к ЦП и в то же время сохраняет быстрый интерактивный отклик для других процессов.

Адаптивная диспетчеризация является методом диспетчеризации, использующимся по умолчанию для программ, запускаемых командным интерпретатором.

Клиент-управляемый приоритет

В QNX обмен данными между процессами в большинстве случаев организован с использованием модели *клиент/сервер*. *Серверы* выполняют некоторые сервисные функции, а *клиенты*, посылая сообщение серверу, запрашивают эти услуги. Как правило, серверы более надежны и жизнеспособны, чем клиенты.

Количество клиентов обычно больше, чем серверов. Как следствие этого, принято запускать сервер с приоритетом более высоким, чем у любого из его клиентов. Может использоваться любой из трех рассмотренных выше методов диспетчеризации, но, наверное, самым распространенным является карусельный.

Если клиент с низким уровнем приоритета посылает сообщение серверу, то его запрос выполняется под более высоким уровнем приоритета, тем, который имеется у сервера. Это

косвенным образом повышает приоритет клиента, т.к. именно его запрос заставил сервер выполняться.

Пока для выполнения запроса серверу достаточно короткого промежутка времени, проблем обычно не возникает. Если же выполнение запроса занимает у сервера более продолжительное время, то клиент с низким приоритетом может неблагоприятно повлиять на выполнение других процессов, приоритеты которых выше, чем у клиента, но ниже, чем у сервера.

Чтобы решить эту дилемму, сервер может поставить свой приоритет в зависимость от приоритета того клиента, чей запрос он выполняет. Когда сервер получает сообщение, приоритет будет установлен таким же, как у клиента. Обратите внимание, что изменился только приоритет сервера - его алгоритм диспетчеризации остается неизменным. Если во время выполнения запроса сервер получает другое сообщение, и приоритет нового клиента выше, чем у сервера, то приоритет сервера повышается. Фактически, новый клиент "заряжает" сервер до своего уровня приоритета, позволяя ему закончить выполнение текущего запроса и приступить к обработке запроса нового клиента. Если этого не делать, то фактически приоритет нового клиента понизится, пока он блокирован на сервере с низким приоритетом.

Если вы выбрали клиент-управляемый приоритет для своего сервера, вам следует также запросить доставку сообщений в порядке убывания приоритетов (в противоположность хронологическому).

Чтобы установить клиент-управляемый приоритет, используйте функцию Си *qnx_pflags()* следующим образом:

```
qnx_pflags(~0, _PPF_PRIORITY_FLOAT | _PPF_PRIORITY_REC, 0, 0);
```

Несколько слов о реальном времени

Как бы мы этого не хотели, компьютеры не являются бесконечно быстрыми. Для системы реального времени жизненно важно, чтобы такты работы ЦП не расходовались зря. Также очень важно свести к минимуму время, которое проходит с момента наступления внешнего события до начала выполнения кода программы, ответственной за обработку данного события. Это время называется *задержкой*.

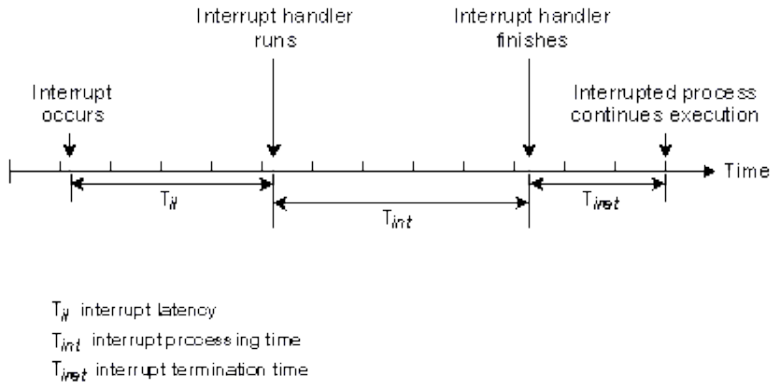
В QNX системе встречается несколько видов задержек.

Задержка прерывания

Задержка прерывания - это интервал времени между аппаратным прерыванием и выполнением первой команды программного обработчика прерывания. QNX практически все время оставляет прерывания полностью разрешенными, поэтому задержка прерывания, как правило, не существенна. Однако некоторые критические фрагменты кода требуют, чтобы прерывания были временно запрещены. Максимальная продолжительность такого запрета обычно определяет худший случай задержки прерывания - в QNX это очень небольшая величина.

Следующая диаграмма иллюстрирует случай, когда аппаратное прерывание обрабатывается в

установленном обработчиком прерывании. Обработчик прерывания либо просто выполнит команду возврата, либо при возврате запустит прокси.



Обработчик прерывания просто завершается

Задержка прерывания (T_{il}) на приведенной выше диаграмме отражает *минимальную* задержку - случай, когда прерывания были полностью разрешены в момент, когда произошло прерывание. В худшем случае задержка прерывания составит это время *плюс* наибольшее время, в течение которого QNX или выполняющийся процесс запрещает прерывания ЦП.

T_{il} для различных процессоров

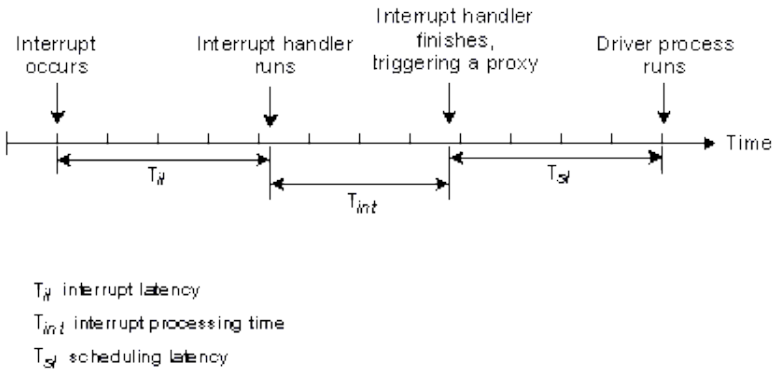
Следующая таблица содержит типичные значения задержки прерывания для разных процессоров:

Задержка прерывания (T_{il}):	**Процессор:**
3.3 микросекунды	166 МГц Pentium
4.4 микросекунды	100 МГц Pentium
5.6 микросекунды	100 МГц 486DX4
22.5 микросекунды	33 МГц 386EX

Задержка диспетчеризации

В некоторых случаях обработчик аппаратного прерывания низкого уровня должен передать управление процессу более высокого уровня. В этом случае обработчик прерывания перед выполнением команды "возврат" запускает прокси. Здесь имеет место второй вид задержки - *задержка диспетчеризации*, - с которой также надо считаться.

Задержка диспетчеризации - это время между завершением работы обработчика прерывания и выполнением первой команды процесса-драйвера. Это время, необходимое для сохранения контекста, выполняющегося в данный момент времени процесса, и восстановления контекста требуемого драйвера. В QNX это время также невелико, хотя и больше задержки прерывания.



Обработчик прерывания завершает работу и запускает прокси

Важно отметить, что обработка *большинства* прерываний завершается без запуска прокси. В большинстве случаев обработчик прерывания сам может выполнить все необходимые действия. Запуск прокси, чтобы "разбудить" *драйвер*, процесс более высокого уровня, происходит только при наступлении важного события. Например, обработчик прерывания для драйвера последовательного порта при каждом прерывании "регистр передачи свободен" будет передавать один байт данных и запустит процесс более высокого уровня (Dev) только тогда, когда выходной буфер, наконец, опустеет.

T_{sl} для различных процессоров

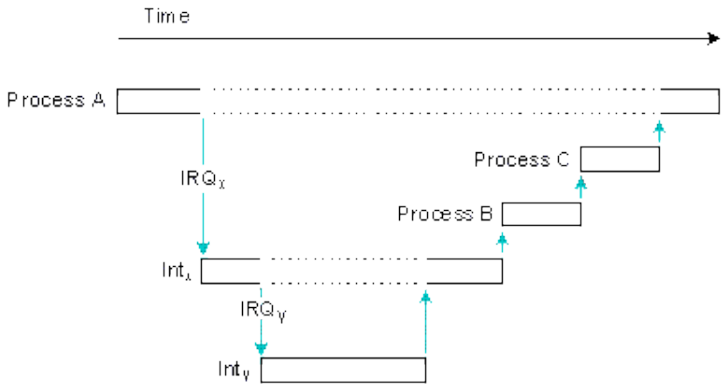
Следующая таблица содержит типичные значения задержки диспетчеризации (T_{sl}) для разных процессоров:

Задержка диспетчеризации (T_{sl}):	**Процессор:**
4.7 микросекунды	166 МГц Pentium
6.7 микросекунды	100 МГц Pentium
11.1 микросекунды	100 МГц 486DX4
74.2 микросекунды	МГц 386EX

Стек прерываний

Так как архитектура микрокомпьютеров позволяет назначать приоритеты аппаратным прерываниям, то прерывания с более высоким приоритетом могут вытеснять прерывания с меньшим приоритетом.

Этот механизм полностью поддерживается в QNX. Выше были рассмотрены простейшие - и наиболее типичные - ситуации, когда происходит только одно прерывание. Практически такой же расчет времени справедлив для прерывания с наивысшим приоритетом. При рассмотрении наихудшего случая для прерывания с низким приоритетом необходимо учитывать время обработки всех прерываний более высокого уровня, т.к. в QNX прерывание с более высоким приоритетом вытеснит прерывание с меньшим приоритетом.



Выполняется процесс A. Прерывание IRQ_x вызывает выполнение обработчика прерывания Int_x, который вытесняется IRQ_y и его обработчиком Int_y. Int_y запускает прокси, вызывающее выполнение процесса B, а Int_x запускает прокси, вызывающее выполнение процесса C

Глава 3. Менеджер процессов

Эта глава охватывает следующие темы:

- Введение
- Жизненный цикл процесса
- Состояния процесса
- Символьные имена процессов
- Таймеры
- Обработчики прерываний

Введение

Обязанности Менеджера процессов

Менеджер процессов тесно взаимодействует с Микроядром, чтобы обеспечить услуги, составляющие сущность операционной системы. Хотя он и является единственным процессом, который использует то же адресное пространство, что и Микроядро, Менеджер процессов выполняется как истинный процесс. И он, как и все остальные процессы, подвергается диспетчеризации со стороны Ядра и использует предоставляемые Микроядром примитивы передачи сообщений для связи с другими процессами в системе.

Менеджер процессов отвечает за создание новых процессов в системе и за управление основными ресурсами, связанными с процессом. Все эти услуги предоставляются посредством сообщений. Так, например, если процесс хочет породить новый процесс, он делает это, посылая сообщение с указанием атрибутов создаваемого процесса. Обратите внимание, что т.к. сообщения передаются по сети, вы можете легко создать процесс на другом узле сети, послав сообщение Менеджеру процессов на этом узле.

Примитивы создания процессов

QNX поддерживают три примитива создания процесса:

- *fork();*
- *exec();*
- *spawn().*

Примитивы *fork()* и *exec()* определены стандартом POSIX, а примитив *spawn()* реализован только в QNX.

Примитив *fork()*

Примитив *fork()* создает новый процесс, который является точной копией вызвавшего его процесса. Новый процесс использует тот же самый код, что и породивший его процесс, и наследует копию всех данных родительского процесса.

Примитив *exec()*

Примитив *exec()* заменяет вызвавший процесс новым. После успешного вызова *exec()* возврата не происходит, т.к. образ вызывающего процесса заменяется образом нового процесса. Обычной практикой в POSIX-системах для создания нового процесса - без удаления вызывающего процесса - является сначала вызов *fork()*, а затем вызов *exec()* из порожденного процесса.

Примитив *spawn()*

Примитив *spawn()* создает новый процесс как потомок вызывающего процесса. С его помощью можно избежать вызовов *fork()* и *exec()*, используя более быстрый и эффективный способ создания новых процессов. В отличие от *fork()* и *exec()*, которые по своей природе выполняются на том же самом узле, что и вызывающий процесс, примитив *spawn()* может создавать процессы *на любом узле сети*.

Наследование

Когда с помощью одного из трех рассмотренных выше примитивов задается новый процесс, он наследует многое из своего окружения от родителя. Это сведено в следующую таблицу:

Наследуемый параметр	***fork()***	***exec()***	***spawn()***
Идентификатор процесса	нет	да	нет
Открытые файлы	да	по выбору*	по выбору
Блокировка файлов	нет	да	нет
Ожидающие сигналы	нет	да	нет
Маска сигналов	да	по выбору	по выбору
Игнорируемые сигналы	да	по выбору	по выбору
Обработчики сигналов	да	нет	нет
Переменные окружения	да	по выбору	по выбору
Идентификатор сеанса	да	да	по выбору
Группа процесса	да	да	по выбору
Реальные UID, GID	да	да	да
Эффективные UID, GID	да	по выбору	по выбору
Текущий рабочий каталог	да	по выбору	по выбору
Маска создания файлов	да	да	да
Приоритет	да	по выбору	по выбору
Алгоритм диспетчеризации	да	по выбору	по выбору
Виртуальные	нет	нет	нет

Наследуемый параметр	***fork()***	***exec()***	***spawn()***
Идентификатор процесса	нет	да	нет
каналы			
Символьные имена	нет	нет	нет
Таймеры реального времени	нет	нет	нет

*по выбору: вызывающий процесс может по необходимости выбрать - да или нет.

Жизненный цикл процесса

Процесс проходит через четыре стадии:

1. Создание
2. Загрузка
3. Выполнение
4. Завершение

Создание

Создание нового процесса состоит из присвоения новому процессу идентификатора (ID) процесса и подготовки информации, которая определяет окружение нового процесса. Большая часть этой информации наследуется от родительского процесса (смотри раздел "**Наследование**").

Загрузка

Загрузка образа процесса производится *нитью загрузчика*. Код загрузчика находится в Менеджере процессов, но нить выполняется под ID нового процесса. Это позволяет Менеджеру процессов обрабатывать и другие запросы во время загрузки программы.

Выполнение

После того, как код программы загружен, процесс готов к выполнению; он начинает конкурировать с остальными процессами за ресурсы ЦП. Заметьте, что все процессы выполняются параллельно со своими родителями. Кроме того, смерть родительского процесса *не* означает автоматическую смерть его дочерних процессов.

Завершение

Процесс завершается одним из двух способов:

- процесс получает сигнал, который вызывает завершение процесса;
- процесс вызывает функцию Си *exit()*, либо в явном виде, либо в качестве действия по умолчанию при возврате из функции *main()*.
-

Завершение включает две стадии:

1. Выполняется *нить завершения* в Менеджере процессов. Этот "заслуживающий доверия" код расположен в Менеджере процессов, но нить выполняется с ID завершающегося процесса. Эта нить закрывает все открытые файловые дескрипторы и освобождает следующие ресурсы:
   - все виртуальные каналы, принадлежащие процессу;
   - всю память, выделенную процессу;
   - все символьные имена;
   - любые старшие номера устройств (только менеджеры ввода/вывода);
   - любые обработчики прерывания;
   - любые прокси;
   - любые таймеры.
2. После того, как нить завершения выполнена, извещение о завершении процесса посылается родительскому процессу (эта фаза выполняется внутри Менеджера процессов).

Если родительский процесс не вызвал *wait()* или *waitpid()*, то дочерний процесс становится "зомби" и не будет завершен, пока родительский процесс не вызовет *wait()* или не завершит выполнение. (Если вы не хотите, чтобы процесс *ждал* смерти дочерних процессов, вы должны либо установить _SPAWN_NOZOMBIE флаг функциями *qnx_spawn()* или *qnx_spawn_options()*, либо установить действие SIG_IGN для сигнала SIGCHLD посредством функции *signal()*. Таким образом, дочерние процессы не будут становиться зомби после смерти.)

Родительский процесс может ждать смерти дочернего процесса, запущенного на удаленном узле. Если родитель процесса-зомби умирает, то зомби освобождается.

Если запущена утилита dumper и процесс завершается в результате получения сигнала, то генерируется дамп памяти. Вы можете затем исследовать его с помощью отладчика.

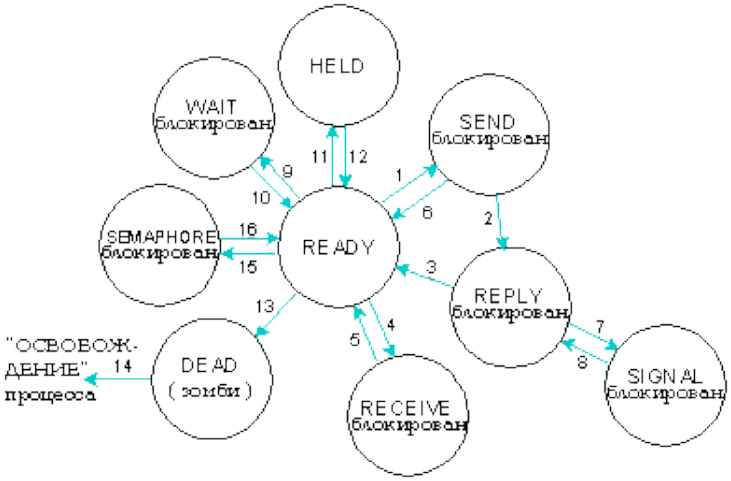
Состояния процесса

Процесс всегда находится в одном из следующих состояний:

1. READY (готов) - процесс способен использовать ЦП (т.е. он не ждет наступления какого-либо события).
2. BLOCKED (блокирован) - процесс в одном из следующих блокированных состояний:
   - SEND-блокирован;
   - RECEIVE-блокирован;
   - REPLY-блокирован;
   - SIGNAL-блокирован;
   - SEMAPHORE-блокирован.
3. HELD (приостановлен) - процесс получил сигнал SIGSTOP. В этом состоянии процесс не имеет права использовать ЦП; выйти из состояния HELD процесс может только в результате получения сигнала SIGCONT, или завершив свою работу после получения другого сигнала.
4. WAIT (блокирован) - процесс выполнил вызов функции *wait()* или *waitpid()* для ожидания сообщения о завершении выполнения дочернего процесса.
5. DEAD (мертв) - процесс завершил выполнение, но не может послать сообщения об

этом родительскому процессу, т.к. родительский процесс не вызвал *wait()* или *waitpid()*. Когда процесс находится в состоянии DEAD, ранее занимаемая им память уже освобождена. Процесс в состоянии DEAD также называют *зомби*.

Примечание. Для получения более полной информации о блокированном состоянии обратитесь к главе "**Микроядро**".



Возможные состояния процесса в QNX

Показаны следующие переходы:

1. Процесс посылает сообщение
2. Процесс-адресат получает сообщение
3. Процесс-адресат отвечает на сообщение
4. Процесс ожидает сообщения
5. Процесс получает сообщение
6. Сигнал разблокирует процесс
7. Сигнал пытается разблокировать процесс; адресат ранее запросил извещение о сигнале путем посылки сообщения
8. Процесс-адресат получает сообщение о сигнале
9. Процесс ожидает смерти дочернего процесса
10. Дочерний процесс умирает, либо сигнал разблокирует процесс
11. Процесс получает SIGSTOP
12. Процесс получает SIGCONT
13. Процесс умирает
14. Родительский процесс вызывает функцию *wait()* или *waitpid()*, завершает ее выполнение, либо ранее уже завершил выполнение
15. Процесс вызывает функцию *semwait()* для семафора с неположительным значением
16. Другой процесс вызывает функцию *sempost()*, или приходит немаскированный сигнал

Определение состояний процесса

Чтобы определить состояние конкретного процесса из командного интерпретатора (Shell),

используйте утилиты ps и sin (внутри приложений используйте функцию *qnx_psinfo()*).

Чтобы определить состояние операционной системы в целом из командного интерпретатора (Shell), используйте утилиту sin (внутри приложений используйте функцию *qnx_osinfo()*).

Утилита ps определена стандартом POSIX; ее использование в командных файлах *может быть* переносимым в другие системы. Утилита sin, напротив, уникальна для QNX; она дает вам полезную информацию, специфическую для QNX, которую вы не можете получить, используя утилиту ps.

Символьные имена процессов

QNX стимулирует разработку приложений, которые разбиты на несколько взаимодействующих процессов. Такое приложение, которое существует как группа взаимодействующих процессов, имеет большие возможности распараллеливания и может быть распределено по сети для лучшей производительности.

Однако разделение приложений на взаимодействующие процессы требует ряда специальных соображений. Чтобы взаимодействующие процессы могли установить связь между собой, они должны иметь возможность определить идентификаторы друг друга. Например, допустим, что имеется сервер базы данных, который может обслуживать произвольное количество клиентов. Клиенты могут запускаться и прекращать работу в любое время, однако сервер всегда остается доступным. Как процессы-клиенты узнают идентификатор процесса-сервера базы данных с тем, чтобы послать ему сообщение?

В QNX процессы могут присваивать себе символьные имена. В контексте одного узла процесса могут зарегистрировать эти имена у Менеджера процессов на том узле, на котором они выполняются. Остальные процессы могут затем запросить у Менеджера процессов идентификатор процесса, ассоциированный с определенным именем.

Ситуация становится более сложной, если мы рассмотрим сеть, в которой сервер обслуживает клиентов, находящихся на различных узлах. Поэтому в QNX предусмотрена поддержка как глобальных, так и локальных имен. Глобальные имена определены для всей сети, в то время как локальные имена определены только для того узла, на котором они зарегистрированы. Глобальные имена начинаются с косой черты (/). Например:

qnx/fsys	локальное имя
company/xyz	локальное имя
/company/xyz	локальное имя

Примечание. Мы рекомендуем вам использовать название вашей компании в качестве префикса для всех регистрируемых имен, чтобы избежать конфликтов имен между программами разных производителей.

Чтобы сделать возможным использование глобальных имен, хотя бы на одном узле сети должен быть запущен процесс, известный как *определитель имен* (утилита nameloc). Этот процесс ведет учет всех зарегистрированных глобальных имен.

В каждый момент времени в сети могут быть запущены до десяти определителей имен. Каждый хранит идентичные копии всех активных глобальных имен. Такая избыточность

гарантирует нормальное функционирование сети, даже если один или несколько узлов, обеспечивающих определение имен, выйдут из строя одновременно.

Чтобы зарегистрировать имя, процесс *сервер* использует функцию Си *qnx_name_attach()*. Чтобы определить процесс по имени, процесс *клиент* использует функцию Си *qnx_name_locate()*.

Таймеры

Исчисление времени

В QNX исчисление времени основывается на системном таймере, поддерживаемом операционной системой. Таймер содержит текущее значение Координированного Всемирного Времени (UTC) относительно 0 часов, 0 минут, 0 секунд, 1 января 1970 года. Чтобы определить местное время, функции исчисления времени используют переменную окружения **TZ** (которая описывается в книге "Руководство пользователя").

Простые средства отсчета времени

Командные файлы и процессы могут делать паузу на определенное количество секунд, используя простые средства отсчета времени. В командных файлах для этой цели используется утилита sleep; в процессах используется функция Си *sleep()*. Также может использоваться функция *delay()*, которой в качестве аргумента передается длина интервала времени в миллисекундах.

Более сложное средство отсчета времени

Кроме того, процесс может создавать таймеры, устанавливать их на определенный интервал времени и удалять таймер. Эти средства отсчета времени основываются на спецификации POSIX Std 1003.4/Draft 9.

Создание таймеров

Процесс может создать один или больше таймеров. Эти таймеры могут быть любых типов и в любом сочетании. С учетом конфигурируемого ограничения общего числа таймеров, поддерживаемых операционной системой (смотри Proc в "Описание утилит"). Для создания таймера используйте функцию *timer_create()*.

Установка таймеров

При установке таймеров вы можете использовать один из двух типов интервалов времени:

- **абсолютный** - время относительно 0 часов, 0 минут, 0 секунд, 1 января 1970 года;
- **относительный** - время относительно текущего показания часов.

Вы также можете создать таймер, периодически срабатывающий с заданным интервалом времени. Например, вы установили таймер таким образом, чтобы он сработал завтра в 9 часов утра. Вы можете задать, чтобы после этого он срабатывал каждые 5 минут.

Вы также можете установить временной интервал для существующего таймера. Результат будет зависеть от типа интервала времени:

- для абсолютного таймера новый интервал *заменяет* текущий интервал времени;
- для относительного таймера новый интервал *добавляется* к любому остающемуся интервалу времени.

Чтобы установить таймер:	**Используйте функцию:**
Абсолютный или относительный интервал	*timer_settime()*

Удаление таймеров

Чтобы удалить таймер, используйте функцию Си *timer_delete()*.

Установка разрешения таймера

Вы можете установить разрешение для таймера с помощью утилиты ticksize или функции *qnx_timerperiod()*. Вы можете настраивать разрешение от 500 микросекунд до 50 миллисекунд.

Чтение таймера

Чтобы узнать оставшийся от таймера интервал или проверить, не был ли таймер удален, используйте функцию Си *timer_gettime()*.

Обработчики прерываний

Обработчики прерываний обслуживают систему аппаратных прерываний компьютера - они реагируют на аппаратные прерывания и осуществляют передачу данных между компьютером и внешними устройствами на низком уровне.

Обработчики прерываний физически являются частью стандартного процесса QNX (например, драйвер), но они всегда выполняются асинхронно по отношению к процессу, в который они включены.

Обработчик прерываний:

- получает управление посредством дальнего вызова, а не непосредственно от прерывания (это может быть написано на Си, а не на ассемблере);
- выполняется в контексте процесс, в который он включен, таким образом имеет доступ ко всем глобальным переменам этого процесса;
- выполняется при разрешенных прерываниях, но вытесняется только в случае, если происходит прерывание с более высоким приоритетом;
- не должен непосредственно управлять контроллером прерывания 8259 (об этом заботится операционная система);
- должен быть как можно короче.

Несколько процессов могут обрабатывать одно и то же прерывание (если это поддерживается аппаратно). Когда происходит физическое прерывание, все обработчики прерываний будут по очереди получать управление. Не должно делаться никаких предположений относительно порядка, в котором будут вызываться обработчики прерываний, разделяющие одно и то же прерывание.

Если вы хотите:	**Используйте функцию:**
Установить обработчик прерывания	*qnx_hint_attach()*
Удалить обработчик прерывания	*qnx_hint_detach()*

Обработчики прерывания таймера

Вы можете установить обработчик прерывания непосредственно для системного таймера таким образом, что обработчик будет вызываться по каждому прерыванию таймера. Чтобы установить период, вы можете использовать утилиту ticksize.

Вы можете также установить обработчик прерывания для отмасштабированного прерывания таймера, которое будет происходить каждые 50 миллисекунд, независимо от *tick size*. Эти таймеры предлагают низкоуровневую альтернативу POSIX 1003.4 таймерам.

Глава 4. Пространство имен ввода/вывода

Эта глава охватывает следующие темы:

- Введение
- Разборка имен путей
- Пространство дескрипторов файлов

Введение

Пространство имен ввода/вывода

Ресурсы ввода/вывода (I/O, от английского Input/Output) в QNX *не* встроены внутрь Микроядра. Процессы обслуживающие ввод/вывод запускаются динамически во время работы системы. Так как файловая система QNX является необязательным элементом, то пространство путей (полных имен файлов и каталогов) не встроено внутрь файловой системы, в отличие от большинства монолитных систем.

Префиксы и области полномочий

В QNX пространство имен путей разделено на области полномочий. Любой процесс, выполняющий файл-ориентированное обслуживание ввода/вывода, должен зарегистрировать у Менеджера процессов свой префикс, определяя часть пространства имен, которое он хочет контролировать (т.е. область своих полномочий). Эти префиксы составляют дерево префиксов, которое хранится в памяти на каждом компьютере под управлением QNX.

Разборка имен путей

Префиксы менеджера ввода/вывода

Когда процесс открывает файл, то полное имя файла сопоставляется с деревом префиксов с тем, чтобы отправить запрос *open()* к соответствующему менеджеру ресурсов ввода/вывода. Например, Менеджер устройств (Dev) обычно регистрирует префикс /dev. Если процесс вызывает *open()* с аргументом /dev/xxx, то совпадет префикс /dev и *open()* будет направлен к Dev (его владельцу).

Дерево префиксов может содержать частично перекрывающиеся области полномочий. В этом случае ищется наиболее длинное совпадение. Например, предположим, что есть три зарегистрированных префикса:

/	дисковая файловая система (Fsys)
/dev	система символьных устройств (Dev)
/dev/hd0	дисковый том без разделов (Fsys)

Менеджер файловой системы зарегистрировал два префикса, один для смонтированной файловой системы QNX (т.е. /) и один для блок-ориентированного файла, который представляет весь физический жесткий диск (т.е. /dev/hd0). Менеджер устройств

зарегистрировал единственный префикс. Следующая таблица иллюстрирует правило наиболее длинного совпадения при разборе имен путей.

Имя пути:	**Совпадает:**	**Относится к:**
/dev/con1	/dev	Dev
/dev/hd0	/dev/hd0	Fsys
/usr/dtdodge/test	/	Fsys

Дерево префиксов хранится как список префиксов, разделенных двоеточиями, следующим образом:

```
''prefix''=''pid'',''unit'':''prefix''=''pid'',
 ''unit'':''prefix''=''pid'',''unit''
```

где *pid* - это ID процесса менеджера ресурсов I/O, а *unit* - это один символ, используемый менеджером для нумерации префиксов, которыми он владеет. В приведенном выше примере, если Fsys был бы процессом 3 и Dev был бы процессом 5, то дерево префиксов выглядело бы как:

```
/dev/hd0=3,a:/dev=5,a:/=3,e
```

Если вы хотите:	**Используйте:**
Показать дерево префиксов	Утилиту prefix
Получить дерево префиксов внутри программы на языке Си	Функцию *qnx_prefix_query()*

Сетевой корень

QNX поддерживают концепцию сетевого корня, которая позволяет сопоставлять имя пути с деревом префиксов на конкретном узле. Для обозначения узла имя пути начинается с *двух* косых черт, за которыми следует номер узла. Это также позволяет вам легко получать доступ к файлу и устройству, которые лежат за пределами пространства путей вашего узла. Например, в типичной сети QNX возможны следующие имена путей:

/dev/ser1	локальный последовательный порт
//10/dev/ser1	последовательный порт в узле 10
//0/dev/ser1	локальный последовательный порт
//20/usr/dtdodge/test	файл на узле 20

Заметьте, что //0 всегда указывает на локальный узел

Сетевой корень по умолчанию

Когда программа выполняется на удаленном узле, вы обычно хотите, чтобы она рассматривала имена путей в контексте пространства имен своего узла. Например, эта команда:

```
//5 ls /
```

которая запускает утилиту ls на узле 5, возвратит то же самое, что и:

```
ls /
```

запускаемая на своем узле. В обоих случаях "/" будет соотноситься с деревом префиксов на вашем узле, а не на узле 5. В противном случае, можно представить себе беспорядок, который мог бы возникнуть, если бы префикс "/" рассматривался своим как для узла 5, так и для своего узла: выбирались бы одновременно файлы из разных файловых систем!

Чтобы правильно интерпретировать имена путей, каждый процесс имеет *сетевой корень по умолчанию*, который определяет, дерево префиксов какого узла будет использоваться для разбора любых имен путей, начинающихся с одиночной косой черты ("/"). Когда разбирается имя пути, начинающееся с одиночной косой черты, то оно предваряется *сетевым корнем по умолчанию*. Например, если процесс имеет сетевой корень по умолчанию //9, тогда:

```
/usr/home/luc
```

будет разбираться, как:

```
//9/usr/home/luc
```

что может быть интерпретировано, как "разобрать путь /usr/home/luc, используя дерево префиксов 9-го узла".

Сетевой корень по умолчанию наследуется процессами при их создании и соответствует локальному узлу, на котором запускается система. Например, допустим, что вы работаете на узле 9, находясь в командном интерпретаторе, для которого сетевой корень по умолчанию установлен как узел 9 (весьма типичный случай). Если вы выполните команду:

```
ls /
```

команда унаследует сетевой корень по умолчанию //9, и в результате вы получите:

```
ls //9/
```

Подобно тому, если бы вы ввели команду:

```
//5 ls /
```

Вы запустили команду ls на узле 5, но она по-прежнему унаследует сетевой корень по умолчанию //9, поэтому в результате вы опять получите ls //9/. В обоих случаях имя пути разбирается относительно одного и того же пространства имен.

Если вы хотите:	**Используйте:**
Получить текущий сетевой корень по умолчанию	Функцию *qnx_prefix_getroot()*
Установить сетевой корень по	Функцию *qnx_prefix_setroot()*

Если вы хотите:	**Используйте:**
умолчанию	
Запустить программу с новым сетевым корнем по умолчанию	Утилиту on

Передача имен путей между процессами

Если запущено одновременно несколько процессов, они не обязательно имеют один и тот же сетевой корень по умолчанию, даже если они выполняются на одном и том же узле. К примеру, один из процессов мог унаследовать сетевой корень по умолчанию от родительского процесса на каком-либо другом узле сети, либо его сетевой корень по умолчанию мог быть в явном виде задан родительским процессом.

При передаче имени пути между процессами, чьи сетевые корни могут отличаться (например, передавая файл спулеру для печати), вы должны добавить сетевой корень в начало имени пути, прежде чем передать путь другому процессу. Если вы уверены, что посылающий процесс и получатель имеют один и тот же сетевой корень по умолчанию (или имя пути уже начинается с *//узел/*), то вы можете опустить этот шаг.

Псевдонимы префиксов

Мы рассмотрели префиксы, которые регистрируются менеджерами ресурсов I/O. Вторая форма префикса известна как "псевдоним префикса", это простая подстановка строки вместо искомого префикса. Псевдоним префикса имеет форму:

```
''prefix''=''строка-заменитель''
```

Например, вы работаете на машине, у которой нет локальной файловой системы (поэтому нет и процесса, который бы отвечал за "/"). Однако, файловая система имеется на другом узле (допустим, 10), и вы хотите ее использовать как "/". Вы можете сделать это с помощью следующего псевдонима префикса:

```
/=//10/
```

Это приведет к тому, что ведущая косая черта (/) будет заменяться на префикс //10/. Например, путь /usr/dtdodge/test будет заменен следующим:

```
  //10/usr/dtdodge/test
```

Это новое имя пути будет снова сравниваться с деревом префиксов; на этот раз будет использоваться уже дерево префиксов 10 узла, т.к. имя пути начинается с //10. Это приведет к Менеджеру файловой системы на узле 10, которому и будет направлен запрос, например, *open()*. С помощью всего лишь нескольких символов этот псевдоним позволил нам обращаться к удаленной файловой системе как к локальной.

Для того чтобы выполнить перенаправление, не обязательно запускать локальную файловую систему. Дерево префиксов для бездисковой рабочей станции может выглядеть примерно так:

```
/dev=5,a:/=//10/
```

При таком дереве префиксов номера путей, начинающихся с /dev, будут направляться к локальному менеджеру символьных устройств, в то время как запросы с любыми другими именами путей, будут направляться к удаленной файловой системе.

Создание специальных имен устройств

Вы также можете использовать псевдонимы создания специальных имен устройств. Например, если спулер печати запущен на узле 20, то вы можете создать для него локальный псевдоним следующим образом:

```
/dev/printer=//20/dev/spool
```

Любой запрос на открытие локального принтера /dev/printer будет направляться по сети к реальному спулеру. Аналогичным образом, если не существует локального дисковода для гибких дисков, псевдоним для удаленного дисковода на узле 20 может иметь вид:

```
/dev/fd0=//20/dev/fd0
```

В обоих рассмотренных выше случаях можно не использовать перенаправление с помощью псевдонима, а обращаться непосредственно к удаленному ресурсу следующим образом:

```
 //20/dev/spool  ИЛИ  //20/dev/fd0
```

Относительные пути

Путь не обязательно должен начинаться с одной или двух косых черт. В таких случаях путь считается *относительным* по отношению к текущему рабочему каталогу. QNX хранит текущий рабочий каталог в виде строки символов. Относительные пути преобразуются в полные сетевые имена пути добавлением текущего рабочего каталога в начало относительного пути.

Учтите, что результат зависит от того, начинается ли текущий рабочий каталог с одинарной косой черты, либо с сетевого корня.

Текущий рабочий каталог

Если текущий рабочий каталог начинается с двойной косой черты (сетевого корня), он называется *определенным* и привязан к пространству имен указанного узла. В противном случае, если он начинается с одинарной косой черты, то в начало добавляется сетевой корень по умолчанию.

Например, эта команда:

```
cd //18/
```

является примером первой (определенной) формы и привязывает последующие разборы относительных путей к узлу 18, независимо от того, какое значение сетевого корня по умолчанию. Выполнив затем команду cd dev, вы попадете в //18/dev.

С другой стороны, такая команда:

```
cd /
```

является примером второй формы, когда сетевой корень по умолчанию влияет на разбор относительного пути. Например, если сетевой корень по умолчанию //9, тогда, выполнив команду cd dev, вы попадете в //9/dev. Так как текущий рабочий каталог не начинается с указания узла, то сетевой корень по умолчанию добавляется для получения полного сетевого пути.

Это на самом деле не так сложно, как может показаться. Обычно сетевые корни (//*узел*/) *не* указываются, и все, что вы делаете, просто работает внутри пространства имен вашего узла (определяемого вашим сетевым корнем по умолчанию). Большинство пользователей, войдя в систему, принимают нормальный сетевой корень по умолчанию (т.е. пространство имен их собственного узла) и работают внутри этой среды.

Замечание о cd

В некоторых традиционных UNIX системах команда cd (сменить директорию) модифицирует заданный путь, если он содержит символьные ссылки. Как результат, путь к новому текущему рабочему каталогу - который можно посмотреть командой pwd, может отличаться от того, который был задан в команде cd.

В QNX, однако, cd не изменяет путь - за исключением сокращенных ссылок "..". Например:

```
cd /usr/home/luc/test/../doc
```

установит текущий рабочий каталог /usr/home/luc/doc, даже если некоторые элементы этого пути являются символическими связями.

Более подробную информацию о символических связях можно получить в главе **"Менеджер файловой системы"**.

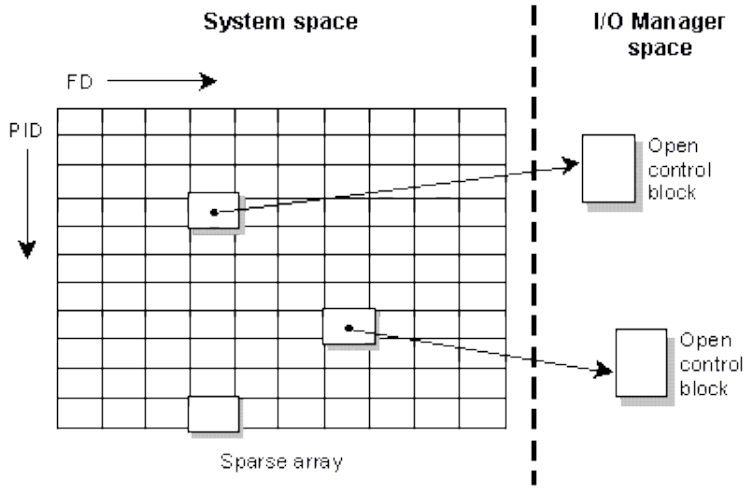
Примечание. Чтобы получить полное сетевое имя пути, используйте утилиту fullpath.

Пространство дескрипторов файлов

При открытии ресурса I/O, *open()* возвращает целое число, называемое *дескриптор файла* (FD), которое используется затем, чтобы направлять все последующие запросы I/O к данному менеджеру. (Внутри библиотечных процедур для направления опроса используется вызов ядра *Sendfd()*.)

Пространство дескрипторов файлов, в отличие от пространства имен пути, полностью локально для каждого процесса. Менеджер использует сочетание PID и FD, чтобы определить управляющую структуру, созданную предыдущим вызовом *open()*. Эта структура называется *блок управления файлом* (OCB от английского open control block) и хранится внутри менеджера I/O.

Следующая диаграмма показывает, как менеджер I/O находит соответствующий OCB для пары PID и FD.



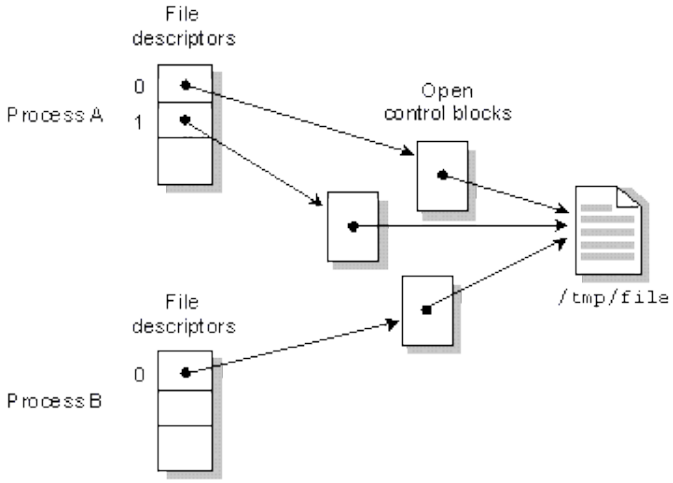
PID и FD определяют OCB для Менеджера ввода/вывода

Блоки управления файлами

OCB содержит действующую информацию об открытом ресурсе. Например, файловая система хранит здесь текущий указатель на позицию в файле. Каждый вызов *open()* создает новый OCB. Таким образом, если процесс дважды открывает один и тот же файл, любые вызовы *lseek()*, использующие один FD, не повлияют на текущую позицию для другого FD.

То же самое справедливо и для различных процессов, открывающих один и тот же файл.

Следующая диаграмма показывает два процесса, один из которых открывает файл дважды, а другой открывает этот же файл один раз. При этом нет разделяемых FD.



Процесс A открывает файл /tmp/file дважды. Процесс B открывает тот же файл один раз

Несколько дескрипторов файла в одном или нескольких процессах могут указывать на один и тот же OCB. Это может быть достигнуто двумя способами:

- процесс может использовать функции Си *dup()*, *dup2()* или *fcntl()* для создания

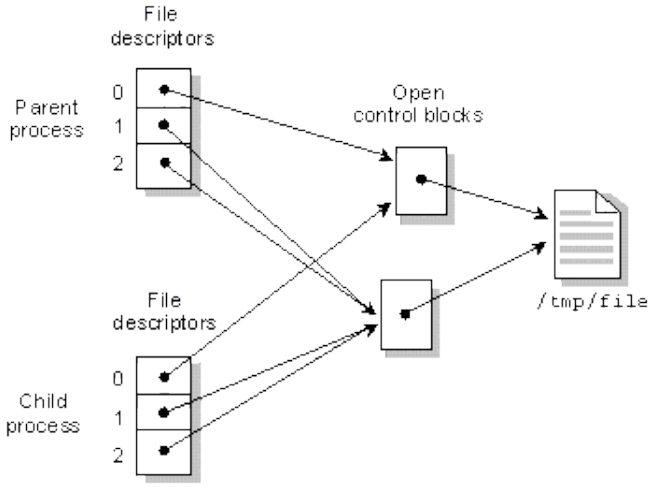
дубликата дескриптора файла, который указывает на тот же самый OCB;

- при создании нового процесса посредством *fork()*, *spawn()* или *exec()*, все дескрипторы открытых файлов по умолчанию наследуются новым процессом; эти наследуемые дескрипторы указывают на те же самые OCB, что и соответствующие дескрипторы файла в родительском процессе.

Когда несколько FD указывают на один и тот же OCB, тогда любое изменение в состоянии OCB немедленно видимо для всех процессов, у которых есть дескрипторы файлов, указывающие на этот OCB.

Например, если один из процессов использует функцию *lseek()* для изменения текущей позиции, тогда чтение или запись начинается с новой позиции, независимо от того, какой из дескрипторов файла используется.

Следующая диаграмма показывает два процесса, из которых один открывает файл дважды, а затем вызывает *dup()*, чтобы получить третий дескриптор. Затем он создает дочерний процесс, который наследует все открытые файлы.



Процесс открывает файл дважды, а затем получает третий FD посредством dup(). *Его дочерний процесс унаследует все эти три дескриптора файла*

Вы можете предотвратить наследование дескриптора файла при использовании *spawn()* или *exec()*, предварительно вызвав функцию *fcntl()* и установив флаг FD_CLOEXEC.

Глава 5. Менеджер файловой системы

Эта глава охватывает следующие темы:

- Введение
- Что такое файл?
- Регулярные файлы и каталоги
- Связи и индексные дескрипторы (inodes)
- Символические связи
- Программные каналы (pipes) и FIFO
- Производительность Менеджера файловой системы
- Надежность файловой системы
- Работа с дисками
- Ключевые компоненты раздела QNX
- Менеджер файловой системы DOS
- Файловая система CD-ROM
- Файловая система флэш
- Файловая система NFS
- Файловая система SMB

Введение

Менеджер файловой системы (Fsys) обеспечивает стандартизованные способы сохранения данных на дисках и доступа к ним. Fsys отвечает за обработку всех запросов на открытие, закрытие, чтение и запись файлов.

Что такое файл?

В QNX *файл* - это объект, в который может производиться запись, из которого может производиться чтение, либо и то и другое. QNX поддерживает шесть типов файлов; пять из них поддерживает Fsys:

- **регулярные файлы** - состоят из последовательности байт с произвольным доступом и не имеют предопределенной структуры;
- **каталоги** - содержат информацию, необходимую для поиска регулярных файлов; также содержат статус и атрибуты для каждого регулярного файла;
- **символические связи** - содержат путь к файлу или каталогу, к которым перенаправляются все запросы; символические связи часто используются для предоставления множества путей к одному файлу;
- **программные каналы** (pipes) и **FIFO** - служат как каналы ввода/вывода между взаимодействующими процессами;
- **блок-ориентированные файлы** - относятся к устройствам, таким, как диски, ленты и разделы дисков. Доступ к этим файлам обычно осуществляется таким образом, что аппаратные характеристики устройств скрыты от приложений.

Все эти типы файлов подробно описываются в этой главе. Шестой тип файлов - *блок-ориентированные файлы* - обслуживается Менеджером устройств.

Метки даты и времени

Fsys поддерживает четыре различных метки времени для каждого файла. Это:

- дата последнего доступа (чтения);
- дата последней записи;
- дата последней модификации;
- дата создания (уникальна для QNX).

Доступ к файлам

Доступ к регулярным файлам и каталогам управляется битами режима, хранящимися в *inode* (индексном дескрипторе) файла. Более подробно inode описан в секции "**Связи и индексные дескрипторы (inodes)**". Эти биты разрешают чтение, запись и выполнение в зависимости от эффективных ID пользователя и группы. При этом пользователи делятся на три категории:

- владелец файла;
- члены группы, к которой принадлежит владелец;
- остальные.

Процесс может выполняться с ID пользователя или ID группы файла, а не родительского процесса. Механизм, который позволяет это, называется *setuid* (установить ID пользователя) и *setgid* (установить ID группы).

Регулярные файлы и каталоги

Регулярные файлы

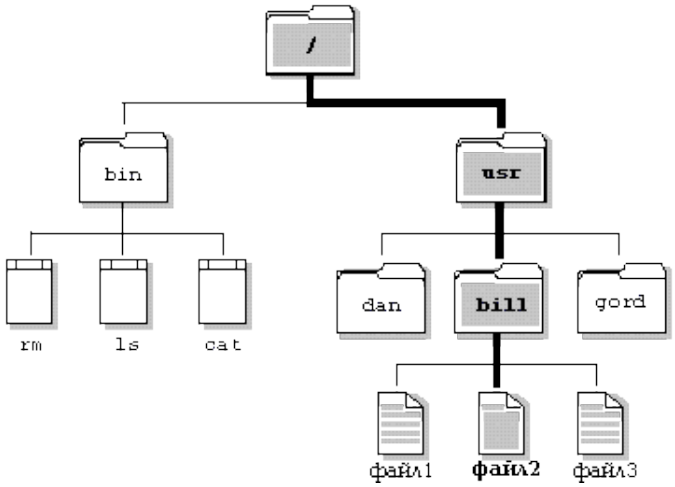
QNX рассматривает *регулярный файл* как последовательность байт с возможностью произвольного доступа и не имеющую другой предопределенной внутренней структуры. Прикладные программы сами несут ответственность за понимание структуры и содержания конкретного регулярного файла.

Регулярные файлы составляют большинство файлов в "файловых системах". Файловые системы поддерживаются Менеджером файловой системы и реализованы на базе блок-ориентированных файлов, соответствующих разделам диска (разделы описаны в секции "**Работа с дисками**").

Каталоги

Каталог - это файл, который содержит *элементы каталога*. Каждый элемент каталога увязывает *имя файла* с файлом. Имя файла - это символьное имя, которое позволяет идентифицировать файл и работать с ним. Файл может быть идентифицирован несколькими именами (смотри секции "**Связи и индексные дескрипторы (inodes)**" и "**Символические связи**").

Следующая диаграмма показывает, как производится поиск файла с именем /usr/bill/file2.



Путь в структуре каталога QNX к файлу /usr/bill/file2

Операции с каталогами

Хотя каталог ведет себя во многом как стандартный файл, Менеджер файловой системы накладывает некоторые ограничения на операции, которые вы можете производить с каталогом. В частности, вы не можете открыть каталог для записи либо создать связь для каталога с помощью функции Си *link()*.

Чтение элементов каталога

Для чтения элементов каталога вы можете использовать набор функций Си, определенных POSIX, которые обеспечивают не зависимый от ОС доступ к элементам каталога. Эти функции включают:

- *opendir()*
- *readdir()*
- *rewinddir()*
- *closedir()*

Так как каталоги QNX - это просто файлы, содержащие "известную" информацию, вы можете также читать элементы каталога непосредственно функциями Си *open()* и *read()*. Однако эта техника не переносима - формат элементов каталога отличается в различных операционных системах.

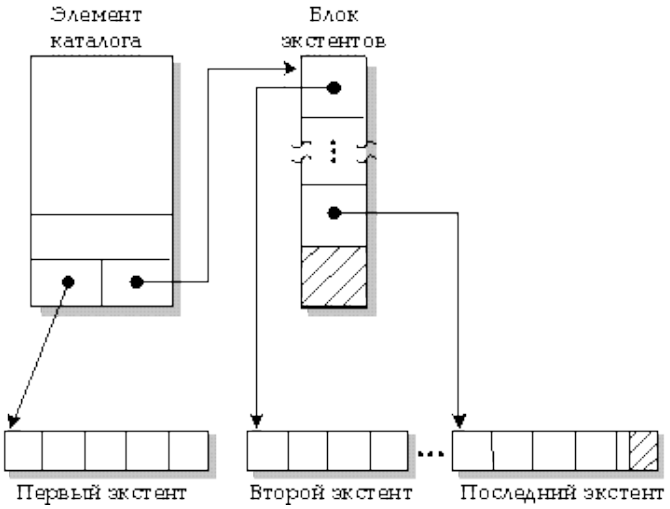
Экстенты

В QNX регулярные файлы и файлы каталога хранятся как последовательность *экстентов*. Экстент - это непрерывная последовательность блоков на диске.

Где хранятся экстенты

Файлы, которые состоят только из одного экстента, хранят информацию об экстенте в элементе каталога. Но, если файл состоит более чем из одного экстента, информация о

расположении экстентов хранится в одном или более *связных блоках экстентов* (связные - имеющие прямые/обратные указатели). Каждый блок экстентов может содержать информацию не более чем о 60 экстентах.



Файл, состоящий из множества непрерывных областей на диске, называемых в QNX экстентами

Увеличение файлов

Когда Менеджеру файловой системы необходимо увеличить файл, он сначала пытается увеличить последний экстент, хотя бы даже на один блок. Но если последний экстент не может быть дополнен, то для расширения файла выделяется новый экстент.

Для выделения новых экстентов Менеджер файловой системы использует метод "первого попадания". Специальная таблица в Менеджере файловой системы содержит сведения обо всех блоках, описанных в файле /.bitmap (этот файл описан в секции **"Ключевые компоненты раздела QNX"**). Для каждого блока указывается размер соответствующего ему свободного экстента. Менеджер файловой системы выбирает из таблицы первый достаточно большой экстент.

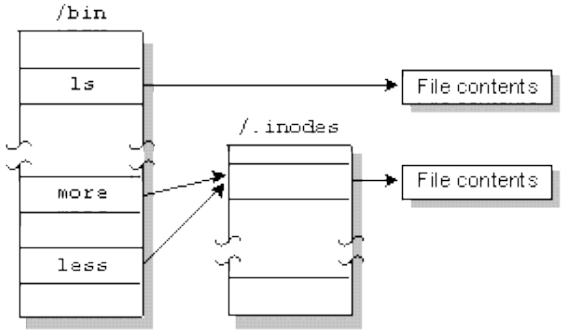
Связи и индексные дескрипторы (inodes)

В QNX файл может обозначаться более чем одним именем. Каждое имя файла называется *связью*. В действительности существует два вида связей: жесткие связи, или просто "связи", и символические связи. **Символические связи** описаны в следующей секции.

Для поддержки связей каждого файла, имя файла отделяется от остальной информации, описывающей файл. Эта информация хранится в структуре, называемой *inode* (индексным дескриптором).

Если файл имеет только одну связь (т.е. одно имя), то блок inode хранится в элементе каталога для этого файла. Но если файл имеет более чем одну связь, то inode хранится как запись в специальном файле /.inodes, а элемент каталога для файла содержит указатель на запись inode.

Учтите, что вы можете создать связь для файла, только если файл и связь находятся в одной и той же файловой системе.



Один и тот же файл обозначен двумя связями с именами "more" и "less"

Существует еще две ситуации, в которых для файла создается запись в файле /.inodes:

- если имя файла длиннее чем 16 символов, то информация inode хранится в файле /.inodes, оставляя место в элементе каталога для 48-символьного имени файла;
- если файл имел более одной связи и все связи, кроме одной, были удалены, то за файлом сохраняется отдельная запись в файле /.inodes. Это сделано, чтобы избежать поиска элемента каталога, указывающего на inode (элементы inode не имеют обратных связей с элементами каталога).

Если вы хотите:	**Используйте:**
Создать связь из командного интерпретатора	Утилиту ln
Создать связь из программы	Функцию *link()*

Удаление связей

При создании файла, для него устанавливается *счетчик связей*, равный единице. По мере добавления ссылок этот счетчик увеличивается; при удалении связи счетчик связей уменьшается. Файл не удаляется с диска до того, как счетчик связей станет равным нулю *и* все программы, использующие этот файл, закроют его. Это позволяет использовать открытый файл даже после того, как у него удалены все связи.

Если вы хотите:	**Используйте:**
Удалить связь из командного интерпретатора	Утилиту rm
Удалить связь из программы	Функции *remove()* или *unlink()*

Связи каталога

Вы не можете создавать жесткие связи для каталога. Однако каждый каталог имеет две жестко определенные связи:

- . ("точка")
- .. ("точка точка")

Имя файла "точка" соответствует текущему каталогу; "точка точка" соответствует каталогу, *предшествующему* текущему каталогу.

Заметьте, что "точка точка" для каталога "/" - это просто "/", - вы не можете подняться выше.

Символические связи

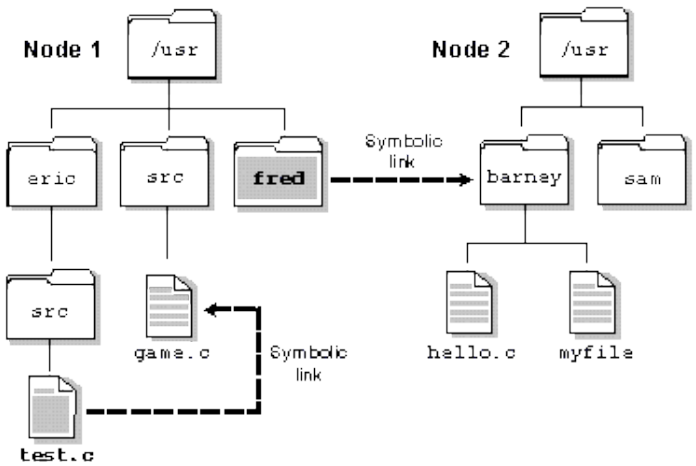
Символическая связь - это особый файл, который содержит в качестве данных имя пути. Когда символическая связь используется в запросе ввода/вывода - например, *open()*, - обозначение связи в имени пути заменяется ее "данными". Символическая связь является гибким средством для перенаправления пути и часто используется для создания множества путей к одному и тому же файлу. В отличие от жестких связей, символические связи могут выходить за пределы файловой системы и также являться связями для каталогов.

В следующем примере каталоги //1/usr/fred и //2/usr/barney являются связями на один и тот же каталог, хотя они находятся в различных файловых системах, и даже на различных узлах (смотри следующую диаграмму). Это не может быть сделано с использованием жестких связей:

```
//1/usr/fred --> //2/usr/barney
```

Заметьте, что символическая связь и адресуемый каталог не обязаны иметь одно и то же имя. В большинстве случаев символические связи используются для привязки одного каталога к другому. Однако они также могут быть использованы для файлов, как в этом примере:

```
//1/usr/eric/src/test.c --> //1/usr/src/game.c
```



Если вы хотите:	**Используйте утилиту:**
Создать символическую связь	ln (с опцией **-s**)
Удалить символическую связь*	rm
Узнать, является ли файл символической связью	ls

* Помните, что удаление символической связи действует только на связь, а *не* на адресуемый объект

Некоторые функции оперируют непосредственно с символическими связями. Для этих функций замена обозначения связи в пути на ее содержимое не производится. К этим функциям относятся *unlink()* (которая удаляет символическую связь), *lstat()* и *readlink()*.

Так как символические связи могут указывать на каталоги, то неверная конфигурация может привести к проблемам, таким, как циклические связи. Чтобы защититься от циклических связей, система накладывает ограничения на количество переходов; этот предел определен как SYMLOOP_MAX во включаемом файлelimits.h.

Программные каналы (pipes) и FIFO

Программные каналы (pipes)

Программный канал - это неименованный файл, который служит как канал ввода/вывода между двумя или более взаимодействующими процессами - один процесс пишет в программный канал, другой читает из программного канала. Менеджер файловой системы обеспечивает буферизацию данных. Размер буфера определен как PIPE_BUF в файле <tt SH="SH"><limits.h>@@. Программный канал удаляется после того как закрыты оба его конца.

Программные каналы обычно используются, когда два процесса хотят выполняться параллельно, с однонаправленной передачей данных от одного процесса к другому. Если требуется двунаправленная передача данных, то должны использоваться сообщения.

Типичное применение программного канала состоит в соединении вывода одной программы с вводом другой программы. Это соединение часто производится командным интерпретатором (Shell). Например:

```
ls | more
```

направляет стандартный вывод от утилиты ls через программный канал в стандартный ввод утилиты `more`.

Если вы хотите:	**Используйте:**
Создать программный канал из командного интерпретатора	Символ программного канала ("|")
Создать программный канал из программы	Функции *pipe()* или *popen()*

Примечание. На бездисковых рабочих станциях вы можете запустить Менеджер программных каналов (Pipe) вместо Менеджера файловой системы, когда требуются только программные каналы. Менеджер программных каналов оптимизирован для канального (конвейерного) ввода/вывода и может обеспечить большую пропускную способность, чем Менеджер файловой системы.

FIFO

FIFO - это по существу то же самое, что и программные каналы, за исключением того, что FIFO являются именованными постоянными файлами, которые хранятся в каталогах файловой системы.

Если вы хотите:	Используйте:
Создать FIFO из командного интерпретатора	Утилиту mkfifo
Создать FIFO из программы	Функцию *mkfifo()*
Удалить FIFO из командного интерпретатора	Утилиту rm
Удалить FIFO из программы	Функции *remove()* или *unlink()*

Производительность Менеджера файловой системы

Свойства Менеджера файловой системы, обеспечивающие высокопроизводительный доступ к диску:

- лифтовый поиск;
- кэш-буфер;
- многопотоковая обработка;
- клиент-управляемый приоритет;
- временные файлы;
- псевдодиски.

Лифтовый поиск

Лифтовый поиск минимизирует общие затраты времени на позиционирование магнитной головки при чтении или записи на диск. Запросы ввода/вывода упорядочиваются таким образом, чтобы все они могли быть выполнены за один проход магнитной головки, от самого младшего к самому старшему адресу на диске.

Лифтовый поиск также имеет усовершенствование, обеспечивающее выполнение мультисекторного ввода/вывода там, где возможно.

Кэш-буфер

Кэш-буфер - это интеллектуальный буфер между Менеджером файловой системы и драйвером диска. Кэш-буфер хранит блоки файлов с целью минимизировать количество обращений Менеджера файловой системы к диску. По умолчанию размер кэша определяется общим объемом оперативной памяти, но вы можете задать другое значение через опцию при запуске Fsys.

Операции чтения являются синхронными. Операции записи, напротив, обычно являются асинхронными. Когда данные попадают в кэш, Менеджер файловой системы отвечает клиенту (функцией *Reply()*), извещая его, что данные записаны. Затем выполняется запись данных на диск с максимально-возможной скоростью, обычно не позднее, чем через пять

секунд.

Приложения могут изменять механизм записи применительно к конкретным файлам. Например, приложение, работающее с базой данных, может потребовать, чтобы запись в определенный файл производилась синхронно. Это обеспечит высокий уровень целостности файла в случае аппаратного или программного сбоя.

Многопотоковая обработка

Менеджер файловой системы является многопотоковым процессом, и, таким образом, он может обрабатывать несколько запросов ввода/вывода одновременно. Это позволяет Менеджеру файловой системы в полной мере реализовать возможности параллельной обработки, так как он в состоянии:

- получить одновременный доступ к нескольким устройствам;
- удовлетворить запросы ввода/вывода из кэш-буфера, в то время как выполняются другие запросы ввода/вывода, осуществляющие доступ к диску.

Клиент-управляемый приоритет

Приоритет Менеджера файловой системы может определяться приоритетом процесса, пославшего ему запрос. В этом случае, когда Менеджер файловой системы получает сообщение, его приоритет устанавливается равным приоритету процесса, пославшего сообщение. Более подробно об этом говорится в разделе "**Диспетчеризация процессов**" главы "Микроядро".

Временные файлы

QNX предусматривает возможность открытия временных файлов, т.е. файлов, которые записываются и затем читаются в течение короткого промежутка времени. Для таких файлов Менеджер файловой системы старается хранить блоки данных в кэш-буфере и производит запись блоков на диск только в случае крайней необходимости.

Псевдодиски

Менеджер файловой системы содержит встроенную возможность поддержки электронного диска, позволяющую использовать до 8 Мегабайт оперативной памяти в качестве псевдодиска. Так как Менеджер файловой системы использует высокоэффективный механизм составных сообщений, то данные из псевдодиска копируются непосредственно в приложение.

Менеджер файловой системы в состоянии обойти кэширование, так как псевдодиск является встроенным, а не реализован как отдельный драйвер. Для информации об обмене составными сообщениями, смотри раздел "**Дополнительные возможности**" в главе "Микроядро".

Так как псевдодиски исключают аппаратные задержки и не используют кэширование данных, поэтому они обеспечивают больший детерминизм при выполнении операций ввода/вывода

по сравнению с жесткими дисками.

Надежность файловой системы

В QNX высокая производительность файловой системы достигается не за счет снижения надежности. Это обеспечивается несколькими способами.

В то время как большая часть данных помещается в кэш-буфер и записывается с небольшой задержкой, критические для файловой системы данные записываются немедленно. Обновления каталогов, индексных дескрипторов (inodes), блоков экстентов и битовой карты производятся без задержки, чтобы гарантировать целостность структуры файловой системы на диске (т.е. что не будет внутреннего несоответствия данных на диске).

Иногда все перечисленные выше структуры данных должны быть обновлены. Например, если вы перемещаете файл в каталоге, последний экстент которого заполнен, то каталог должен вырасти. В таких случаях порядок операций тщательно подобран таким образом, что даже если произойдет катастрофический сбой в момент, когда операция еще не полностью завершена (например, отключение питания), то файловая система после перезапуска все же сохранит работоспособность. В худшем случае, некоторые блоки будут выделены, но не использованы. Исправить подобную ситуацию можно, запустив утилиту chkfsys.

Восстановление файловой системы

Даже самые надежные системы не застрахованы от аварийных ситуаций, таких как:

- появление плохих блоков на диске в результате бросков или провалов напряжения;
- неумелый или злонамеренный пользователь, имеющий доступ к привилегиям администратора системы, выполнил инициализацию файловой системы (утилитой dinit);
- некорректная программа (в особенности, выполняющаяся не в среде QNX) игнорирует информацию о разделах диска и перезаписывает часть раздела QNX.

Чтобы после таких ситуаций можно было восстановить как можно больше файлов, на диск записываются уникальные "сигнатуры" для автоматической идентификации и восстановления критических частей файловой системы. Файл с индексными дескрипторами (/.inodes), так же как и каждый каталог, и блок экстентов, все содержат уникальные образцы данных, которые могут быть использованы утилитой chkfsys для восстановления серьезно поврежденной файловой системы.

Более подробная информация о восстановлении файловой системы содержится в документации к утилите chkfsys.

Работа с дисками

Менеджер файловой системы управляет *блок-ориентированными файлами*. Эти файлы определяют *диски* и *разделы дисков*.

Диски и дисковые подсистемы

QNX считает каждый физический диск на компьютере *блок-ориентированным* файлом. Как блок-ориентированный файл, диск рассматривается файловой системой QNX как последовательность блоков размером по 512 байт, независимо от размера диска. Блоки нумеруются, начиная с первого блока на диске (блок 1).

Так как каждый диск является блок-ориентированным файлом, он может быть открыт как одно целое для низкоуровневого доступа с использованием стандартных POSIX Си функций, таких как *open()*, *read()*, *write()* и *close()*. На уровне блок-ориентированного файла, который определяет целый диск, QNX не делает абсолютно никаких предположений о каких-либо структурах данных, которые могут существовать на диске.

Компьютер под управлением QNX может иметь одну или несколько *дисковых подсистем*. Каждая дисковая подсистема состоит из контроллера и одного или более дисков. Вы запускаете драйвер устройства для каждой дисковой подсистемы, которая должна управляться Менеджером файловой системы.

Разделы ОС

QNX соответствует промышленному стандарту де-факто, который позволяет различным операционным системам разделять один и тот же физический диск. В соответствии с этим стандартом, таблица разделов может определять до четырех первичных разделов на диске. Таблица хранится в первом блоке диска.

Каждый раздел должен иметь "тип", узнаваемый операционной системой, которая должна использовать этот раздел. Следующий список содержит используемые на данный момент типы разделов:

Тип:	**Операционная система**
1	DOS (12-битная FAT)
4	DOS (16-битная FAT; разделы <32Мбайт)
5	Расширенный раздел DOS
6	DOS 4.0 (16-битная FAT; раздел >=32Мбайт)
7	OS/2 HPFS
7	Предыдущая версия QNX 2 (до 1988)
8	QNX 1.x и 2.x ("qny")
9	QNX 1.x и 2.x ("qnz")
11	DOS 32-битная FAT; разделы до 2047Gбайт
12	То же, что тип 11, но использует LBA расширения прерывания Int 13h
14	То же, что тип 6, но использует LBA расширения прерывания Int 13h
15	То же, что тип 5, но использует LBA расширения прерывания Int 13h
77	QNX POSIX раздел
78	QNX POSIX раздел (вторичный)
79	QNX POSIX раздел (вторичный)
99	UNIX

Если вы хотите иметь несколько разделов QNX 4.x на одном физическом диске, вам следует

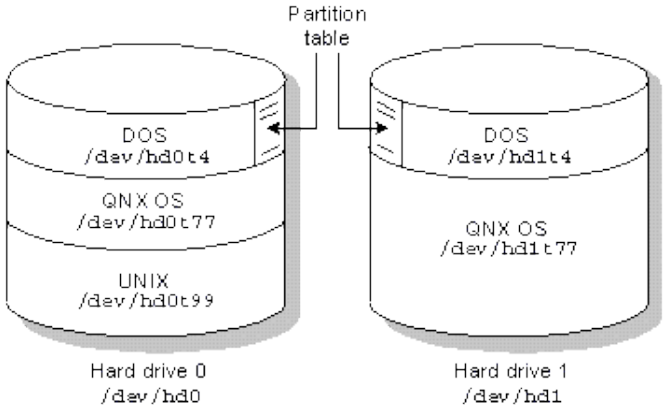
использовать тип 77 для первого QNX раздела, тип 78 для второго QNX раздела, и тип 79 для третьего. Вы можете использовать другие типы для второго и третьего QNX разделов, но 78 и 79 предпочтительнее. Чтобы пометить любой из этих разделов как загрузочный, используйте утилиту fdisk.

Во время загрузки, загрузчик QNX (опционально устанавливаемый утилитой fdisk) позволяет выбирать в таблице разделов в качестве загрузочного другой раздел, не являющийся загрузочным по умолчанию.

Вы можете использовать утилиту fdisk для создания, модификации или удаления разделов.

Так как QNX рассматривает каждый раздел на диске как блок-ориентированный файл, то это дает возможность доступа к следующему:

- диску целиком, игнорируя разделы, как к блок-ориентированному файлу;
- отдельному разделу как к блок-ориентированному файлу; этот блок-ориентированный файл будет подмножеством блок-ориентированного файла, который определяет весь диск.



Два физических диска. Первый содержит DOS, QNX и UNIX разделы. Второй диск имеет DOS и QNX разделы

Определение блок-ориентированный файлов

Имена всех блок-ориентированных файлов помещаются в дерево префиксов того компьютера, на котором расположены блок-ориентированные файлы (дерево префиксов рассматривается в главе "Пространство имен ввода/вывода"). Когда запускается драйвер контроллера дисков, Менеджер файловой системы автоматически регистрирует префиксы, которые определяют блок-ориентированный файл для каждого физического диска, подключенного к контроллеру. Утилита mount используется для того, чтобы зарегистрировать префикс для блок-ориентированного файла для каждого раздела на диске.

Пусть, например, у вас имеется стандартный контроллер Western Digital, к которому подключены два диска. На одном диске вы хотите смонтировать раздел DOS, раздел QNX и раздел UNIX. На другом диске вы хотите смонтировать раздел DOS и раздел QNX.

Менеджер файловой системы определит блок-ориентированные файлы /dev/hd0 и /dev/hd1 для двух дисков, подключенных к контроллеру.

Затем вы используете утилиту mount, чтобы определить блок-ориентированный файл для каждого раздела. Например:

```
mount -p /dev/hd0  -p /dev/hd1
```

определит следующие блок-ориентированные файлы:

Раздел ОС:	**Блок-ориентированный файл**
Раздел DOS на диске hd0	/dev/hd0t4
Раздел QNX на диске hd0	/dev/hd0t77
Раздел UNIX на диске hd0	/dev/hd0t99
Раздел DOS на диске hd1	/dev/hd1t4
Раздел QNX на диске hd1	/dev/hd1t77

Заметьте, что обозначение t*n* используется для обозначения разделов на диске, используемых определенными операционными системами. Например, раздел DOS обозначается t4, раздел UNIX - это t99 и т.д.

Монтирование файловой системы

Обычно файловая система QNX монтируется на блок-ориентированном файле. Для этого вы снова используете утилиту mount - она позволяет задать префикс, который идентифицирует файловую систему. Например:

```
mount /dev/hd0t77 /
```

монтирует файловую систему с префиксом / на разделе, который определен блок-ориентированным файлом с именем hd0t77.

Примечание. Если диск разбит на разделы, то вы должны смонтировать блок-ориентированный файл для раздела QNX 4.x (например /dev/hd0t77), а не основной блок-ориентированный файл, который определяет весь диск (например, /dev/hd0). Если вы попытаетесь смонтировать основной блок-ориентированный файл для всего диска, то при попытке доступа к файловой системе получите сообщение "corrupt filesystem" (поврежденная файловая система).

Демонтирование файловой системы

Чтобы демонтировать файловую систему, используйте утилиту umount. Так, например, следующая команда демонтирует файловую систему на первичном разделе QNX:

```
umount /dev/hd0t77
```

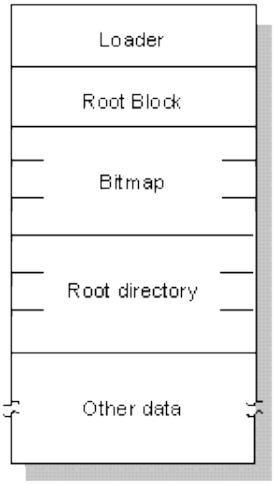
После того, как файловая система демонтирована, файлы в этом разделе уже не доступны.

Ключевые компоненты раздела QNX

В начале каждого раздела QNX располагаются следующие ключевые компоненты:

- блок загрузчика;
- корневой блок;
- битовая карта;
- корневой каталог.

Эти структуры создаются при инициализации файловой системы утилитой dinit.



Структура файловой системы QNX внутри раздела диска

Блок загрузчика

Это первый физический блок раздела диска. Этот блок содержит код, который загружается и затем исполняется BIOS компьютера для загрузки операционной системы из раздела. Если диск не разбит на разделы (например, гибкий диск), этот блок является первым физическим блоком на диске.

Корневой блок

Корневой блок имеет ту же структуру, что и обычный каталог. Он содержит информацию о четырех особых файлах:

- *корневой каталог* файловой системы (/)
- файл /.inodes
- файл /.boot
- файл /.altboot

Файлы /.boot и /.altboot содержат образы операционной системы, которые загружаются программой начальной загрузки QNX.

Обычно загрузчик QNX загружает образ ОС, хранящийся в файле /.boot. Но если файл /.altboot не пуст, то вам будет предложена опция загрузки образа, хранящегося в файле /.altboot.

Битовая карта

Чтобы распределять пространство на диске, QNX использует *битовую карту*, хранящуюся в файле /.bitmap. Этот файл содержит битовую маску для всех блоков на диске, показывающую, какие блоки заняты. Каждому блоку соответствует один бит. Если значение бита 1, то соответствующий блок на диске занят.

Корневой каталог

Корневой каталог раздела ведет себя как обычный каталог, за двумя исключениями:

- как ".", так и ".." являются связями к одной и той же inode информации, а именно, корневым каталогом inode в корневом блоке;
- корневой каталог всегда содержит элементы для файлов /.bitmap, /.inodes, /.boot и /.altboot. Эти элементы предусмотрены для того, чтобы программы, которые выдают информацию о файловой системе, видели эти элементы как обычные файлы.

Менеджер файловой системы DOS

В QNX пространство имен ввода/вывода управляется через префиксы, которые направляют запросы соответствующим процессам-менеджерам. Одним из таких процессов является Менеджер файловой системы DOS (Dosfsys). Dosfsys регистрирует префикс /dos и представляет диски DOS внутри пространства имен QNX как "гостевые" файловые системы.

Dosfsys обеспечивает прозрачный доступ к дискам DOS, так что вы можете рассматривать файловые системы DOS как файловые системы QNX. Такая прозрачность позволяет процессам работать с файлами DOS без каких-либо специальных знаний или действий. Стандартные библиотечные функции ввода/вывода, такие как *open()*, *close()*, *read()* и *write()* работают с файлом в разделе DOS точно так же, как и с файлом в разделе QNX. Например, чтобы копировать файл из раздела QNX в раздел DOS, достаточно выполнить команду:

```
cp /usr/luc/file.dat /dos/c/file.dat
```

Заметьте, что /dos/c - это путь к DOS диску C. Команда cp не содержит какого-либо специального кода для определения, находится ли копируемый файл на диске DOS. Другие команды работают с такой же прозрачностью (например, утилиты cd, ls и mkdir).

Если не существует эквивалента DOS для функции QNX, такой как mkfifo() или link(), то Dosfsys возвращает соответствующий код ошибки (т.е. errno).

Dosfsys работает как с гибкими дисками, так и с разделами жестких дисков. Весь требуемый доступ к диску на низком уровне производится через стандартные функции Менеджера файловой системы. Поэтому, без доступа к коду низкого уровня, Dosfsys способен обеспечить прозрачный интерфейс между приложениями QNX и файловой системой DOS.

Файловая система CD-ROM

Файловая система CD-ROM, Iso9660fsys, обеспечивает прозрачный доступ к носителям CD-

ROM, таким образом можно работать с файловыми системами CD-ROM, как будто это файловые системы POSIX. Такая прозрачность обеспечивает процессам доступ к файлам на CD-ROM стандартными средствами.

Менеджер Iso9660fsys реализует стандарт ISO 9660, включая расширения Rock Ridge. Этому стандарту соответствуют компакт-диски DOS и Windows. В дополнение к обычным файлам, Iso9660fsys также поддерживает аудио.

Файловая система флэш

Менеджер файловой системы флэш Efsys.* был специально разработан для работы, как со встроенной, так и со сменной флэш-памятью. Файлы, записанные на сменные носители флэш (карты PC-Card), переносимы в другие системы, которые также поддерживают этот стандарт.

Менеджер Efsys.* сочетает функции менеджера файловой системы и драйвера устройства. Так как Efsys.* содержит драйвер устройства, то существуют отдельные версии Efsys.* для различных видов оборудования встраиваемых систем. Например:

- Efsys.explr2 для платы Intel EXPLR2 (с RFA);
- Efsys.elansc400 для платы AMD Elan SC400;
- Efsys.pcmcia для смешанного использования файловых систем флэш и устройств PCMCIA.

Ограничения

Функциональность файловой системы ограничена используемыми устройствами памяти. Например, для устройств ROM файловая система доступна только для чтения.

Для устройств флэш-памяти существуют ограничения на запись файлов. Вы можете только дозаписывать файл. Кроме того, не обновляется время последнего доступа к файлу. В настоящий момент эти ограничения распространяются даже на SRAM устройства.

Восстановление свободного пространства

Менеджер Efsys.* хранит каталоги и файлы, используя связные (связные - имеющие прямые/обратные указатели) списки структур данных, а не блоки фиксированного размера, как на диске, используемые в традиционных файловых системах с вращающимся носителем. При удалении каталога или файла, принадлежащие ему структуры данных помечаются как удаленные, но не стираются, чтобы избежать непрерывного стирания и перезаписи (и тем самым потерь времени на эти операции).

Со временем свободное место закончится, и менеджеру файловой системы придется выполнить восстановление свободного пространства (иногда эту операцию называют еще "уборка мусора"). Во время этой процедуры Efsys.* восстанавливает свободное место, занимаемое удаленными файлами и каталогами. Для проведения этой операции менеджеру файловой системы требуется хотя бы один свободный блок. Утилита mkffs автоматически резервирует для этой цели один блок при создании файловой системы.

Сжатие и распаковка

Менеджер Efsys.* поддерживает распаковку, что увеличивает объем данных, который может храниться на носителе. Распаковка выполняется менеджером файловой системы для приложений прозрачно.

Для этого файлы должны быть, перед запуском утилиты mkffs, предварительно сжаты с помощью утилиты bpe. Если же копировать сжатый файл в уже созданную файловую систему флэш, то он останется сжатым и при чтении.

Доступ к файлам

Если запретить расширения POSIX, то владельцем файлов всегда будет считаться root, а биты режима всегда будут установлены в rwx. Команды chgrp, chmod и chown в этом случае не будут работать.

Монтирование

Может производиться только при инициализации разделов или при запуске менеджера файловой системы.

Доступ на низком уровне

При запуске Efsys.*, он создает для каждого устройства памяти специальный файл в каталоге /dev. В системе с двумя устройствами памяти Efsys.* создаст файлы /dev/skt1 и /dev/skt2. Специальные устройства игнорируют разбиение на разделы, позволяя доступ к носителям на низком уровне.

Доступ к разделу, содержащему образ файловой системы, возможен только на низком уровне (как к "сырому" устройству). Для каждого такого раздела Efsys.* создает специальный файл вида /dev/skt*X*img*Y*, где *X* - это номер гнезда (socket), а *Y* - номер раздела на этом носителе.

Файловая система NFS

Первоначально разработанная компанией Sun Microsystems, NFS (Network File System - Сетевая Файловая Система) является TCP/IP приложением, которое с тех пор было реализовано на большинстве DOS и UNIX систем. Его реализация в QNX не заменима для прозрачного доступа к файловым системам других ОС, поддерживающих NFS.

Примечание. QNX изначально поддерживает сетевые файловые системы. Использовать NFS необходимо *только* для доступа к не-QNX NFS файловым системам, либо если вы хотите открыть NFS-клиентам доступ к файлам QNX.

NFS позволяет отображать удаленные файловые системы - полностью или частично - в локальное пространство имен. Каталоги на удаленной системе выглядят как часть локальной файловой системы, и все утилиты работы с файлами (ls, cp и mv) работают с удаленными файлами так же, как и с локальными.

Примечание. В QNX 4 для NFS требуется менеджер Socket, который поддерживает сетевые протоколы TCP/IP. Заметьте, что его "облегченный" вариант, Socklet, может использоваться,

если не нужна NFS.

Файловая система SMB

SMBfsys реализует протокол SMB (Server Message Block) совместного использования файлов, который используется различными серверами, такими как Windows NT, Windows 95, Windows for Workgroups, LAN Manager, Samba. SMBfsys обеспечивает QNX-клиенту прозрачный доступ к удаленным дискам таких серверов.

SMBfsys реализует этот протокол, используя только NetBIOS поверх TCP/IP, *не* NetBEUI. Соответственно, необходимо, чтобы TCP/IP был установлен, как на QNX-машине, так и на удаленном сервере. После того, как запущен SMBfsys и смонтирован удаленный сервер, файловая система сервера появляется в локальном дереве каталог.

Глава 6. Менеджер устройств

Эта глава охватывает следующие темы:

- Введение
- Обслуживание устройств
- Режим редактируемого ввода
- Режим необрабатываемого ввода
- Драйверы устройств
- Консоль QNX
- Последовательные устройства
- Параллельные устройства
- Производительность подсистемы устройств

Введение

Менеджер устройств QNX (Dev) является интерфейсом между процессами и *терминальными устройствами*. Эти ерминальные устройства располагаются в пространстве имен ввода/вывода с именами, начинающимися с /dev. Например, консольное устройство в QNX будет иметь имя:

```
/dev/con1
```

Обслуживание устройств

Программы в QNX получают доступ к терминальным устройствам, используя стандартные функции *open()*, *close()*, *read()* и *write()*. Для процесса QNX терминальное (оконечное) устройство представляется двунаправленным потоком байт, который может считываться и записываться процессом.

Менеджер устройств регулирует поток данных между приложением и устройством. Dev выполняет некоторую обработку этих данных в соответствии с параметрами *управляющей структуры терминала* (называемой *termios*), которая существует для каждого устройства. Пользователи могут запрашивать и/или изменять эти параметры с помощью утилиты stty; программы могут использовать функции *tcgetattr()* и *tcsetattr()*.

Параметры структуры *termios* управляют функциональностью низкого уровня, такой как:

- параметры линии (включая скорость передачи, контроль четности, стоп-биты, биты данных);
- эхо-вывод символов;
- редактирование строки ввода;
- распознавание и реакция на команду "Break" и зависания;
- программное и аппаратное управление потоком данных;
- преобразование выводимых символов.

Менеджер устройств также предоставляет набор дополнительных услуг, доступных

процессам для работы с терминальным устройством. В следующей таблице приведены некоторые из этих услуг.

Процесс может:	**Функция Си:**
Выполнять операции чтения с контролем времени	*dev_read()* или *read()* и *tcsetattr()*
Получать асинхронное извещение о доступных данных	*dev_arm()*
На одном или более устройствах ввода ждать полного завершения операции вывода	*tcdrain()*
Посылать команду Break по каналу связи	*tcsendbreak()*
Разорвать соединение	*tcdropline()*
Вставить входные данные	*dev_insert_chars()*
Выполнять неблокирующиеся чтение и запись	*open()* и *fcntl()* (O_NONBLOCK mode)

Режим редактируемого ввода

Наиболее важный режим работы устройства управляется битом ICANON в управляющей структуре *termios*. Если этот управляющий бит установлен, то Менеджер устройств выполняет функции редактирования строки для принимаемых символов. Таким образом, только когда строка "введена" - обычно, когда получен символ перевода каретки (CR), - данные станут доступны для прикладных процессов. Такой режим работы называется *редактируемым* - от английского *edited*. Этот режим еще называют *canonical* (каноническим) и иногда *cooked* (приготовительным).

Большинство неполноэкранных приложений выполняются в редактируемом режиме. Типичным примером является командный интерпретатор (Shell).

Следующая таблица показывает, как Dev обрабатывает некоторые специальные управляющие символы, если соответствующие параметры установлены в структуре *termios*.

Dev выполнит:	**Когда получит символ:**
Сдвиг курсора на один символ влево	LEFT
Сдвиг курсора на один символ вправо	RIGHT
Сдвиг курсора в начало строки	HOME
Сдвиг курсора в конец строки	END
Удаление символа слева от курсора	ERASE
Удаление символа в текущей позиции курсора	DEL
Удаление всей строки ввода	KILL
Стирание текущей строки и восстановление предыдущей	UP
Стирание текущей строки и восстановление следующей	DOWN
Переключение между режимами вставки и замены	INS

Символы редактирования строки отличаются для различных терминалов. При запуске консоли QNX всегда определен полный набор редактирующих символов.

Если терминал подключен к QNX через последовательный канал, необходимо установить редактирующие символы, которые будут использоваться для данного конкретного терминала. Для этого вы можете использовать утилиту stty. Например, если терминал VT100 подключен к последовательному порту (/dev/ser1), то следующая команда извлечет соответствующие редактирующие символы из базы данных *terminfo* и применит их к /dev/ser1:

```
stty term=vt100 </dev/ser1
```

Если же к этому последовательному порту подключен модем, для связи с другой QNX системой с помощью утилиты qtalk, редактирующие символы следует установить так:

```
stty term=qnx </dev/ser1
```

Режим необрабатываемого ввода

Когда бит ICANON не установлен, то говорят, что устройство находится в *необрабатываемом* ("*сыром*", английский термин *raw*) режиме. В этом режиме не производится никакое редактирование ввода, а все получаемые данные немедленно становятся доступными для QNX-процессов.

Полноэкранные программы и коммуникационные программы являются примерами приложений, которые переводят устройство в *сырой* режим.

При чтении из *сырого* устройства приложение может задать условия, при которых будет удовлетворен запрос на ввод данных. Критерии, используемые при приеме *сырых* данных, базируются на двух параметрах структуры *termios*: MIN и TIME. Приложение может определить еще один дополнительный параметр при вызове функции *dev_read()*. Этот параметр, TIMEOUT, используется при написании протоколов или приложений реального времени. Учтите, что для функции *read()* TIMEOUT всегда 0.

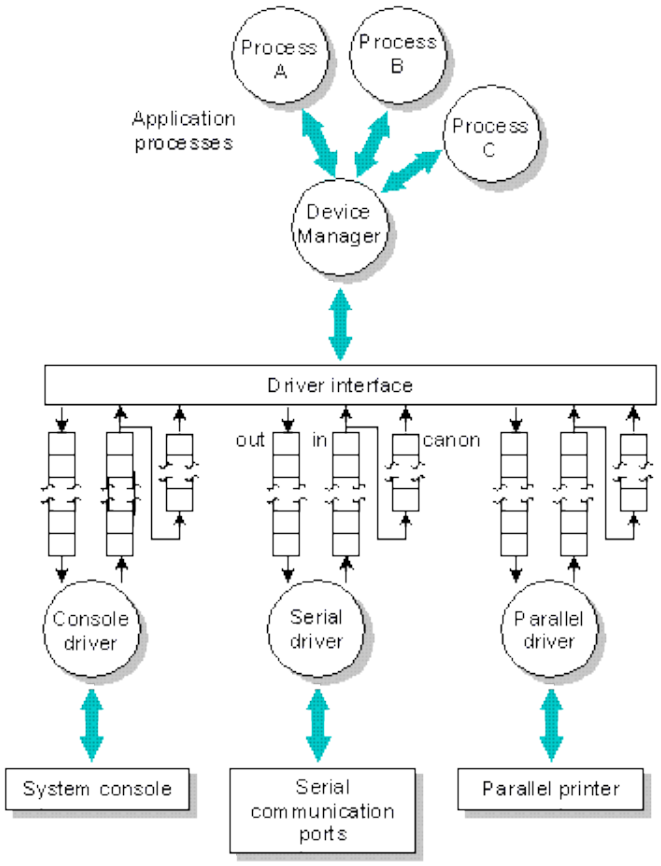
Когда процесс запрашивает ввод *n* байт данных, эти три параметра определяют, когда должен быть удовлетворен запрос:

MIN	**TIME**	**TIMEOUT**	**Описание:**
0	0	0	Вернуть немедленно столько байт, сколько доступно в данный момент (но не более *n* байт)
M	0	0	Вернуть не более *n* байт только тогда, когда доступны, по меньшей мере, M байт
0	T	0	Вернуть не более *n* байт только тогда, когда доступен хотя бы один байт либо истекло T x .1 секунд
M	T	0	Вернуть не более *n* байт только тогда, либо когда будут доступны M байт, либо будет принят хотя бы один байт и интервал между любыми последовательно принятыми символами превысил T x .1 секунд
0	0	t	Зарезервировано
M	0	t	Вернуть не более *n* байт только тогда, когда доступны M байт либо истекло t x .1 секунд
0	T	t	Зарезервировано

MIN	TIME	TIMEOUT	Описание:
M	T	t	Вернуть не более *n* байт только тогда, когда доступны M байт, либо истекло t x .1 секунд и не был получен ни один символ, либо был принят хотя бы один байт и интервал между любыми последовательно принятыми символами превысил T x .1 секунд

Драйверы устройств

На рисунке показана типичная подсистема устройств в QNX.



Менеджер устройств (Dev) организует обмен данными между устройствами и прикладными программами. Аппаратный интерфейс управляется индивидуальными процессами *драйверами*. Dev, для каждого из *устройств*, обменивается данными с *драйверами* через очереди в разделяемой памяти.

Примечание. Использование разделяемой памяти требует, чтобы Dev и драйверы находились на одном и том же физическом компьютере, преимуществом же является повышенная производительность.

Для каждого *терминального устройства* используются три очереди. Каждая очередь реализована на основе механизма "первый вошел - первый вышел". Каждой очереди также соответствует своя управляющая структура.

Принимаемые данные помещаются драйвером в очередь сырого ввода, и Dev извлекает их, только когда прикладные программы запрашивают данные. Обработчики прерываний внутри драйверов обычно вызывают проверенную библиотечную процедуру внутри Dev для добавления данных в эту очередь - это обеспечивает единообразный порядок ввода и существенно минимизирует ответственность драйвера.

Dev помещает выводимые данные в очередь вывода; данные извлекаются драйвером по мере того, как символы физически передаются устройству. Dev вызывает проверенную процедуру внутри драйвера каждый раз, когда добавляются новые данные, таким образом он "подталкивает" драйвер к работе (в случае, если он не был занят). Благодаря использованию очередей вывода Dev реализует *буферизованную запись* (write-behind) для всех *терминальных устройств*. Только когда буферы вывода заполнены, Dev вызывает блокирование процесса при записи.

Редактируемая очередь полностью управляется Dev и используется при вводе данных в *редактируемом* режиме. Размер этой очереди определяет максимальный размер строки редактируемого ввода для конкретного устройства.

Размеры всех этих очередей могут конфигурироваться системным администратором; единственное ограничение состоит в том, что общий суммарный размер всех трех очередей не может превышать 64K. Значений по умолчанию обычно более чем достаточно для большинства аппаратных конфигураций, но вы можете "настраивать" их, либо для уменьшения общей потребности системы в памяти, либо в случае нестандартных аппаратных конфигураций.

Управление устройствами

Драйверы устройств просто добавляют получаемые данные в очередь сырого ввода или извлекают и передают данные из очереди вывода. Dev решает, когда вывод должен быть приостановлен, должно ли быть "эхо" принимаемых данных и т.д.

Чтобы обеспечить хорошую реакцию на входные события, Dev должен выполняться с достаточно высоким приоритетом. Dev обычно мало загружен работой, поэтому он редко снижает общую производительность системы. Сами драйверы, как и любые другие процессы в QNX, могут выполняться с различными приоритетами в зависимости от характера обслуживаемых устройств.

Сами драйверы, как и любые другие процессы в QNX, могут выполняться с различными приоритетами в зависимости от характера обслуживаемых устройств.

Управление устройствами на низком уровне выполняется через дальний вызов входной точки *ioctl* внутри каждого драйвера. Общий набор *ioctl* команд поддерживается непосредственно Dev. Специфические для устройства *ioctl* команды могут быть посланы QNX процессами драйверу через функцию Си *qnx_ioctl()*.

Консоль QNX

Системные консоли поддерживаются драйвером Dev.con. Экран, адаптер дисплея и системная клавиатура - вместе называются *консолью*.

QNX позволяет параллельное выполнение множества сессий на консолях посредством *виртуальных консолей*. Драйвер Dev.con обычно поддерживает более одного набора очередей ввода/вывода к Dev, которые доступны пользовательским процессам как множество *терминальных устройств* с именами вида /dev/con1, /dev/con2 и т.д. С точки зрения приложения, это "настоящие" доступные консоли.

Конечно, существует только один *физический* экран и клавиатура, и поэтому только *одна* из этих виртуальных консолей действительно показывается в каждый момент времени. Клавиатура "подключена" к той виртуальной консоли, которая видима в текущий момент.

Функции, специфичные для консоли

В дополнение к реализации стандартного терминала QNX (описан в "Руководстве пользователя"), драйвер консоли также обеспечивает набор специфических для консоли функций, которые позволяют приложениям непосредственно взаимодействовать с драйвером консоли путем обмена сообщениями. Связь устанавливается вызовом функции Си *console_open()*. После того как связь установлена, процесс имеет доступ к следующим возможностям:

Процесс может:	**Через функцию Си:**
Читать непосредственно с экрана консоли	*console_read()*
Писать непосредственно на экран консоли	*console_write()*
Получать асинхронное извещение о важных событиях (например, изменение положения курсора, размера дисплея, переключение консоли и т.д.)	*console_arm()*
Управлять размерами консоли	*console_size()*
Переключать видимую консоль	*console_active()*

Драйвер консоли QNX выполняется как нормальный процесс QNX. Вводимые с клавиатуры символы помещаются обработчиком прерывания клавиатуры непосредственно в очередь ввода. Выводимые данные отображаются Dev.con в то время, когда он выполняется как процесс.

Последовательные устройства

Последовательные каналы связи обслуживаются драйвером Dev.ser. Этот драйвер может обслуживать более одного физического канала; он обеспечивает терминальные устройства с именами вида /dev/ser1, /dev/ser2 и т.д.

При запуске Dev.ser вы можете задать аргументы командной строки, которые определяют, какие - и сколько - последовательные порты установлены. Чтобы увидеть доступные последовательные порты, используйте утилиту ls:

```
ls /dev/ser*
```

Dev.ser является примером управляемого прерываниями сервера ввода/вывода. После инициализации оборудования сам процесс переходит в состояние ожидания ("засыпает"). Прерывания помещают входные данные непосредственно в очередь ввода. Первый выводимый символ передается устройству, когда Dev "подталкивает" драйвер. Последующие символы передаются при обработке соответствующего прерывания.

Параллельные устройства

Параллельные порты принтера обслуживаются драйвером Dev.par. При запуске Dev.par вы можете задать аргумент командной строки, который определяет, какой последовательный порт установлен. Чтобы увидеть, доступен ли последовательный порт, используйте утилиту ls:

```
ls /dev/par*
```

Dev.par является только выводящим драйвером, поэтому у него нет очередей ввода. Вы можете конфигурировать размер буфера вывода с помощью аргументов командной строки при запуске Dev.par. Буфер вывода может быть сделан очень большим, если вы хотите обеспечить программный буфер печати.

Dev.par является примером сервера ввода/вывода, не использующим прерывания. Этот процесс нормально находится в состоянии RECEIVE-блокирован, ожидая появления данных в очереди вывода и "толчка" от Dev. Когда доступны данные для вывода на печать, Dev.par выполняется в цикле работа-ожидание (с относительно низким адаптивным приоритетом), ожидая, когда символы будут приняты принтером. Такой низкоприоритетный цикл работа-ожидание не влияет на общую производительность системы и, в то же время, достигает максимально возможную пропускную способность к параллельному устройству.

Производительность подсистемы устройств

Поток событий внутри подсистемы устройств сконструирован так, чтобы минимизировать избыточность и достичь максимальной пропускной способности, когда устройство работает в *сыром* режиме. С этой целью используются следующие правила:

- Обработчики прерываний помещают принятые данные непосредственно в очереди в памяти. Только при наличии ждущего запроса на чтение *и* при условии, что этот запрос может быть удовлетворен, обработчик прерывания инициирует выполнение Dev. Во всех остальных случаях выполняется просто возврат из прерывания. Более того, если Dev уже выполняется, то никакой диспетчеризации не производится.
- Когда запрос на чтение удовлетворен, Dev копирует данные в приложение *непосредственно* из буфера сырого ввода. Как результат, данные копируются только один раз.

Эти правила - в сочетании с исключительно маленькими задержками прерывания и диспетчеризации в QNX - обеспечивают очень эффективную модель ввода.

Глава 7. Менеджер сети

Эта глава охватывает следующие темы:

- Введение
- Обязанности Менеджера сети
- Интерфейс Микроядро/Менеджер сети
- Сетевые драйверы
- Идентификаторы узла и сети
- Выбор сети
- Сеть TCP/IP

Введение

Менеджер сети (Net) дает пользователям QNX прозрачное расширение мощных возможностей механизма передачи сообщений. Взаимодействуя непосредственно с Микроядром, Менеджер сети усиливает механизм IPC на основе обмена сообщениями, передавая сообщения на удаленные компьютеры. Кроме того, Менеджер сети обеспечивает:

- повышенную пропускную способность за счет балансировки нагрузки;
- отказоустойчивость за счет резервирования соединений;
- функции моста между QNX сетями.

Обязанности Менеджера сети

Менеджер сети отвечает за распространение примитивов передачи сообщений QNX в пределах локальной сети. Стандартные примитивы передачи сообщений используются *без изменения* для связи с удаленным компьютером. Другими словами, не существует особых "сетевых" *Send(), Receive()* или *Reply()*.

Независимый модуль

Менеджер сети *не* должен быть обязательно встроен в образ операционной системы. Он может быть запущен и остановлен в любое время, чтобы обеспечить или удалить сетевые возможности передачи сообщений.

При запуске Менеджер сети регистрируется у Менеджера процессов и Ядра. Это активизирует код внутри последних, который взаимодействует с Менеджером сети. Это означает, что передача сообщений по сети и создание удаленных процессов - это не просто добавляемый поверх операционной системы слой. Сетевые возможности интегрированы в самую сердцевину примитивов передачи сообщений и управления процессами.

Такая глубокая интеграция на самом нижнем уровне обеспечивает QNX сетевую прозрачность и делает ее полностью распределенной операционной системой. Поскольку приложения запрашивают и получают доступ ко всем обслуживающим программам посредством сообщений и Менеджер сети обеспечивает прозрачное прохождение сообщений по сети, то узлы QNX функционируют вместе как единый логический компьютер.

Поскольку Менеджер сети и Микроядро отделены друг от друга, Микроядро может достичь независимости от сетевого оборудования, а компьютеры, не подключенные к сети, могут выиграть за счет меньшего объема кода.

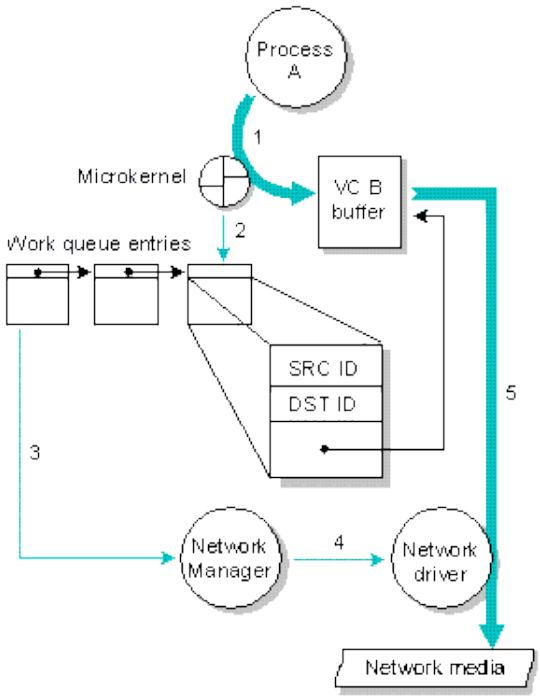
Интерфейс Микроядро/Менеджер сети

Микроядро и Менеджер процессов взаимодействуют с Менеджером сети через специальную неблокирующую очередь в памяти. Эта очередь представляет собой список передач, которые должен выполнить Менеджер сети. Элементы очереди содержат всю информацию о конкретной операции (например, *Send()*, *Reply()*, создание виртуального канала (VC), передача удаленного сигнала и т.д.).

Другим ресурсом операционной системы, используемым для обеспечения прозрачной передачи сообщений, является *буфер виртуального канала*. Выделяемый при создании VC процессом, буфер виртуального канала хранит данные до завершения операции передачи сообщения на другой узел.

Ниже приведены диаграммы, иллюстрирующие процесс посылки и приема удаленных сообщений.

Посылка сообщения на удаленный узел



Процесс вызывает Send() или Reply() к удаленному узлу

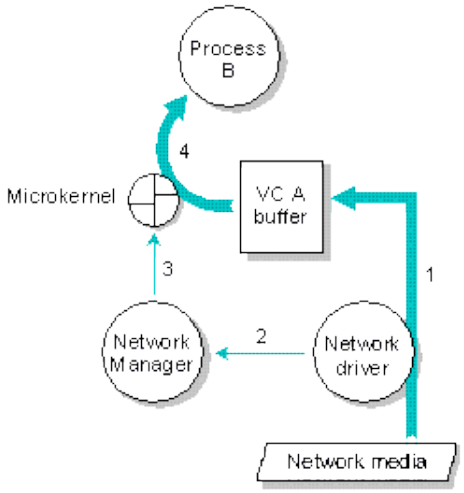
В случае вызова *Send()* или *Reply()* к удаленному узлу, имеют место следующие события:

1. Процесс вызывает *Send()* или *Reply()*, и Микроядро копирует данные из области данных процесса в буфер виртуального канала.
2. Микроядро добавляет в очередь Менеджера сети элемент, содержащий идентификаторы отправителя, удаленного получателя и указатель на данные в буфере

виртуального канала. Если перед этим очередь была пуста, то Менеджер сети получает прокси, извещающее о том, что появилась новая работа.
3. Менеджер сети извлекает элемент из очереди.
4. Менеджер сети посылает элемент соответствующему сетевому драйверу.
5. Менеджер сети начинает передачу данных по сети и отвечает за доставку.

В случае распространения сигнала или создания VC, Менеджер процессов, а не Микроядро, добавляет управляющий пакет в очередь . Как и в предыдущем случае, Менеджер сети передаст пакет по назначению.

Получение сообщения с удаленного узла



Процесс получает удаленный Send() или Reply()

Допустим, что удаленный узел послал сообщение, как описано выше. В этом случае на принимающем узле имеют место следующие события:

1. Сетевой драйвер помещает поступившие по сети данные в соответствующий буфер виртуального канала.
2. Сетевой драйвер информирует Менеджер сети о том, что прием завершен.
3. Менеджер сети использует частный вызов Ядра, извещая его о том, что прием завершен.
4. Микроядро копирует данные из буфера виртуального канала в буфер процесса (при условии, что он RECEIVE- или REPLY-блокирован на этом виртуальном канале).

Любые управляющие пакеты, которые получает Менеджер сети, немедленно переправляются Менеджеру процессов через стандартный примитив *Send()*. Эти управляющие пакеты используются для распространения сигналов и создания виртуальных каналов.

Сетевые драйверы

Подобно Менеджеру файловой системы и Менеджеру устройств, Менеджер сети не содержит аппаратно-зависимого кода. Эта функциональность обеспечивается драйверами сетевых плат.

Менеджер сети может поддерживать одновременно несколько сетевых драйверов. Каждый драйвер обычно обслуживает одну сетевую плату. Вы можете иметь драйверы/платы одного и того же типа или различных типов - например, два драйвера/платы Ethernet или, допустим, драйвер/плату Ethernet и драйвер/плату Arcnet.

Интерфейс между Менеджером сети и драйверами реализован через очереди в разделяемой памяти. Этот интерфейс разработан таким образом, чтобы достичь максимально возможной производительности. Драйвер определяет протокол, подходящий для данного сетевого носителя.

Драйвер отвечает за формирование пакетов, организацию последовательности и в случае, когда требуется гарантированная доставка данных удаленному узлу, повторную передачу. Такая архитектура позволяет QNX легко поддерживать новое сетевое оборудование и протоколы за счет написания или модификации только сетевого драйвера.

Идентификаторы узла и сети

Каждый узел в локальной сети идентифицируется двумя числами:

- физическим идентификатором (ID) узла (или идентификаторами, если узел имеет более одной сетевой платы);
- комбинацией логического ID узла и логического ID сети.

Физический ID узла

Физический ID узла определяется оборудованием. Сетевые платы обмениваются данными друг с другом, указывая физический ID удаленного узла, с которым должна быть установлена связь. В случае сети Ethernet или Token Ring - это большое число, которым неудобно оперировать людям и утилитам. Например, каждая плата Ethernet или Token Ring имеет уникальный 48-битный физический ID узла в соответствии со стандартом IEEE 802. Платы Arcnet, напротив, имеют всего лишь 8-битный ID.

Использование физического ID узла имеет существенный недостаток: при соединении некоторых сетей может возникнуть конфликт адресов (особенно в случае Arcnet), или формат адресов может радикально отличаться.

Логический ID узла

Чтобы обойти названные выше проблемы, возникающие при использовании физических ID узлов, каждому узлу QNX присваивается *логический ID узла*. Все процессы QNX оперируют логическими ID узлов. Физические ID узлов скрыты от процессов, выполняющихся в QNX.

Использование логических ID узлов упрощает лицензирование. Также они позволяют утилитам, которые опрашивают сеть, использовать простой цикл, в котором логический ID узла изменяется от 1 до количества узлов.

Соответствие между логическими и физическими ID узлов устанавливается Менеджером сети. Менеджер сети, когда поручает драйверу передать данные на другой узел, передает ему физический ID этого узла.

Логические ID узлов обычно присваиваются по порядку, начиная с 1. Например, узел, имеющий плату Ethernet, может получить логический ID 2, который будет соответствовать физическому ID узла 00:00:c0:46:93:30.

Логические ID узлов должны быть уникальны для всех узлов во *всех* соединенных QNX-сетях, чтобы сделать возможным функционирование мостов.

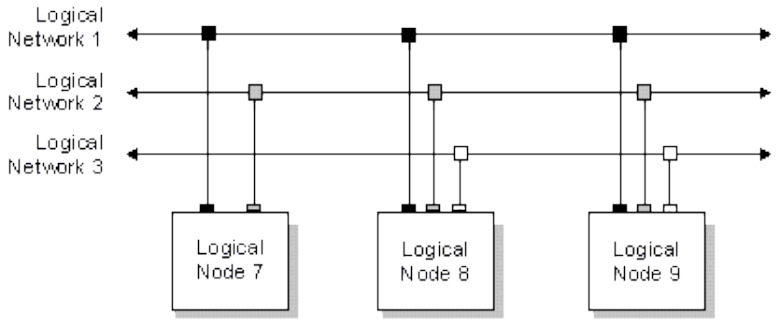
Логический ID сети

ID сети идентифицирует конкретную *логическую сеть*. Логическая сеть - это любое оборудование, которое позволяет сетевому драйверу непосредственно взаимодействовать с сетевым драйвером на другом узле. Это может быть как просто последовательный порт, так и сложная сеть Ethernet с *аппаратными* мостами.

На следующей диаграмме узел 7 имеет две сетевые платы, которые позволяют ему иметь доступ к узлам в логических сетях 1 и 2. Узлы 8 и 9 имеют по три платы, подключающие их к сетям 1, 2 и 3.

Заметьте, что все логические ID узлов уникальны для всех трех логических узлов.

Примечание. Логические ID узлов сети назначаются системным администратором. Более подробно см. в главе "Установка сети" в книге "Руководстве пользователя".



Несколько физических сетей сосуществуют посредством логических сетей

Выбор сети

В случае, когда узлы соединены более чем одной логической сетью, Менеджер сети может выбирать, какую из сетей использовать для передачи к удаленному узлу. Например, на приведенном выше рисунке узел 7 может передавать данному узлу 8, используя либо сеть 1, либо сеть 2.

Распределение нагрузки

Пропускная способность сети определяется совокупностью скорости компьютера и скорости сети. Если компьютер может выдавать данные быстрее, чем сеть может их передавать, то сеть будет ограничивать пропускную способность.

Например, два компьютера Pentium, соединенные сетью 10BASE-T Ethernet, будут

ограничены 1.1 миллионом байт в секунду, то есть скоростью передачи данных, обеспечиваемой сетевым оборудованием. Однако если поместить по две платы Ethernet в каждый из компьютеров и соединить их отдельными кабелями, то Менеджер сети сможет передавать данные по обеим сетям одновременно. При большой нагрузке это обеспечит увеличение пропускной способности в два раза по сравнению с одной сетью.

Менеджер сети будет пытаться сбалансировать нагрузку, выбирая сетевой драйвер. В рассмотренном выше примере, если производится передача с узла 7 на узел 8 по сети 1 и другая передача на узел 8 инициируется на узле 7, то сеть 2 будет *автоматически* выбрана для передачи данных.

Отказоустойчивость

Когда узлы соединены двумя и более сетями, то существует больше чем один возможный путь для связи. В случае отказа платы в одной из сетей, когда связь по этой сети невозможна, Менеджер сети автоматически перенаправит все данные через другую сеть. Это происходит *на лету* без какого-либо вмешательства со стороны прикладных программ и обеспечивает прозрачную отказоустойчивость сети. Если кабели различных сетей проложены раздельно, то вы также будете защищены от случайного обрыва кабеля.

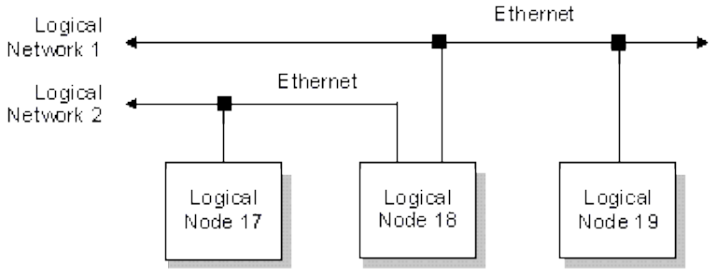
Вы также можете проектировать системы-"тандемы", в которых две машины соединены высокоскоростной сетью для нормальной работы и другой, более дешевой и медленной сетью (например, по последовательному каналу), которая служит резервом. В случае отказа первой сети соединение не оборвется, хотя пропускная способность, конечно же, снизится.

Мосты между сетями QNX

Менеджер сети позволяет любому узлу играть роль моста между двумя отдельными сетями QNX, базирующимися на стандарте IEEE 802.

Примечание. Так как QNX использует одинаковый формат пакетов и протокол на всех IEEE 802 сетях, можно создавать мосты между сетями Ethernet, Token Ring и FDDI. Для сетей Arcnet *нельзя* создавать мосты.

Рассмотрим следующую диаграмму, где одной сети принадлежат узлы 17 и 18, а другой - узлы 18 и 19:



Мост между двумя IEEE 802 QNX сетями

Узлы 17 и 18 находятся в одной сети, поэтому они могут общаться друг с другом напрямую. То же справедливо для узлов 18 и 19. Но как могут общаться узлы 17 и 19?

Так как *обе* локальные сети базируются на IEEE 802, узел 18 автоматически перенаправляет пакеты, позволяя узлам 17 и 18 создать виртуальный канал. Хотя они и не подключены к одной и той же локальной сети, узлы 17 и 19, тем не менее, могут общаться друг с другом.

Сеть TCP/IP

Присущая QNX поддержка сети реализует локальную сеть на основе собственного *частного* протокола и оптимизирована для организации интерфейса между QNX компьютерами. Но для связи с не-QNX системами, QNX использует ставший *промышленным стандартом* набор протоколов, называемый TCP/IP.

По мере того как Интернет стал занимать все большее место в нашей повсеместной жизни, протокол, на котором он основан - IP (Internet Protocol) - приобретает все большее значение. Даже если вы не подключаетесь непосредственно к Интернет как к таковому, IP протокол и связанный с ним инструментарий поистине вездесущи, делая IP стандартом "де-факто" для многих частных сетей.

IP используется везде, начиная от простых задач (например, удаленный вход в систему (login)) и до более сложных (например, отслеживание биржевых котировок в реальном времени). Все больше и больше компаний используют World Wide Web ("всемирную паутину"), основанную на IP, для переписки с клиентами, рекламы и другой деловой активности.

Менеджер TCP/IP

Менеджер TCP/IP в QNX происходит из Berkley BSD 4.3, который является наиболее распространенным стеком TCP/IP в Интернет и использован как основа для многих систем.

Сокет API

Библиотека BSD сокета API была очевидным выбором для QNX 4. Сокет API является стандартным API для программирования TCP/IP в среде Unix. В среде Windows, Winsock API базируется на BSD сокете API. Это облегчает переход между ними.

Имеются все процедуры, которые могут понадобиться прикладным программистам:

accept()

bind()

bindresvport()

connect()

dn_comp()

dn_expand()

endprotoent()

endservent()

gethostbyaddr()

gethostbyname()

getpeername()

getprotobyname()

getprotobynumber()

getprotoent()

getservbyname()

getservent()

getsockname()

getsockopt()

herror()

hstrerror()

htonl()

htons()

h_errlist()

h_errno()

h_nerr()

inet_addr()

inet_aton()

inet_lnaof()

inet_makeaddr()

inet_netof()

inet_network()

inet_ntoa() ioctl()

listen()

ntohl()

ntohs()

recv()

recvfrom()

res_init()

res_mkquery()

res_query()

res_querydomain()

res_search()

res_send()

select()

send()

sendto()

setprotoent()

setservent()

setsockopt()

shutdown()

socket()

Распространенные утилиты и демоны из Интернет могут быть легко перенесены или просто перекомпилированы в такой среде. Это облегчает использование имеющихся готовых наработок.

Возможность взаимодействия сетей

При разработке Менеджера TCP/IP в QNX в первую очередь принималась во внимание возможность взаимодействия сетей. Учитывались как требования RFC, так и реальные условия. Менеджер TCP/IP охватывает всю функциональность, предлагаемую RFC 1122. Также поддерживаются протоколы ARP, IP, ICMP, UDP и TCP.

NFS

Network File System (NFS) является приложением TCP/IP, реализованным на большинстве DOS и Unix систем. NFS позволяет отображать удаленные файловые системы - или их части - в локальное пространстве имен. Файлы на удаленной системе показываются как часть локальной файловой системы QNX.

Примечание. В QNX 4 для поддержки NFS требуется менеджер Socket. Учтите, что "облегченная" версия менеджера, Socklet, может быть использована, если нет необходимости в NFS.

SMB

Server Message Block (SMB), который используется многими различными серверами, такими как Windows NT, Windows 95, Windows for Workgroups, LAN Manager и Samba. SMBfsys позволяет клиенту QNX прозрачный доступ к удаленным дискам на таких серверах.

Глава 8. Оконная система Photon microGUI

Эта глава охватывает следующие темы:

- Графическое микроядро
- Пространство событий
- Графические драйверы
- Масштабируемые шрифты
- Многоязычная поддержка Unicode
- Поддержка анимации
- Поддержка печати
- Менеджер окон Photon
- Библиотека виджетов
- Резюме

Графическое микроядро

Многие встроенные системы нуждаются в пользовательском интерфейсе. Для сложных приложений или для максимальной простоты использования, естественным выбором является графическая оконная система. Однако оконные системы настольных ПК требуют слишком много системных ресурсов для практического применения во встроенных системах, где память и стоимость ограничены.

При создании оконной системы Photon microGUI была применена архитектура микроядра, успешно воплощенная в QNX для создания POSIX ОС для встроенных систем.

Для успешной реализации ОС на основе микроядра в первую очередь было необходимо добиться максимальной эффективности IPC (так как от IPC зависит производительность всей ОС). Благодаря воплощенному в QNX механизму IPC с низкими издержками, стало возможным создание структуры GUI как графического "микроядра", окруженного командой взаимодействующих процессов, общающихся через IPC.

Хотя на первый взгляд это может показаться похожим на построение графической системы по классической схеме клиент/сервер, используемой в X Window System, архитектура Photon отличается за счет ограничения функциональности, реализуемой внутри самого графического микроядра (или сервера), и распределения большей части функций GUI между взаимодействующими процессами.

Микроядро Photon выполняется как маленький процесс (размер кода 45K), реализуя только несколько фундаментальных примитивов, которые внешние опциональные процессы используют для построения более высокого уровня функциональности оконной системы. По иронии, для самого микроядра Photon "окна" не существуют. Микроядро Photon не может также "рисовать" что-либо или управлять мышью либо клавиатурой.

Для управления средой GUI, Photon создает 3-мерное "пространство событий" и ограничивается только оперированием регионами и выполнением отсечения и направления различных событий по мере их прохождения сквозь регионы в этом пространстве событий.

Эта абстракция напоминает концепцию микроядра ОС, которое не поддерживает функции

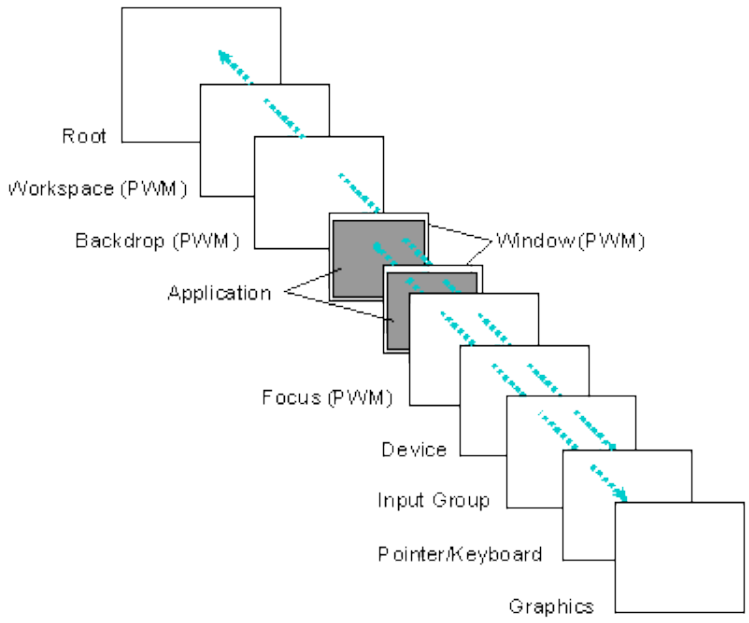
ввода/вывода для устройств или файловой системы, а полагается на внешние процессы, предоставляющие эти услуги высокого уровня. Это обеспечивает масштабируемость ОС и GUI, построенных на основе микроядра, по размеру и функциональности .

В основе "абстракции" микроядра Photon лежит воображаемое графическое *пространство событий*, в которое другие процессы могут помещать *регионы*. Используя QNX IPC для связи с микроядром Photon, эти процессы управляют своими регионами для предоставления графических сервисных функций высокого уровня или для выполнения функций пользовательских приложений. Для систем с ограниченными ресурсами Photon может масштабироваться "вниз" за счет *удаления* процессов, предоставляющих сервисные функции, а за счет *добавления* процессов, предоставляющих сервисные функции, Photon может масштабироваться "вверх" до полнофункциональной настольной системы.

Пространство событий Photon

"Пространство событий" можно представить как пустое трехмерное пространство с "корневым регионом" на заднем плане. Пользователи как будто "смотрят внутрь" этого пространства событий. Приложения помещают регионы в трехмерное пространство между корневым регионом и пользователем; они используют эти регионы для генерации и приема различных типов событий в этом пространстве.

Процессы, которые выполняющие обслуживание драйверов устройств, помещают регионы на передний план пространства событий. В дополнение к управлению пространством событий и корневым регионом, микроядро Photon поддерживает экранный указатель (курсор), проецируемый как *события рисования* по направлению к пользователю.



Photon использует последовательность регионов, начиная от корневого региона на заднем плане пространства событий до графического региона спереди. События рисования двигаются от регионов приложений к графическому региону. События ввода возникают в регионе курсора/клавиатуры и двигаются по направлению к корневому региону

Двигающиеся в пространстве событий события можно представить себе как "фотоны" (что и дало название оконной системе). Сами события состоят из набора прямоугольных областей и прикрепленных к ним данных. По мере движения событий в пространстве событий их прямоугольники пересекают регионы, принадлежащие различным процессам (приложениям).

Про события, которые двигаются от корневого региона, говорят, что они перемещаются наружу (по направлению к пользователю), в то время как про события от пользователя говорят, что они двигаются внутрь, по направлению к корневому региону.

Взаимодействие между событиями и регионами лежит в основе ввода и вывода в Photon. События мыши, клавиатуры и светового пера двигаются от пользователя к корневому региону, с "прикрепленным" к ним положением курсора. События рисования возникают в регионах и двигаются по направлению к региону устройства и пользователю.

Регионы

Каждому региону соответствует прямоугольная область, определяющая его положение в 3-мерном пространстве событий. Регион также имеет атрибуты, определяющие, как он взаимодействует с различными классами событий при пересечении ими региона. Взаимодействие региона с событиями определяется двумя битовыми масками:

- маска *чувствительности*;
- маска *непрозрачности*.

Маска чувствительности определяет, должен ли процесс-владелец региона оповещаться о пересечении региона тем или иным событием. Каждый бит маски чувствительности определяет, чувствителен ли регион к определенному типу событий. Когда событие пересекает регион, для которого установлен бит (равен 1), копия этого события помещается в очередь процесса-владельца региона, извещая приложение о прохождении события через регион. Такое извещение никак не изменяет само событие.

Маска непрозрачности определяет прозрачность региона для тех или иных событий. Каждый бит этой маски определяет, является ли регион прозрачным для определенного типа события. При прохождении события сквозь "непрозрачный" регион, оно модифицируется.

Эти две битовые маски могут быть совместно использованы для достижения различных результатов. Возможны следующие четыре сочетания для региона:

Сочетание битовых масок:	**Описание:**
Нечувствительный, прозрачный	При прохождении события через регион, оно не модифицируется, и владелец региона не извещается. Процесс-владелец региона просто не интересуется событием
Нечувствительный, непрозрачный	При прохождении события через регион, оно отсекается; владелец региона не извещается. Большинство приложений используют такую комбинацию атрибутов для отсечения событий рисования, чтобы избежать перерисовки окна событиями рисования,

Сочетание битовых масок:	**Описание:**
	исходящими от лежащих под ним окон
Чувствительный, прозрачный	Копия события направляется владельцу региона; событие продолжит движение в пространстве событий, не изменяясь. Процесс, желающий регистрировать прохождение всех событий, может использовать такую комбинацию
Чувствительный, непрозрачный	Копия события направляется владельцу региона; событие отсекается регионом. Установив такую комбинацию масок, событие может играть роль фильтра или преобразователя. Приложение может обработать любое полученное событие, регенерировать его и при необходимости преобразовать его каким-либо образом при этом, возможно, изменив направление движения или координаты. Например, регион может поглощать события светового пера, выполнять распознавание почерка, а затем генерировать эквивалентные события нажатия клавиш

События

Подобно регионам, события могут относиться к различным классам и иметь различные атрибуты, как, например:

- регион происхождения;
- тип;
- направление;
- прикрепленный список прямоугольников;
- специфические для данного события данные.

В отличие от большинства оконных систем, Photon классифицирует *не только* ввод (перо, мышь, клавиатура т.д.), но и вывод (запросы рисования) как события. События могут генерироваться как регионами, которые процессы поместили в пространство событий, так и самим микроядром Photon. Определены следующие типы событий:

- нажатия клавиш;
- нажатия пера и кнопок мыши;
- перемещения пера и мыши;
- пересечение границ региона;
- события экспозиции и перекрытия;
- события рисования;
- события перетаскивания (drag).

Приложения могут либо ждать наступления событий, и при этом блокироваться, либо получать асинхронные извещения о приходе события.

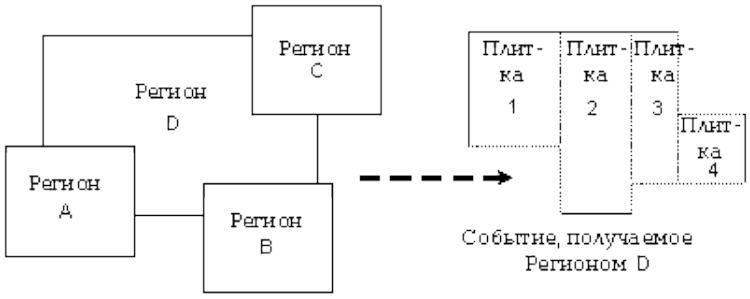
Список прямоугольников, прикрепленный к событию, может определять одну или более прямоугольных областей, либо "исходную точку" - единственный прямоугольник, у которого координаты верхнего левого и нижнего правого углов совпадают.

При пересечении событием непрозрачного региона, прямоугольник региона "вырезается" из списка прямоугольников события так, что список описывает теперь только видимую часть события.

Лучше всего иллюстрирует такое отсечение то, как изменяется список прямоугольников события рисования по мере его прохождения сквозь различные регионы. Когда событие рисования генерируется, список прямоугольников содержит единственный прямоугольник, описывающий породивший событие регион.

Если событие проходит через регион, который отсекает, например, верхний левый угол события рисования, то список прямоугольников модифицируется и будет содержать уже два прямоугольника, которые определяют область, подлежащую отрисовке. Эти результирующие прямоугольники называются "плитки" (tiles).

Подобным образом, каждый раз при пересечении событием рисования непрозрачного региона, список прямоугольников будет модифицироваться таким образом, чтобы описывать область, оставшуюся видимой после "вырезания" непрозрачного региона. Когда, наконец, событие рисования достигнет графического драйвера, то список прямоугольников будет точно описывать только его видимую часть.



Непрозрачные для события рисования регионы вырезаются, в результате чего получается область, состоящая из прямоугольных "плиток"

В том случае, если событие рисования целиком отсекается при пересечении с регионом, оно прекращает существование. Этот механизм "непрозрачных" окон, изменяющих список прямоугольников события рисования, обеспечивает правильное отсечение событий рисования по мере их продвижения от исходного региона (и связанного с ним процесса) к пользователю.

Графические драйверы

Графические драйверы реализованы как процессы, которые помещают регион на переднем плане пространства событий. Регион графического драйвера *чувствителен* к событиям

рисования, исходящим из пространства событий. Графический драйвер получает события рисования, когда они пересекают его регион. Можно представить себе, что регион покрыт "фосфором", который светится при попадании "фотонов".

Так как API рисования Photon накапливает запросы рисования в пакеты, посылаемые как одно событие рисования, то каждое событие рисования, получаемое драйвером, содержит список графических примитивов, подлежащих отрисовке. К моменту пересечения событием рисования региона драйвера, список прямоугольников будет содержать также "список отсечений", описывающий, какие именно части списка рисования должны отображаться на дисплее. Работа драйвера заключается в том, чтобы преобразовать результирующий список в визуальное отображение на контролируемом графическом оборудовании.

Одно из преимуществ использования списка прямоугольников внутри события состоит в том, что каждое передаваемое драйверу событие представляет собой фактически "пакет" запросов. По мере совершенствования графического оборудования, все больше и больше такой "пакетной" работы может передаваться непосредственно оборудованию. Многие видеоадаптеры уже поддерживают аппаратно одну область отсечения, а некоторые поддерживают и несколько областей.

Хотя использование механизма QNX IPC для передачи запросов рисования от приложений к графическому драйверу и может показаться неприемлемой избыточностью, тесты производительности показывают, что производительность в данном варианте не хуже, чем в случае, когда приложения выполняют прямые вызовы драйвера. Одной из причин является то, что при использовании событий многочисленные запросы рисования группируются, что уменьшает количество посылаемых сообщений по сравнению с количеством прямых вызовов драйвера.

Несколько графических драйверов

Из того, что графический драйвер просто помещает регион в пространство событий Photon, естественно следует, что одновременно могут быть запущены несколько графических драйверов для нескольких видеоадаптеров, при этом каждый драйвер будет иметь свой, чувствительный к событиям рисования, регион.

Эти регионы могут быть расположены рядом, либо перекрывать друг друга произвольным образом. Так как Photon наследует от QNX сетевую прозрачность, то приложения или драйверы Photon могут выполняться на любом узле сети, позволяя, таким образом, дополнительным графическим драйверам расширять графическое пространство Photon за счет физических дисплеев других компьютеров в сети. За счет перекрытия регионов графических драйверов, события рисования могут дублироваться на нескольких экранах.

Многие интересные приложения стали возможны благодаря этим свойствам Photon. Например, на заводе оператор с портативным компьютером, имеющим беспроводное подключение к сети, может подойти к рабочей станции и "перетащить" панель управления с ее монитора на экран портативного компьютера, а затем перейти в цех и осуществлять управление.

В других приложениях встроенная система без пользовательского интерфейса может проецировать дисплей на любой из узлов сети. Кроме того, становится возможным коллективный режим работы - несколько человек, находясь за своими компьютерами, могут

одновременно видеть одни и те же окна и работать с одним и тем же приложением.

С точки зрения приложения, это выглядит как одно единое графическое пространство. С точки зрения пользователя, это выглядит как группа соединенных компьютеров, где можно перетаскивать окна с одного физического экрана на другой.

Цветовая модель

Для представления цветов используется 24-битная RGB модель (по 8 бит для красного, зеленого и синего), что обеспечивает 16,777,216 цветов. В зависимости от используемого типа оборудования, драйвер либо непосредственно отображает 24-битный цвет, либо использует различные варианты смешивания цветов, чтобы отобразить требуемый цвет на оборудовании, поддерживающем меньшее число цветов.

Так как графические драйверы используют аппаратно-независимое представление цветов, то приложения могут работать без изменения на различном оборудовании, независимо от того, какую цветовую модель оно поддерживает. Это позволяет "перетаскивать" приложения с одного монитора на другой, не задумываясь о том, какая цветовая модель аппаратно реализована в каждом конкретном случае.

Масштабируемые шрифты

В дополнение к поддержке растровых шрифтов, Photon также предлагает масштабируемые шрифты. Эти шрифты могут масштабироваться практически с любым размером точки и использовать технологию сглаживания (16 оттенков) для четкого и ясного отображения на экране с любым разрешением.

Масштабируемые шрифты в Photon поддерживаются быстродействующим сервером шрифтов, который загружает описания шрифтов, хранящиеся в сжатом виде в файлах *.pfr (Portable Font Resource, ресурсы переносимых шрифтов), и затем приводит вид символов в соответствие с любым размером точки и разрешением. Стоит отметить, что формат PFR обеспечивает более чем в два раза лучшее сжатие по сравнению с PostScript шрифтами.

Наборы шрифтов

Основной латинский набор

Основной латинский (Core Latin) набор шрифтов Photon (latin1.pfr), который охватывает два набора символов стандарта Unicode, *Basic Latin* (U+0000 - U+007F) и *Latin-1 Supplement* (U+0080 - U+00FF), включает следующие масштабируемые шрифты:

- Dutch;
- Dutch Bold;
- Dutch Italic;
- Dutch Bold Italic;
- Swiss;
- Swiss Bold;
- Swiss Italic;

- Swiss Bold Italic;
- Courier;
- Courier Bold;
- Courier Italic;
- Courier Bold Italic.

Расширенный латинский набор

Расширенный латинский (Extended Latin) набор (latinx.pfr) охватывает наборы символов Unicode "Latin Extended-A *(U+0100 - U+017F) и* Latin Extended-B" (U+0180 - U+0217) и включает следующие шрифты:

- Dutch;
- Dutch Bold;
- Dutch Italic (генерируется алгоритмически);
- Dutch Bold Italic (генерируется алгоритмически);
- Swiss;
- Swiss Bold;
- Swiss Italic (генерируется алгоритмически);
- Swiss Bold Italic (генерируется алгоритмически).

Поддерживаемые языки

Имея в своем распоряжении Основной латинский набор (latin1.pfr), разработчик может поддерживать множество языков, включая:

- Датский;
- Голландский;
- Английский;
- Финский;
- Фламандский;
- Французский;
- Немецкий;
- Гавайский;
- Исландский;
- Индонезийский;
- Ирландский;
- Итальянский;
- Норвежский;
- Португальский;
- Испанский;
- Суахили;
- Шведский.

Расширенный набор (`latinx.pfr`) позволяет дополнительно поддерживать:

- Африканский;
- Баскский;
- Каталонский;
- Хорватский;
- Чешский;
- Эсперанто;
- Эстонский;
- Гренландский;
- Венгерский;
- Латышский;
- Литовский;
- Мальтийский;
- Польский;
- Румынский;
- Словацкий;
- Турецкий;
- Валлийский.

Дополнительные языковые пакеты

Для Photon предлагаются несколько дополнительных пакетов для поддержки национальных языков:

- *Японский*;
- *Китайский*;
- *Корейский*;
- *Кириллица*.

Многоязычная поддержка Unicode

Photon разработан с учетом поддержки национальных символов. Следуя стандарту Unicode (ISO/IEC 10646), Photon предоставляет разработчикам возможность создавать приложения, поддерживающие основные мировые языки.

Unicode основывается на наборе символов ASCII, но использует 16-битную кодировку для полной поддержки многоязычного текста. Нет никакой необходимости прибегать к escape-последовательностям или управляющим кодам для задания любого символа любого языка. Заметьте, что кодировка Unicode обрабатывает все символы - алфавитные, идеограммы, специальные символы - абсолютно одинаковым образом.

UTF-8 кодировка

Известная раньше как UTF-2, UTF-8 (от "8-битная форма") кодировка определяет использование символов Unicode в 8-битной среде UNIX.

Вот некоторые основные характеристики UTF-8:

- Unicode-символы от U+0000 до U+007E (набор ASCII) отображаются в UTF-8-байты от 00 до 7E (ASCII-значения);
- ASCII-значения не встречаются иным образом в UTF-8, обеспечивая полную совместимость с файловыми системами, которые анализируют ASCII-байты;
- UTF-8 упрощает преобразование в Unicode-текст и из него;
- Первый байт указывает количество байт в многобайтной последовательности, обеспечивая эффективный разбор;
- Можно легко найти начало символа из любого места в потоке байт, так как для этого необходимо перебрать не более четырех байт, а начальный байт легко определить. Например: isInitialByte = ((byte & 0xC0) != 0x80);
- UTF-8 достаточно компактен, имея в виду количество байт, используемых для кодировки.

Системная библиотека включает ряд функций преобразования:

Функция:	**Описание:**
mblen()	Длина многобайтной строки в символах
mbtowc()	Преобразовать многобайтный символ в двухбайтный символ
mbstowcs()	Преобразовать многобайтную строку в двухбайтную строку
wctomb()	Преобразовать двухбайтный символ в его многобайтное представление
wcstombs()	Преобразовать строку двухбайтных символов в многобайтную строку

В дополнение к перечисленным выше функциям, разработчики могут также воспользоваться собственной библиотекой Photon, функциями *PxTranslate*, которые выполнят различные преобразования наборов символов в/из UTF-8.

Поддержка анимации

Photon обеспечивает немерцающую анимацию через специальный виджет-контейнер с "двойным буфером" (PtDBContainer), который создает специальный контекст в памяти для отрисовки изображений.

Виджет PtDBContainer использует блок разделяемой памяти, достаточный для хранения изображения соответствующего размера.

Поддержка печати

Photon предусматривает встроенную поддержку печати с выводом на различные устройства, включая:

- файлы битовых карт;
- PostScript;
- Hewlett-Packard PCL;
- Epson ESC/P2;Canon;Lexmark.

Photon также содержит виджет/диалог выбора принтера, облегчая печать из приложений.

Менеджер окон Photon

Добавление Менеджера окон превращает Photon в полнофункциональный настольный графический интерфейс (GUI). Менеджер окон не является обязательным и может отсутствовать в большинстве встроенных систем. Менеджер окон позволяет пользователям манипулировать окнами приложений, изменяя их размер, перемещая и минимизируя.

Менеджер окон использует концепцию фильтрации событий. Он помещает дополнительные регионы за регионами приложений, на которых "нарисованы" элементы управления окнами. Так как вид и поведение интерфейса определяются заменяемым Менеджером окон, то можно реализовать различные виды пользовательских интерфейсов.

Библиотека виджетов

Photon предлагает библиотеку компонентов, называемых *виджетами*,- объектов, способных, в основном, автоматически управлять своим поведением без непосредственного программирования. В результате, завершенное приложение может быть быстро собрано из виджетов, с последующей привязкой C-кода к соответствующим callback-функциям виджетов. Photon Application Builder (PhAB), который является частью системы разработки Photon, поддерживает широкую палитру виджетов в визуальной среде разработки.

Photon предлагает широкий спектр виджетов:

- базовые виджеты (например, кнопка);
- виджеты-контейнеры (например, окно);
- сложные виджеты (например, HTML-виджет).

Базовые виджеты

Виджет Ярлык (PtLabel)

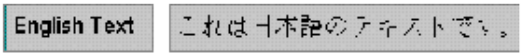
Данный виджет используется в основном для отображения текстовой информации. Виджет PtLabel является надклассовым для всех текстовых виджетов, обеспечивая многие настраиваемые атрибуты (например, шрифт, всплывающие баллоны, цвета, бордюр, выравнивание, поля и т.д.), которые наследуются всеми подклассами.

Виджет Кнопка (PtButton)

Кнопки являются неотъемлемым компонентом любой оконной оболочки. Обычно они имеют выпуклый вид, который при нажатии сменяется на утопленный, визуально отражая выбор кнопки. В дополнение к визуальному отражению состояния, при выборе кнопки

автоматически вызывается определенная приложением callback-функция.

Виджеты Текстовый ввод (PtText, PtMultiText)



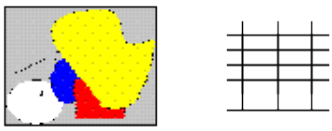
Photon предлагает два виджета текстового ввода:

- простой однострочный (PtText), обычно используемый в формах;
- многострочный, с мощными возможностями редактирования виджет (PtMultiText), обеспечивающий все возможности редактирования, автоматической прокрутки, поддержку множества шрифтов.

Виджеты Кнопка с фиксацией (PtToggleButton, PtOnOffButton)

Кнопки с фиксацией отображают одно из двух возможных состояний - включено или выключено. Photon предлагает два различных типа кнопок с фиксацией с различным внешним видом. Кнопки с фиксацией используются для отображения или ввода информации о состоянии какой-либо команды или действия.

Графические виджеты (PtArc, PtPixel, PtRectangle, PtLine, PtPolygon, PtEllipse, PtBezier, PtGrid)



Photon не испытывает недостатка в графических виджетах. Существуют виджеты для всего, начиная от простых линий и прямоугольников и до сложных многосегментных кривых Безье. Графические виджеты имеют атрибуты для цветов, заполнения, толщины линий и много другого.

Виджет Полоса прокрутки (PtScrollbar)

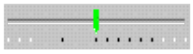
Данный виджет используется для прокрутки изображения в видимой области. Полоса прокрутки также используется в составе других виджетов (например, PtList, PtScrollArea) для обеспечения прокрутки.

Виджет Разделитель (PtSeparator)

Данный виджет используется для разделения двух или более областей, придавая лучший

внешний вид. Разделитель может быть настроен в соответствии с различными стилями и видами.

Виджет Движок (PtSlider)

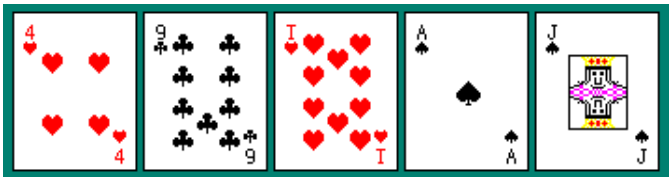


Движок отличается от полосы прокрутки тем, что полоса прокрутки определяет интервал, в то время как движок задает единственное значение. Виджет Движок предусматривает большой список настраиваемых атрибутов.

Виджет Таймер (PtTimer)

Виджет Таймер существенно упрощает использование таймера. Этот виджет не отображается визуально - он просто определяет callback-функцию, вызываемую всякий раз при срабатывании таймера. Приложение может устанавливать значение таймера и, по выбору, интервал повторения.

Виджеты Графическое изображение (PtBitmap, PtLabel, PtButton)

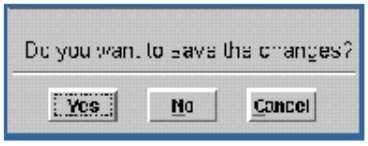


Photon поддерживает все основные форматы графических файлов, что позволяет импортировать изображения и показывать их внутри виджетов. Многие виджеты Photon непосредственно поддерживают отображение графики - наиболее используемыми являются PtButton, для создания панелей кнопок, и PtLabel, для показа изображений.

Виджет Индикатор хода процесса (RtProgress)

Если приложению необходимо выполнить какую-либо длительную операцию (например, загрузить файл), оно может использовать индикатор хода процесса, чтобы показать пользователю, что происходит и, что еще более важно, сколько еще времени займет этот процесс. Индикатор хода процесса может быть горизонтальным или вертикальным и имеет много настраиваемых атрибутов.

Виджет Сообщение (PtMessage)



Всплывающие сообщения и предупреждения типичны для оконной среды. Photon предусматривает очень удобный виджет диалога, который показывает сообщение, и до 3 кнопок для ответа пользователя. Имеется также полезная функция вызова модального диалога (*PtAskQuestion()*), основанная на данном виджете.

Числовые виджеты (PtNumericInteger, PtNumericFloat)

PtNumericInteger позволяет пользователю задать целочисленное значение в пределах между установленными минимальной и максимальной величинами. PtNumericFloat позволяет ввести число с плавающей точкой.

Виджет PtUpDown (стрелки вверх/вниз) позволяет пользователю увеличивать или уменьшать число на заданную величину.

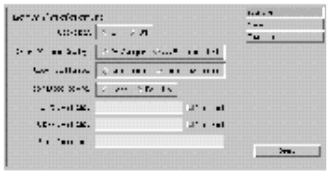
Виджеты-контейнеры

Виджеты Окно и Пиктограмма (PtWindow, PtIcon)

Окна являются основными контейнерами для приложений. Основные компоненты пользовательского интерфейса (линейки меню, линейки инструментов и т.д.) появляются с виджетом Окно. Виджет автоматически выполняет все необходимые взаимодействия с Менеджером окон Photon (PWM) - вам требуется только задать требуемую функциональность.

Виджеты Пиктограммы тесно связаны с окнами и показываются в папках Photon Desktop Manager и на панели задач PWM.

Виджет Панель (PtPane)



Виджеты Панель являются простыми виджетами-контейнерами, которые используются для размещения других виджетов. Хотя он и является родительским виджетом, панель никаким образом не управляет дочерними виджетами. Панели очень удобны для построения форм, обычно встречающихся в окнах диалога.

Виджет Группа (PtGroup)

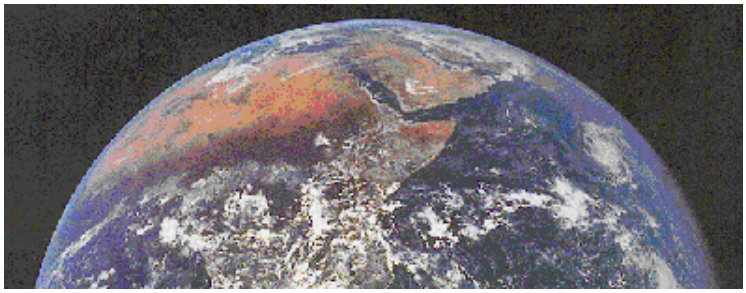
Виджет Группа - это очень мощный виджет, который управляет геометрией всех своих дочерних виджетов. Он может выравнивать виджеты по горизонтали, по вертикали или в виде матрицы. Группа может быть привязана к стороне любого другого контейнера (например, окна) таким образом, чтобы группа автоматически изменяла размер при изменении размера окна. Виджет Группа также предусматривает атрибуты, которые позволяют задать необходимость растягивания дочерних виджетов при увеличении размеров группы.

Виджет Область прокрутки (PtScrollArea)

Область прокрутки обеспечивает "окно" просмотра содержимого контейнера потенциально большего размера. Вы можете поместить любое количество виджетов внутрь области прокрутки, и она автоматически создаст полосы прокрутки в случае, если виджеты выходят за границы видимой области. Области прокрутки могут быть использованы для создания окна просмотра текстовых файлов, текстовых редакторов, просмотра списков и так далее.

Для быстрой прокрутки дочерних виджетов область прокрутки использует аппаратный блиттер (при условии, что он поддерживается графическим драйвером).

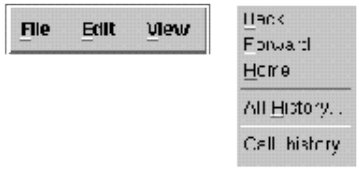
Виджет Фон (PtBkgd)



Данный виджет позволяет создавать любой фон, начиная от простого перехода цветов до симметрично расположенных текстур. Практически любое требование к фону может быть удовлетворено этим виджетом.

Сложные виджеты

Виджеты Меню (PtMenu, PtMenuBar, PtMenuButton)



Photon предусматривает все необходимое для организации меню. Виджет PtMenuBar упрощает создание стандартной линейки меню. Виджет PtMenu обрабатывает отображение всплывающего меню, нажатие-перемещение-отпускание (мыши), указание и нажатие, ввод с клавиатуры и выбор пунктов меню. Виджет PtMenuButton используется для создания отдельных пунктов меню.

Виджет Список (PtList)

Данный виджет управляет списком элементов. Он предусматривает много различных режимов выбора, включая единичный выбор, множественный выбор и выбор диапазона. Виджет Список также поддерживает многостолбцовые списки при использовании виджета PtDivider.

Виджет Разворачиваемый список (PtComboBox)

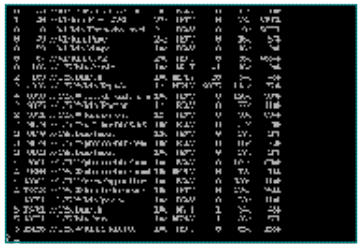
Разворачиваемый список совмещает виджет PtText (для ввода текста) с кнопкой для отображения виджета PtList. При выборе пользователем элемента списка, виджет Текст автоматически обновляется в соответствии с текущим выбором. Разворачиваемый список очень полезен в для отображения списка в ограниченном пространстве. Диалоги и контейнеры занимают значительно меньше места на экране, что особенно важно для встроенных приложений.

Виджет Дерево (PtTree)

Виджет Дерево напоминает виджет Список - они имеют общих предшественников. Основное отличие состоит в том, что виджет Дерево показывает элементы в виде иерархии. Элементы, называемые ветвями, могут быть развернуты или сжаты; может быть создано любое количество ветвей. Для каждой ветви можно определить свое уникальное графическое изображение.

В число приложений Photon, использующих деревья, входят: Файл-Менеджер (показ каталога), PhAB (иерархия виджетов), vsin (список процессов) и многие другие.

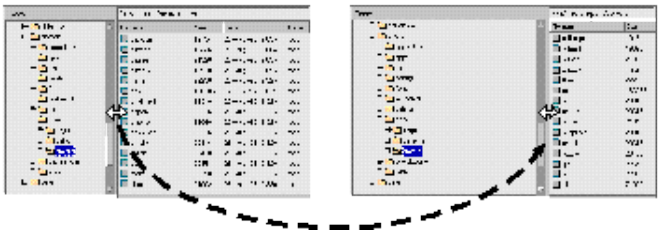
Виджеты Терминал (PtTty, PtTerminal)



Благодаря этому виджету есть возможность поместить текстовую консоль в свое приложение. Виджет Терминал создает текстовый терминал и управляет им.

Более того - он обеспечивает полную функциональность "cut-and-paste" и быстрый вызов справки путем выделения текста внутри виджета.

Виджет Делитель (PtDivider)

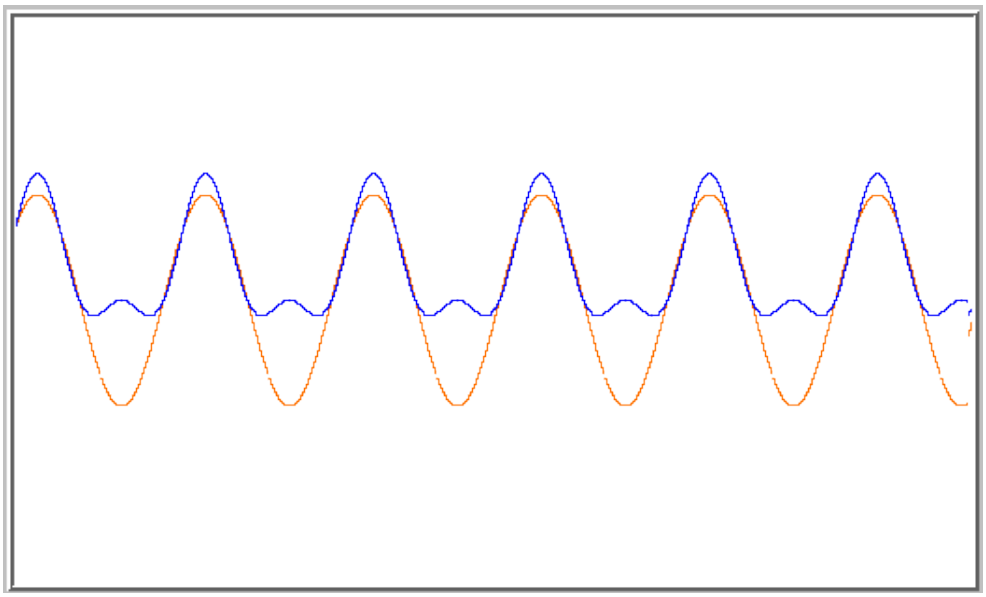


Этот виджет осуществляет управление дочерними виджетами уникальным образом. Если поместить два или более виджета внутрь виджета PtDivider, то он автоматически создает небольшие разделители между дочерними виджетами. Передвигая эти разделители, пользователь может изменять размеры дочерних виджетов. Это очень удобно, в частности,

для создания списков со столбцами изменяемой ширины. Фактически, если поместить виджет PtDivider внутрь PtList, это автоматически превратит простой список в список с множественными столбцами изменяемой ширины.

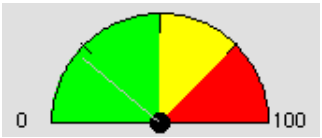
Витжеты Делители не ограничиваются только этикетками или кнопками. Любой виджет может быть помещен внутрь, чтобы создавать рядом деревья с изменяемыми размерами, области прокрутки и так далее.

Виджет Тренд (RtTrend)



Системы реального времени часто требуют отображения графических трендов состояния процесса. Виджет RtTrend поддерживает отображение нескольких трендов одновременно.

Виджет Измерительный прибор (RtMeter)

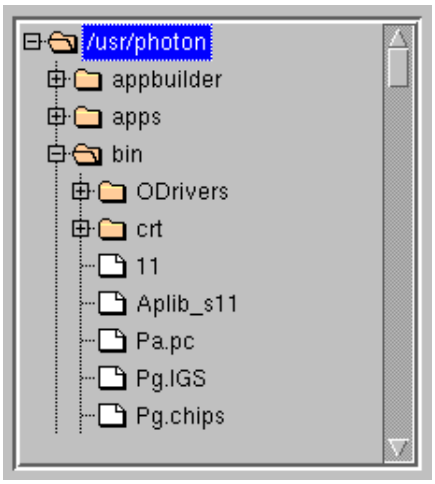


Виджет RtMeter имеет вид полукруга с рисками, отмечающими 1/3, 1/2 и 2/3 длины дуги. Стрелка может перемещаться с помощью мыши или клавиатуры или программно. Однократное нажатие кнопки мыши перемещает стрелку в текущую позицию курсора; при нажатии и последующем перемещении мыши ("drag") стрелка следует за курсором.

Диалог выбора шрифта (PtFontSel)

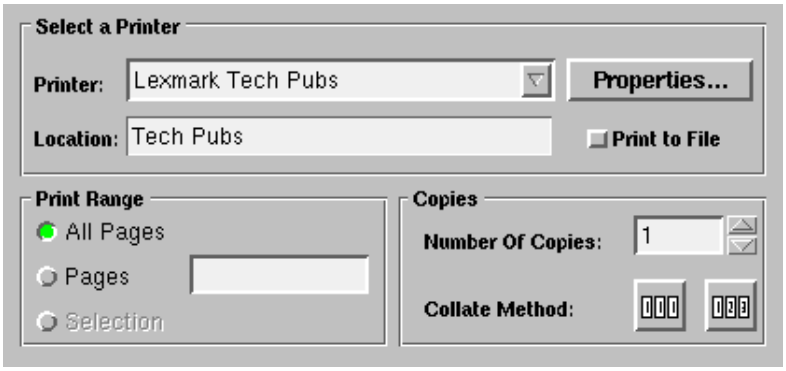
Этот виджет читает стандартные файлы конфигурации шрифтов и показывает список доступных шрифтов. Он позволяет выбрать шрифт и стиль (жирный, курсив т.д.) и также указать необходимость использования технологии сглаживания (anti-alias).

Виджет Выбор файла (PtFileSel)



Виджет PtFileSel позволяет отображать древовидную иерархию файлов, каталогов или произвольных элементов. С помощью этого виджета пользователь может просматривать структуру файловой системы и выбирать требуемый файл или каталог.

Диалог настройки печати (PtPrintSel)



Виджет PtPrintSel позволяет пользователю выбрать принтер и произвести необходимую настройку параметров печати. Пользователь может задать диапазон страниц для вывода на

печать и количество копий.

Виджет HTML (PtHtml)

Использование данного виджета облегчает создание собственного средства просмотра документации формата HTML. Виджет сам выполняет форматирование стандартного HTML-файла и даже автоматически загружает картинки. Он обрабатывает прокрутку, изменение размера, практически все требуемые функции.

Создание новых виджетов

Если стандартных виджетов Photon недостаточно, то вы можете легко создать свои собственные новые виджеты! В состав среды разработки Photon входит полная документация и примеры исходного кода для создания собственных виджетов. Вы можете создавать подклассы существующих виджетов, чтобы обеспечить наследование их функциональности, или создать собственное дерево виджетов.

Резюме

Photon олицетворяет новый подход к созданию графического пользовательского интерфейса с использованием микроядра и "команды" взаимодействующих процессов, а не монолитный подход, характерный для других оконных систем. В результате Photon демонстрирует уникальные характеристики:

- Низкие требования к объему памяти позволяют Photon обеспечивать высокий уровень функциональности оконной оболочки в условиях, где ранее было возможно только использование графической библиотеки.
- Photon обладает очень гибкой, наращиваемой архитектурой, которая позволяет разработчикам расширять возможности графического интерфейса в необходимом для своего приложения направлении.

Благодаря гибким возможностям кросс-платформенной связи, приложения Photon могут быть использованы практически в любой настольной среде.