HIGH TECH

QNX/UNIX

Анатомия параллелизма

Олег Цилюрик, Егор Горошко

Приложение
Организация обмена сообщениями

Владимир Зайцев

СИМВОЛ®

Санкт-Петербург — Москва
2006

Серия «High tech»

Олег Цилюрик, Егор Горошко

QNX/UNIX: анатомия параллелизма

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Редактор	*А. Петухов*
Художник	*В. Гренда*
Корректор	*О. Макарова*
Верстка	*О. Макарова*

Цилюрик О., Горошко Е.
QNX/UNIX: анатомия параллелизма. – СПб.: Символ-Плюс, 2006. – 288 с., ил.
ISBN 5-93286-088-X

Книга адресована программистам, работающим в самых разнообразных ОС UNIX. Авторы предлагают шире взглянуть на возможности параллельной организации вычислительного процесса в традиционном программировании. Особый акцент делается на потоках (threads), а именно на тех возможностях и сложностях, которые были привнесены в технику параллельных вычислений этой относительно новой парадигмой программирования. На примерах реальных кодов показываются приемы и преимущества параллельной организации вычислительного процесса. Некоторые из результатов испытаний тестовых примеров будут большим сюрпризом даже для самых бывалых программистов. Тем не менее излагаемые техники вполне доступны и начинающим программистам: для изучения материала требуется базовое знание языка программирования C/C++ и некоторое понимание «устройства» современных многозадачных ОС UNIX.

В качестве «испытательной площадки» для тестовых фрагментов выбрана ОСРВ QNX, что позволило с единой точки зрения взглянуть как на специфические механизмы микроядерной архитектуры QNX, так и на универсальные механизмы POSIX. В этом качестве книга может быть интересна и тем, кто не использует (и не планирует никогда использовать) ОС QNX: программистам в Linux, FreeBSD, NetBSD, Solaris и других традиционных ОС UNIX.

ISBN 5-93286-088-X

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.

Подписано в печать 28.11.2005. Формат 70x100 1/16 . Печать офсетная.
Объем 18 печ. л. Тираж 2000 экз. Заказ N
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Предисловие

Зачем написана эта книга и кому она предназначена? Различные аспекты построения программных приложений для операционной системы реального времени QNX, родственные тем, которые мы обсуждаем в данном издании, весьма обстоятельно описаны в литературе. Это и основополагающие труды Э. Дейкстры [10] и других авторов, и общая литература по POSIX (Portable Operating System Interface) и ОС UNIX [2, 3, 5–7]. Другие, сугубо специфические аспекты для ОС QNX, такие как обмен сообщениями микроядра, построение менеджеров ресурсов, пулы потоков и еще ряд других приятных вещей, прекрасно описаны в книге Р. Кертена [1].

Однако все эти источники имеют ряд недостатков:

- Техническая документация по QNX, тщательно описывающая API и детали реализации, оставляет в стороне (возможно, как относительно известные) общие вопросы построения параллельных приложений и их взаимодействия.
- Несмотря на наличие множества кратких примеров кода по использованию *отдельных* вызовов API, в технической документации явно недостаточно примеров их применения в логической последовательности и организации их совместного взаимодействия.
- Издания по UNIX, посвященные общим вопросам, напротив, изобилуют образцами кода, но в силу объективных причин, связанных с длительностью издательского процесса, не отображают новые механизмы, появившиеся в стандартах начиная с конца 90-х годов.
- И наконец, подавляющее большинство переводных книг по программированию для UNIX рассчитано на начальный уровень. Счастливое исключение – книги У. Стивенса, которые мы с удовольствием используем сами и рекомендуем читателям.

В итоге возник замысел написать книгу, в которой будет подведен итог некоторого периода нашего собственного использования ОС QNX и которая будет насыщена примерами в виде законченных проектов или отдельных фрагментов кода. Для удобства читателей `*.tgz`-архив описываемых в книге приложений размещен по адресу *http://www.symbol.ru/library/qnx-unix/pthread.tgz*, что позволит полноценно работать с текстом.

По большей части проекты трансформировались из реальных задач, из которых исключалась вся специфика конкретного заказа, только затуманивающая смысл примера. Многие примеры программного ко-

да и приложения подготовлены так, что несут двойную нагрузку: они построены как тесты тех или иных механизмов ОС и, иллюстрируя рассматриваемые аспекты, одновременно позволяют получить численные характеристики тестируемых параметров. Этим книга существенно отличается от технической документации и описательных статей по QNX, которые не слишком богаты численными показателями, подтверждающими то, о чем в них говорится.

Первоначально предполагалось создать достаточно компактный текст, систематизирующий механизмы, итак хорошо известные и понятные... Но по ходу работы объем материала, который необходимо было хотя бы затронуть, начал разрастаться как снежный ком: уточнение незначительного вопроса порождало два новых и часто гораздо более существенных, чем первоначальный; начали «вылезать» несоответствия документации и результатов тестирования; слабо связанные, на первый взгляд, механизмы (например, примитивы синхронизации POSIX и сигналы UNIX) при перекрестном взаимодействии порождают такие эффекты, которые просто нельзя оставить без внимания.

Для того чтобы хоть как-то бороться с лавинным нарастанием объема, было принято решение выделить те механизмы, программные техники и элементы API, которые наиболее слабо затронуты в литературных источниках, и рассматривать их с максимально возможной обстоятельностью (это относится, например, к базовой форме вызова `spawn()`). Напротив, те элементы, которые достаточно детально описаны и обсуждены или интуитивно понятны (например, все семейство вызовов [1], производных от `spawn()`), лишь поверхностно перечисляются (даже если позже в примерах кода мы и используем именно эти формы). В конце концов, мы не собирались пересказывать техническую документацию QNX, а хотели детально рассмотреть тонкие механизмы и их несоответствия (между собой или с изложением в документации) на работающих образцах кода.

При написании этого материала нам не были доступны никакие внутренние или специфические материалы разработчиков, кроме официальной технической документации ОС QNX, общедоступной литературы, информации, циркулирующей в Интернете в обсуждениях QNX-сообществ (в первую очередь *http://qnx.org.ru* и *http://qnxclub.net*), и контактов с коллегами-разработчиками. Многие тонкие детали приходилось восстанавливать путем тестирования и последующего толкования наблюдаемых результатов.

Как следствие, совершенно не исключено, что некоторые полученные нами результаты были неверно интерпретированы, а описания содержат определенные неточности – текущее состояние текста отображает наше *сегодняшнее* понимание наблюдаемых процессов и механизмов в системе, оно может измениться уже завтра. Мы были бы в высшей степени признательны за всякое указание на сомнительные в этом смысле места, что поможет сделать этот текст максимально адекватным и пригодным для использования.

Более того, не все наши вопросы к системе на сегодняшний день находят удовлетворительное решение – в силу ли ошибочной интерпретации процессов, наблюдаемых нами в системе, или в силу наличия в системе естественных «узлов и закрутов» (А. Ремизов), которые мы пока не в состоянии раскрутить. На таких вопросах мы сознательно акцентируем внимание, даже если пока и не можем предложить на них удовлетворительных ответов.

Но в этой технологии формирования материала по принципу «черного ящика»[1] есть и своя положительная оборотная сторона: описанные и тестированные фрагменты кода отображают состояние и функционирование механизмов ОС «как они есть», а не «как они должны быть». Это вдвойне актуально, учитывая, что разработчик QNX, фирма QSSL, не раз в своей технической документации описывала компоненты как реально существующие, в то время как их только предполагалось реализовать в последующих версиях (маршрутизация QNET «над» IP, механизмы shared memory и др.), или описывала механизмы в таком туманном изложении (дисциплины балансировки нагрузки QoS, HAM), что трудно понять, в какой степени они уже пригодны к практическому применению, а в какой – являются лишь экспериментальными наработками на будущее.

Такая неполнота информации недопустима при разработке целевых систем повышенной надежности для критических областей применения. Хочется надеяться, что наше изложение органично дополнит техническую документацию QNX, что позволит приблизить QNX к практическому использованию. Также надеемся, что эта книга окажется полезной для программистов, работающих в сфере реальных разработок.

Примечание

Предполагается, что читатель уже достаточно обстоятельно знаком с общей техникой программирования в C/C++[2] и имеет некоторый опыт в описываемой области по другим ОС (Linux, MS-DOS, MS Windows и др.). На основах программной техники мы останавливаться не будем, многие образцы кода, использующие общеизвестные и часто употребляемые приемы, будут даваться без комментариев. Более того, текст на протяжении книги очень «неровный» по глубине изложения. Например, текст приложения, предложенный В. Зайцевым, требует существенных предварительных знаний, а еще лучше – опыта разработки в QNX. Однако при первом чтении

[1] По тому же принципу писались книги, ставшие самыми информативными источниками в мировой практике, например описания Даниэля Нортона по MS-DOS или Джеффри Рихтера по Win32.

[2] Все технические описания QNX API сформированы в ориентации на классический C. Напротив, все используемые в тексте примеры кода излагаются в синтаксисе C++, а прилагаемые к тексту приложения транслированы в C++. Это обусловлено рядом аргументов, которые обсуждать не будем, но отметим, что такое различие подходов в любом случае расширяет информационную базу относительно использования QNX API.

такие «усложненные» фрагменты могут быть опущены без ущерба для понимания основного материала.

И наконец, последнее (по порядку, но не по значимости): большая часть излагаемого материала и образцов программного кода базируется на POSIX-стандартизованных механизмах (там, где обратное не оговорено особо) и поэтому может быть отнесена не только к ОС QNX, но и расширена на другие UNIX-подобные системы. Мы же в подавляющем большинстве примеров кода для программной проверки своих положений в качестве эталонной платформы используем ОС QNX как одну из доступных систем UNIX.

На основе ревизии последних лет можно сказать, что в QNX механизмы POSIX реализованы наиболее полно и последовательно относительно других систем, однако и в среде других ОС проводится активная работа по приведению их API в точное соответствие с POSIX. Один из примеров подобной эволюции в ОС Linux мы рассматриваем в тексте; в среде ОС NetBSD заявлено о приведении к соответствию с расширениями POSIX реального времени механизмов потоков и синхронизации. Большая часть кода, приводимого в тексте, может быть перенесена (иногда с незначительными изменениями) в другие системы: Linux, FreeBSD, NetBSD и проч. Отличия же в деталях функционировании рассматриваемых механизмов в разных системах, наоборот, только помогают прояснить детальную картину происходящего. Именно поэтому эта двойственность и была вынесена в заголовок книги: «QNX/UNIX».

Теперь несколько слов о многочисленных примерах программного кода в тексте. Так уж получилось, что по ходу работы над текстом мы постепенно стали нацеливать программные примеры под тестирование возможностей и эффективности иллюстрируемых программных механизмов. Однако такая ориентация, попутно предоставляющая разработчику количественные ориентиры по ОС, не должна затуманивать главное предназначение примеров кода: в них мы стараемся наиболее широко манипулировать разнообразными средствами API, с тем чтобы фрагменты этого кода могли быть непосредственно заимствованы читателями для своих будущих проектов и далее развивались там. Тем не менее в некоторых случаях мы показываем в коде, «как это можно сделать» (когда нужно иллюстрировать специфический механизм), но это вовсе не значит, что «так нужно делать» из соображений производительности, переносимости и т. д. Программный код всех примеров и все необходимое для их сборки, исполнения и проверки находятся в составе файлов архива, доступного по адресу *http://www.symbol.ru/library/qnx-unix/pthread.tgz*.

Чего нет в этой книге...

В этой книге нет множества приятных и полезных вещей: кулинарных рецептов, эротических сцен, кроссвордов... Но здесь мы хотим перечис-

лить те тематические разделы, которые имеют прямое отношение к вопросам параллельного программирования и которые должны были бы войти в книгу, но в силу определенных обстоятельств в нее не вошли:

- Общие UNIX-механизмы IPC (Inter Process Communication). Из всех механизмов, традиционно относимых к IPC, мы детально затрагиваем только один – сигналы. Другие, крайне интересные в применениях и полноценно представленные в API QNX, такие как неименованные (pipe) и именованные (FIFO) каналы, очереди сообщений POSIX (`mq_*`), блокирование записей и файлов (`fcntl()`) и ряд других механизмов, полностью и сознательно обойдены вниманием. Это связано с тем, что: а) в программирование этих механизмов мало что привносит именно «потоковая» (thread) ориентация, положенная во главу угла нашего рассмотрения; б) эти механизмы настолько исчерпывающе описаны У. Стивенсом [2], что добавить что-либо трудно; в) нам крайне не хотелось раздувать объем текста сверх некоторой разумной меры без крайней на то необходимости.
- Совместно используемая (разделяемая) память (shared memory). Это также один из механизмов, традиционно относимый к подмножеству IPC, но мы его выделяем особо. Это именно тот механизм, который **должен** быть описан применительно к QNX самым тщательным образом, но… Самые поверхностные эксперименты наводят на мысль, что именно в QNX реализации механизмов разделяемой памяти выполнены достаточно «рудиментарно»: ряд возможностей, рассматриваемых POSIX как стандартные, не реализован или реализован в ограниченной мере. Поэтому механизмы разделяемой памяти в QNX требуют отдельного пристального изучения и тестирования. Возможно, это должно быть сделано в отдельной публикации, что мы и планируем восполнить в ближайшем будущем.
- Таймеры в системе. Таймерные механизмы в QNX развиты в полной мере, что неудивительно для ОС реального времени, ориентированной во многом на «встраиваемые» (embedded) применения. Однако таймеры а) имеют все же косвенное отношение к вопросам параллелизма и синхронизации и б) блестяще и полно описаны Р. Кертеном [1].

Вообще, при отборе материала для книги мы старались максимально придерживаться следующего алгоритма: чем шире некоторый предмет освещен в литературе (объект или механизм ОС, приемы его использования и тому подобное), по крайней мере, в известной нам литературе, тем меньше внимания мы уделяли ему в своем тексте.

Благодарности

Предварительный вариант книги был вынесен на обсуждение широкой QNX-общественности (да и UNIX/Linux) на форуме *http://qnxclub.net*. Было высказано столько замечаний, пожеланий и рекомендаций, что окончательный текст, в котором все они были учтены, стал радикально

отличаться от исходной редакции. Невозможно перечислить всех членов интернет-сообщества (в первую очередь, конечно, *http://qnx.org.ru* и *http://qnxclub.net*), кто внес свой вклад в это издание – мы благодарны всем без исключения. Но особую признательность мы выражаем:

- Владимиру Зайцеву из г. Харькова, который не только предоставил свой авторский материал, составивший отдельное, самоценное приложение, дополнившее книгу, но и обстоятельно вычитал весь остальной текст, а также внес ряд весьма ценных уточнений.
- Евгению Видревичу из г. Монреаля, который вычитал весь текст с позиции профессионального разработчика и указал на ряд сомнительных мест, а вместо некоторых наших путаных и не совсем внятных формулировок предложил свои – ясные и прозрачные.
- Евгению Тарнавскому из г. Харькова, который высказал ряд очень точных рекомендаций и замечаний, нашедших свое отражение в окончательном варианте текста.

Типографские соглашения

В тексте содержится множество ссылок на программные конструкции: фрагменты кода, имена функций API, символические константы и многое другое, которые при их использовании должны в неизменном виде (именно в таком написании) «перекочевывать» в программный код. Такие фрагменты, конструкции и лексемы выделены моноширинным шрифтом, например `pthread_create()`.

Фрагменты текста, цитируемые из указанных источников, выделены *курсивом*. Таких мест очень немного: прямое цитирование допускалось нами только в отношении крайне принципиальных утверждений.

В отличие от коротких (в две-три строки) фрагментов кода, листинги программ, приводимых и обсуждаемых в тексте, предваряются отчетливо выделенным заголовком. Это указывает на то, что данную программу как законченную программную единицу можно найти в архиве по адресу *http://www.symbol.ru/library/qnx-unix/pthread.tgz*. Помимо крупных законченных проектов там же можно найти и отдельные фрагменты кода, обсуждаемые в тексте. Для удобства поиска названия программных файлов, содержащихся в архиве, приводятся в тексте книги перед соответствующим кодом в скобках, например (*файл s2.cc*).[1]

---

[1] В книге в примерах кода мы часто используем русскоязычные символьные константы для вывода сообщений, например `"Получен сигнал SIGINT"`, что способствует большей доходчивости обсуждаемого кода. Однако в файлах работающих приложений, представленных в архиве, вы увидите только англоязычные эквиваленты выводимых сообщений, поскольку работающие примеры кода являются консольными приложениями, а текстовая консоль QNX не русифицируема в принципе и графические псевдотерминалы (pterm, xterm) имеют определенные сложности.

1

Введение

Параллелизм

Феномен параллелизма при выполнении принципиально последовательного по своей природе компьютерного кода возникает даже раньше, чем он начинает отчетливо требоваться для многозадачных и многопользовательских операционных систем:

- Код обработчиков аппаратных прерываний, являющихся принципиально асинхронными, в самых последовательных ОС выполняется параллельно прерываемому ими коду.
- Для работы многих системных служб необходимо, чтобы они выполнялись параллельно с выполнением пользовательской задачи.

Примечание

Например, в принципиально однозадачной операционной системе MS-DOS исторически первой службой, требующей параллельного выполнения, была подсистема спулинга печати. Но добавлять ее в систему пришлось «по живому», поскольку основная структура системы уже сложилась и стабилизировалась (к версии 2.x), а механизмы параллелизма в этой структуре были изначально отвергнуты на корню. И с этого времени начинается затянувшаяся на многие годы история развития уродливой надстройки над MS-DOS – технологии создания TSR-приложений (terminate and stay resident), программного мультиплексора INT 2F и других.

Новое «пришествие» механизмов параллельного выполнения (собственно, уже хорошо проработанных к этому времени в отрасли мэйнфреймов) начинается с появлением многозадачных ОС, разделяющих во времени выполнение нескольких задач. Для формализации (и стандартизации поведения) развивающихся параллельно программных ветвей создаются абстракции процессов, а позже и потоков. Простейший случай параллелизма – когда N (N>1) задач разделяют между собой ресурсы: время единого процессора, общий объем физической оперативной памяти…

Но многозадачное разделение времени – не единственный случай практической реализации параллельных вычислений. В общем случае программа может выполняться в аппаратной архитектуре, содержащей более одного (M) процессора (SMP-системы). При этом возможны принципиально отличающиеся по поведению ситуации:

- Количество параллельных ветвей (процессов, потоков) N больше числа процессоров M, при этом некоторые вычислительные ветви находятся в блокированных состояниях, конкурируя с выполняющимися ветвями за процессорное время. (Частный случай – наиболее часто имеющее место выполнение N ветвей на одном процессоре.)
- Количество параллельных ветвей (процессов, потоков) N меньше числа процессоров M, при этом все ветви вычисления могут развиваться действительно параллельно, а блокированные состояния возникают только при необходимости синхронизации и обмена данными между параллельными ветвями.

Все механизмы параллелизма проектируются (и это находит прямое отражение в POSIX-стандартах, а еще более в текстах комментариев к стандартам) и должны использоваться так, чтобы неявно не допускались какие-либо предположения об относительных скоростях параллельных ветвей и моментах достижения ими (относительно друг друга) конкретных точек выполнения.[1] Так, в программном фрагменте:

```
void* threadfunc ( void* data ) {
  // оператор 1:
};
...
pthread_create( NULL, NULL, threadfunc, NULL );
// оператор 2:
...
```

нельзя допускать никаких априорных предположений о том, как пойдет дальнейшее выполнение после точки ветвления (точки вызова `pthread_create()`): а) будет выполняться «оператор 2» в родительском потоке; б) будет выполняться «оператор 1» в порожденном потоке; в) на различных процессорах будут действительно одновременно выполняться «оператор 1» и «оператор 2»... Программный код должен быть организован так, чтобы в любых аппаратных конфигурациях (количество процессоров, их скорости, особенности кэширования па-

[1] Это положение напрямую диктуется определением «слабосвязанных процессов», впервые сформулированным Э. Дейкстрой [10]. Заметим, что фундаментальная и стройная «картина мира», выстроенная Э. Дейкстрой и считающаяся классикой, исчерпывающе («необходимо и достаточно») описывает систему процессов равного приоритета. Расширение реальных систем атрибутом приоритета затуманивает прозрачность этой модели и делает все гораздо сложнее...

мяти процессорами и другие характеристики) результаты выполнения были полностью эквивалентны.

Благодаря наличию в составе ОС QNX сетевой подсистемы QNET, органично обеспечивающей «прозрачную» интеграцию сетевых узлов в единую многомашинную систему, возникает дополнительный источник параллелизма (а вместе с тем и дополнительных хлопот), еще более усложняющий общую картину: запросы по QNET к сервисам, работающим на одном сетевом узле, со стороны клиентских приложений, работающих на других. Например, ежедневно выполняя простейшую команду:

```
# cp /net/host/dev/ser1 ./file
```

часто ли мы задумываемся над тем, кого и в каком порядке будет вытеснять код, выполняющий копирование файлов.

Для текущей выполняющейся задачи такой удаленный запрос из сети QNET является скрытым источником параллелизма, а благодаря наследованию приоритетов даже удаленный запрос по сети может привести к немедленному вытеснению локальной задачи, выполняющейся до получения запроса.

Приведенная выше аргументация – это далеко не полный перечень причин, по которым стоит еще пристальнее и с большей заинтересованностью взглянуть на техники параллельной организации вычислительного процесса. В литературе неоднократно отмечалось (например, [11]), что даже в тех случаях, когда приложение заведомо никогда и нигде не будет использоваться на многопроцессорной платформе, более того, когда логика приложения не предполагает естественного параллелизма как «одновременности выполнения», – даже тогда расщепление крупного приложения на логические фрагменты, которые построены как параллельные участки кода, взаимодействующие в ограниченном числе точек контакта, – это путь построения «прозрачного» для написания и понятного для сопровождения программного кода. И как следствие, этот путь (иногда на первый взгляд кажущийся несколько искусственным и привнесенным) – путь построения приложений высокой надежности, свободных от ошибок, характерных для громоздких монолитных приложений, и простых в своем последующем развитии и сопровождении.

Как уже неоднократно отмечалось, параллельная техника выражения в программном коде, пусть даже принципиально последовательных процессов, сопряжена с определенными трудностями: необходимость отличного, «параллельного», взгляда на описываемые процессы и отсутствие привычки применять специфические разделы API, редко используемые в классическом «последовательном» программировании. Единожды освоив эту технику, применять ее в дальнейшем становится легко и просто. Возможно и большее число рутинных приемов ис-

пользования параллельной техники – в своей книге мы постарались «рассыпать» по тексту множество программных иллюстраций.

Наконец, есть еще одна, последняя особенность предлагаемого вашему вниманию материала: значительная часть приводимых здесь примеров и описаний относится ко всему многообразию ОС, поддерживающих POSIX-стандарт, однако акцент делается на не совсем очевидные особенности построения так называемых «приложений реального времени» [4]. В первую очередь это касается принципов синхронизации задач, совместно использующих общий ресурс. К сожалению, приемы программирования, широко распространенные при параллельном выполнении задач общего назначения, могут привести к не совсем предсказуемым результатам (по времени реакции) при построении систем реального времени. Особенности построения параллельно исполняемых систем в сферах реального времени и стали тем ключевым моментом, ориентируясь на который мы строили этот текст.

Семейства API

Общее множество вызовов API (Application Program Interface – интегральное наименование всего множества вызовов из программной среды к услугам операционной системы), реализуемое операционной системой (ОС) реального времени QNX, естественным образом разделяется на три независимых подгруппы:

- Native QNX API – это самодостаточный набор вызовов, развиваемый со времен ранних версий QNX (когда вопрос о совместимости с POSIX еще не стоял); является естественным базисом этой системы, отображающим «микроядерность» ее архитектуры, но по соображениям возможной совместимости и переносимости он является также и исключительной принадлежностью этой ОС.
- POSIX (BSD) API – это уровень API, регламентируемый постоянно расширяющейся системой стандартов группы POSIX, которым должны следовать все ОС, претендующие на принадлежность к семейству UNIX.
- System V API (POSIX) – это та часть API, которая заимствует модели, принятые в UNIX-ах, относящихся к ветви развития System V, а не к ветви BSD.

Native QNX API

Именно этот слой является базовым слоем, реализующим функциональность самой системы QNX. Два последующих слоя в значительной мере являются лишь «обертками», которые ретранслируются в вызовы native QNX API после выполнения реструктуризации или перегруппировки аргументов вызова в соответствии с синтаксисом, требуемым этим вызовом.

Совершенно естественно, что прикладное программное приложение может быть полностью прописано в этом API (как, впрочем, и в каждом другом из описываемых ниже), но это не лучший выбор (на этом акцентирует внимание и техническая документация QNX) по двум причинам: во-первых, из соображений переносимости, а во-вторых, этот слой является самым «мобильным» – разработчики QSSL могут изменить его отдельные вызовы при последующем развитии системы. Примером вызова этого слоя является, в частности, `ThreadCreate()`, применяемый для создания нового потока.

Тем не менее нужно сразу отметить, что многие возможности и модели (например, реакция на сигналы в потоках, тонкое управление поведением мьютексов и другие моменты) не могут быть реализованы в рамках POSIX-модели и выражаются только в native API QNX.

POSIX (BSD) API

Эта часть API наиболее полно соответствует API ОС UNIX, относящихся к ветви BSD (BSD, FreeBSD, NetBSD и другие).[1] Ее наименование можно было бы сузить до «BSD API», так как описанный далее набор API System V также регламентируется POSIX, но мы будем использовать именно термин «POSIX API», следуя терминологии фундаментальной книги У. Стивенса [2]. Эквивалентом названного выше для native API `ThreadCreate()` здесь будет выступать `pthread_create()`.

Именно на API этого слоя и будет строиться последующее изложение и приводимые примеры кода (параллельно с вызовами этого API мы будем для справки кое-где указывать имена комплиментарных им вызовов native API), за исключением случаев использования тех возможностей QNX, которые не имеют эквивалентов в POSIX API. Как раз все, что будет выражено в этом API далее по тексту, может быть перенесено на все UNIX-подобные операционные системы, о чем мы и говорили выше.

Примечание

Самый ранний стандарт POSIX известен как IEEE 1003.1-1988 и, как следует из его названия, относится к 1988 году (если точнее, то ему предшествовал рабочий вариант под названием IEEEIX 1986 года, когда термин POSIX еще не был «придуман»). Более поздняя редакция его развития, IEEE 1003.1-1996, наиболее широко известна как «стандарт POSIX», иногда называемый POSIX.1. Набор стандартов POSIX

[1] На сегодняшний день практически ни одна из ОС UNIX уже не может быть отнесена чисто к System V или BSD, во многом исходя именно из требования совместимости с POSIX, который требует одновременного наличия и того и другого API (хотя в каждом случае комплиментарный набор API реализуется как «обертка» к базовому). Одними из первых (к 1997–1998 гг.) ОС, поддерживающих оба набора API, стали Sun Solaris 2.6 и Digital Unix 4.0B [3].

находится в постоянном развитии и расширении и к настоящему времени включает в себя набор более чем из 30 автономных стандартов.

Для целей операционных систем реального времени возникла потребность определить отдельные механизмы особыми стандартами, на семь из которых ссылаются наиболее часто: 1003.1a, 1003.1b, 1003.1c, 1003.1d, 1003.1j, 1003.21, 1003.2h. Например:

1003.1a (OS Definition) – определяет базовые интерфейсы ОС;

1003.1b (Realtime Extensions) – описывает расширения реального времени, такие как модель сигналов реального времени, диспетчеризация по приоритетам, таймеры, синхронный и асинхронный ввод-вывод, IPC-механизмы (семафоры, разделяемая память, сообщения);

1003.1c (Threads) – определяет функции поддержки потоков, такие как управление потоками, атрибуты потоков, примитивы синхронизации (мьютексы, условные переменные, барьеры и др., но не семафоры), диспетчеризация.

System V API

Этот набор API является базовым для второй ветви[1] UNIX – System V (AT&T Unix System V). Как и оба предыдущих, этот набор API самодостаточен для реализации практически всех возможностей ОС, но использует для этого совершенно другие модели, например сетевую абстракцию TLI вместо сокетов BSD. Для области рассматриваемых нами механизмов – потоков, процессов, синхронизирующих примитивов и др. – в POSIX API и System V API почти всегда существуют функциональные аналоги, отличающиеся при этом как синтаксически, так и семантически. Например, в POSIX API семафор представлен типом `sem_t` и основными операциями с ним `sem_wait()` и `sem_post()`, а в System V API семафор описывается структурой ядра `sem`, а операции (и `wait`, и `post`) осуществляются вызовом `semop()`. Кроме того, операции производятся не над единичными семафорами, а над наборами (массивами) семафоров (в наборе может быть и один семафор). Как отсюда видно, логика использования принципиально единообразных примитивов существенно отличается.

[1] При общей истории UNIX, начинающейся с 1971 г. [7], две ветви API – BSD и System V – в их современном виде сформировались достаточно поздно: BSD к 1983 г., а System V к 1987 г. [3, 7]. Но многие IPC-механизмы System V (например, семафоры) сформировались по времени заметно раньше своих аналогов из BSD. Как отмечается в [3]: «*Информация об истории разработки и развитии функций System V IPC не слишком легко доступна <...> очереди сообщений, семафоры и разделяемая память этого типа были разработаны в конце 70-х в одном из филиалов Bell Laboratories в городе Колумбус... Эта версия называлась Columbus Unix, или CB Unix*».

Примечание

В технической документации присутствие System V API в QNX не упоминается ни одним словом, но он, как того и требует POSIX, действительно предоставляется и в виде библиотек, и в виде необходимых файлов определений (заголовочных файлов). Просто его заголовочные файлы, определяющие структуры данных и синтаксис вызовов, находятся в других относительно POSIX-интерфейсов местах. Так, например, описание семафоров POSIX API (тип `sem_t`) расположено в файле `<semaphore.h>`, а описание семафоров System V API – в файле `<sys/sem.h>` (аналогично относительно всех конструкций, моделируемых этим API).

С позиции программиста System V API присутствует в QNX главным образом для переносимости программных проектов, ранее созданных с использованием этого API, например первоначально созданных для других ОС UNIX (Sun Solaris, HP UNIX и др.). В данной книге это семейство API рассматриваться не будет.

2

Процессы и потоки

При внимательном чтении технической документации [8] и литературы по ОС QNX [1] отчетливо бросается в глаза, что тонкие детали создания и функционирования процессов и потоков описаны крайне поверхностно и на весьма некачественном уровне. Возможно, это связано с тем, что общие POSIX-механизмы уже изучены и многократно описаны на образцах кода в общей литературе по UNIX. Однако большинство литературных источников написано в «допотоковую» эпоху, когда основной исполняемой единицей в системе являлся процесс.

Детальное рассмотрение особенностей именно QNX[1] (версии 6.X после приведения ее в соответствие с POSIX, в отличие от предыдущей 4.25) лишний раз подчеркивает, что:

- Процесс является только «мертвой» статической оболочкой, хранящей учетные данные и обеспечивающей окружение динамического исполнения… Чего? Конечно же, потока, даже если это единственный (главный) исполняемый поток приложения (процесса), как это принято в терминологии, не имеющий отношения к потоковым понятиям.
- Любые взаимодействия, синхронизация, диспетчеризация и другие механизмы имеют смысл только применительно к потокам, даже если это потоки, локализованные в рамках различных процессов. Вот здесь и возникает термин, ставший уже стереотипным: «IPC – средства взаимодействия **процессов**». Для однопотоковых приложений этот терминологический нюанс не вносит ровно никакого различия, но при переходе к многопотоковым приложениям мы должны рассуждать в терминах именно взаимодействующих потоков, локализованных внутри процессов (одного или различных).
- В системах с аппаратной трансляцией адресов памяти (MMU – Memory Menegment Unit) процесс создает для своих потоков дополнительные «границы существования» – защищенное адресное про-

[1] [4]: глава Д. Алексеева «Получение системной информации».

странство. Большинство сложностей, описываемых в литературе в связи с использованием IPC, обусловлено необходимостью взаимодействующих потоков преодолевать адресные барьеры, устанавливаемые процессами для каждого из них. (Что касается MMU, то в данной книге предполагается исключительно x86-архитектура, хотя количество аппаратных платформ, на которых работает ОС QNX, на сегодняшний день уже перевалило за десяток.)

Примечание

Модель потоков QNX в значительной степени напоминает то, что происходит с процессами в MS-DOS или с задачами (task) в существенно более поздней ОС реального времени VxWorks: исполнимые единицы разделяют единое адресное пространство без каких-либо ограничений на использование всего адресного пространства. В рамках подобной модели в QNX можно реализовать и сколь угодно сложный комплекс, трансформировав в потоки отдельные процессы, составляющие этот комплекс, с тем только различием, что в QNX все элементы собственно операционной системы продолжают работать в изолированном адресном пространстве и не могут быть никоим образом включены (и тем самым повреждены) в пространство приложения.

И в технической документации QNX, и в книге Р. Кертена [1] много страниц уделено описанию логики процессов, потоков, синхронизации и многим другим вещам в терминах аллегорических аналогий: коллективное пользование ванной комнатой, кухней... Если согласиться, что такие аллегории более доходчивы для качественного описания картины происходящего (что, похоже, так и есть), то для иерархии «операционная система – процесс – поток» можно найти существенно более близкую аллегорию: «аквариумное хозяйство». Действительно:

- В некотором общем помещении, где имеются все средства жизнеобеспечения – освещение, аэрация, терморегуляция, кормление (операционная система), – размещаются аквариумы (процессы), внутри которых (в одних больше, в других совсем немного) живут активные сущности (растения, рыбы, улитки). Помимо всех прочих «удобств» в помещении время от времени появляется еще одна сущность – «хозяин». Он является внешней по отношению к системе силой, которая асинхронно предпринимает некоторые действия (кормление, пересадка животных), нарушающие естественное «синхронное» течение событий (это служба системного времени операционной системы, которая извне навязывает потокам диспетчеризацию).
- Аквариумы (процессы) являются не только контейнерами, заключающими в себе активные сущности (потоки). Они также ограничивают ареал существования (защищенное адресное пространство) для их обитателей: любое нарушение границ обитания в силу каких-либо форс-мажорных обстоятельств, безусловно, означает гибель нарушителя (ошибка нарушения защиты памяти в потоке).

- Обитатели аквариумов (потоки) легко и непринужденно взаимодействуют между собой (сталкиваются при движении или, напротив, уступают друг другу место) в пределах контейнера (процесса). Однако при этом они не могут взаимодействовать с обитателями других контейнеров (процессов); более того, они даже ничего не знают об их существовании. Если обитатель требует вмешательства, например перемещения его в другой контейнер, то он может лишь способствовать этому, взывая своим поведением (при помощи особых знаков) к инстанции более высокого уровня иерархии, в отличие от контейнера некоторой «общесистемной субстанции», взывающего к хозяину (операционной системе) о вмешательстве (диспетчеризации).
- Все жизненно необходимые ресурсы (кислород, корм, свет) поступают непосредственно к контейнеру как единице распределения (операционная система выделяет ресурсы процессу в целом). Обитатели контейнера (потоки) конкурируют за распределение общих ресурсов контейнера на основании своих характеристик (приоритетов) и некоторой логики (дисциплины) распределения относительно «личностных» характеристик: размера животного, быстроты реакции и движения и т. д.

Такая ассоциативная аналогия, возможно, позволит отчетливее ощутить, что процесс и поток относятся к различным уровням иерархии понятий ОС. Это различие смазывается тем обстоятельством, что в любой ОС (с поддержкой модели потоков или без нее) всякий процесс всегда наблюдается в неразделимом единстве хотя бы с одним (главным) потоком и нет возможности наблюдать и анализировать поведение «процесса без потока».

Отсюда и происходят попытки объединения механизмов создания и манипулирования процессами и потоками «под одной крышей» (единым механизмом). Например, в ОС Linux создание и процесса (`fork()`), и потока (`pthrad_create()`) свели к единому системному вызову `_clone()`, что явилось причиной некоторой иллюзорной эйфории, связанной с непонятной, мифической «дополнительной гибкостью».

Усилия последующих лет были направлены как раз на разделение этих механизмов, ликвидацию этой «гибкости» и восстановление POSIX-модели. Отсюда же вытекают и разработки последних лет в области новых «экзотических» ОС, направленные на сближение модели процесса и потока, и попытки создания некой «гибридной» субстанции, объединяющей атрибуты процесса и потока, если того захочет программист (на момент создания). По нашему мнению, идея «гибридизации» достаточно сомнительна и согласно нашей аналогии направлена на создание чего-то, в головной своей части напоминающего аквариум, а в задней – рыбу. Получается даже страшнее, чем русалка...

Отмеченный выше дуализм абстракций процессов и потоков (а в некоторых ОС и их полная тождественность) приводит к тому, что крайне

сложно описывать одно из этих понятий, не прибегая к упоминанию атрибутов другого. В итоге, с какой бы из двух абстракций ни начать рассмотрение, нам придется, забегая вперед, ссылаться на атрибутику другой, дуальной ей. В описании процессов нам не обойтись без понятия приоритета (являющегося атрибутикой потока), а в описании потоков мы не сможем не упомянуть глобальные (относительно потока) объекты, являющиеся принадлежностью процесса, например файловые дескрипторы, сокеты и многое другое.

По этой причине наше последующее изложение при любом порядке его «развертывания» обречено на некоторую «рекурсивность». Итак, следуя сложившейся традиции, начнем с рассмотрения процессов.

Процессы

Создание параллельных процессов настолько полно описано в литературе по UNIX, что здесь мы приведем лишь минимально необходимый беглый обзор, останавливаясь только на отличительных особенностях ОС QNX.

Всякое рассмотрение предполагает наличие системы понятий. Интуитивно ясное понятие процесса не так просто поддается формальному определению. Процитируем (во многом качественное) определение, которое дает Робачевский [3]:

> *Обычно программой называют совокупность файлов, будь то набор исходных текстов[1], объектных файлов или собственно выполняемый файл. Для того чтобы программа могла быть запущена на выполнение, операционная система сначала должна создать окружение или среду выполнения задачи, куда относятся ресурсы памяти, возможность доступа к устройствам ввода/вывода и различным системным ресурсам, включая услуги ядра.*

Процесс всегда содержит хотя бы один поток, поскольку мы говорим об исполняемом, развивающемся во времени коде. Для процессов, исходный код которых подготовлен на языке C/C++, **главным потоком** процесса является поток, в котором исполняется функция, текстуально описанная под именем `main()`. Код и данные процесса размещаются в оперативной памяти в **адресном пространстве** процесса. Если операционная система и реализующая платформа (наше рассмотрение ограничено только реализацией x86) поддерживают MMU и виртуализацию адресного пространства на физическую память, то каждый процесс

[1] Здесь Робачевский мимоходом расширяет понятие процесса и на программу, представленную, например, текстом для интерпретатора shell, или языков Perl, Tcl/Tk, или других интерпретаторов. В контексте нашего обсуждения в случаях выполнения таких «программ» «процессом» будет процесс, интерпретирующий текст скрипта, и именно к нему в полной мере относятся все детали нашего рассмотрения относительно процессов.

имеет собственное изолированное и уникальное адресное пространство и у него нет возможности непосредственно обратиться в адресное пространство другого процесса.

Любой процесс может содержать произвольное количество потоков, но не менее одного и не более 32 767 (для QNX версии 6.2). Совокупность данных, необходимых для выполнения любого из потоков процесса, а также контекст текущего выполняемого потока называются **контекстом процесса**.

Согласно ранним «каноническим» спецификациям UNIX [3] ОС должна поддерживать не менее 4095 отдельных процессов (точнее 4096, из которых 0-вой представляет собой процесс, загружающий ОС и, возможно, реализующий в дальнейшем функции ядра). Во всей документации ОС QNX нам не удалось найти предельное значение этого параметра. Но если из этого делается «тайна мадридского двора», то наша задача – найти это значение:

```
int main( int argc, char* argv[] ) {
   unsigned long n = 1;
   pid_t pid;
   while( ( pid = fork() ) >= 0 ) {
      n++;
      if( pid > 0 ) {
         waitpid( pid, NULL, WEXITED );
         exit( EXIT_SUCCESS );
      };
   };
   if( pid == -1 ) {
      cout << "exit with process number: "
           << n << " - " << flush;
      perror( NULL );
   };
};
```

Этот достаточно непривычный по внешнему виду код дает нам следующий результат:

```
# pn
exit with process number: 1743 - Not enough memory
```

Системному сообщению о недостатке памяти достаточно трудно верить: чуть меньше 4 Кбайт программного кода в своих 1743 «реинкарнациях» требуют не более 6,6 Мбайт для своего размещения при свободных более 230 Мбайт в системе, в которой мы испытывали это приложение. Оставим это на совести создателей ОС QNX.

В продолжение нашей основной темы любопытно рассмотреть результаты вывода команды `pidin`, а именно последнюю ее строку с информацией о последнем запущенном в системе процессе:

- до запуска обсуждаемого приложения:

```
47366186    1 /photon/bin/phcalc  10r REPLY       241691
```

- и после его завершения:

```
54652947    1 bin/pidin           10r REPLY       1
```

Легко видеть, что разница PID, равная 54652947 – 47366186 = 7286761, никак не является числом активированных на этом временном промежутке процессов, которое равно 1743. Поэтому к численным значениям PID нужно относиться с заметной осторожностью: это не просто инкрементированное значение числа запущенных процессов, схема формирования PID заметно сложнее.

В любом случае мы можем принять, что в ОС QNX Neutrino 6.2.1, как и в других «канонических» UNIX, количество процессов (если, конечно, эта ОС не дает нам более вразумительных оценок) ограничено цифрой 4095. Видно, что общее количество независимых потоков исполнения в системе может достигать совершенно ошеломляющей цифры. Но как бы много потоков мы ни создавали, им все равно придется конкурировать за доступ к самому главному ресурсу – процессору. В настоящее время реализованные в QNX дисциплины диспетчеризации работают над суммарным полем всех потоков в системе (рис. 2.1): если в системе выполняется N процессов и i-й процесс реализует M_i потоков, то в очередях диспетчеризации одновременно задействовано $\sum_{i=1}^{N} M_i$ управляемых объектов (потоков).

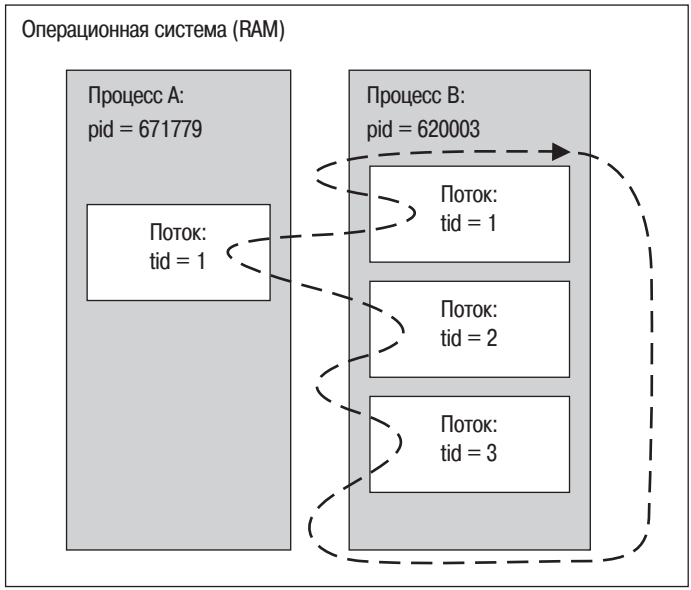

Рис. 2.1. *Диспетчеризация процессов*

На рис. 2.1 изображены два процесса, выполняющиеся под управлением системы. Каждый процесс создал внутри себя различное количество потоков равного приоритета. Обратите внимание, что фактическая диспетчеризация производится не между процессами, а между потоками процессов, даже если иногда для простоты говорят «диспетчеризация процессов». Потоки объединены в циклическую очередь диспетчеризации, и пунктирная линия показывает порядок, в котором (в направлении стрелки) они будут поочередно получать квант времени.

Если ни один из потоков не будет выполнять блокирующих операций (`read()`, `delay()`, `accept()`, `MsgSend()` и множество других), что реально встречается крайне редко, то показанный порядок «следования» потоков при диспетчеризации будет сохраняться неограниченно долго. Как только поток выполнит блокирующий вызов, он будет удален из очереди готовых к выполнению потоков, а после завершения вызова возвращен в очередь, причем (что характерно!) в голову очереди. После этого топология «петли» (порядок чередования), показанной на рисунке пунктиром, может произвольным образом измениться.

Из рисунка хорошо видно, что при диспетчеризации «в рамках системы» (об этом мы будем говорить позже) два запущенных процесса будут выполняться в неравных условиях: на каждый полный цикл диспетчеризации программный код, выполняющийся в рамках процесса A, будет получать 1 квант времени, а код в процессе B – 3 кванта.

Примечание

Стандарт POSIX, определяя названную стратегию диспетчеризации константой `PTHREAD_SCOPE_SYSTEM`, предусматривает и другую стратегию, обозначаемую константой `PTHREAD_SCOPE_PROCESS`, когда потоки конкурируют за процессорный ресурс в пределах процесса, к которому они принадлежат (в Sun Solaris первой стратегии соответствуют «bound thread», а второй – «unbound thread»). Реализация стратегии `PTHREAD_SCOPE_PROCESS` связана с серьезными трудностями. Насколько нам известно, в настоящее время из числа широко распространенных ОС она реализована только в Sun Solaris. В QNX для совместимости с POSIX даже присутствуют системные вызовы относительно стратегии диспетчеризации:

```
int pthread_attr_setscope( pthread_attr_t* attr, int scope );
int pthread_attr_getscope( const pthread_attr_t* attr, int* scope );
```

но в качестве параметра `scope` они допускают... только значение `PTHREAD_SCOPE_SYSTEM` и на поведение потоков никакого влияния не оказывают.

PID (Process ID) – идентификатор процесса, присваиваемый процессу при его создании, например вызовом `fork()`. PID позволяет системе однозначно идентифицировать каждый процесс. При создании нового процесса ему присваивается первый свободный (то есть не ассоциированный ни с каким процессом) идентификатор. Присвоение происходит по возрастающей: идентификатор нового процесса больше иденти-

фикатора процесса, созданного перед ним. Когда последовательность идентификаторов достигает максимального значения (4095), следующий процесс получает минимальный свободный (за счет завершившихся процессов) PID, и весь цикл повторяется снова. Значения PID нумеруются, начиная с 0. Процесс, загружавший ОС, является родительским для всех процессов в системе? и его PID = 0.

Из других важных атрибутов процесса отметим[1]:

- PPID (Parent Process ID) – PID процесса, породившего данный процесс. Таким образом, все процессы в системе включены в единую древовидную иерархию.
- TTY – терминальная линия: терминал или псевдотерминал, ассоциированный с процессом. Если процесс становится процессом-демоном, то он отсоединяется от своей терминальной линии и не имеет ассоциированной терминальной линии. (Запуск процесса как фонового – знак «&» в конце командной строки – не является достаточным основанием для отсоединения процесса от терминальной линии.)
- RID и EUID – реальный и эффективный идентификаторы пользователя. Эффективный идентификатор служит для определения прав доступа процесса к системным ресурсам (в первую очередь к файловым системам). Обычно RID и EUID совпадают, но установка флага SUID для исполняемого файла процесса позволяет расширить полномочия процесса.
- RGID и EGID – реальный и эффективный идентификаторы группы пользователей. Как и в случае идентификаторов пользователя, EGID не совпадает с RGID, если установлен флаг SGID для исполняемого файла процесса.

Часто в качестве атрибутов процесса называют и приоритет выполнения. Однако приоритет является атрибутом не процесса (процесс – это статическая субстанция, контейнер), а потока, но если поток единственный (главный, порожденный функцией `main()`), его приоритет и есть то, что понимается под «приоритетом процесса».

Создание нового процесса

Созданию процессов (имеется в виду создание процесса из программного кода) посвящено столько описаний [1–9], что детальное рассмотрение этого вопроса было бы лишь пересказом. Поэтому мы ограничимся только беглым перечислением этих возможностей, тем более что в ходе обсуждения нас главным образом интересуют не сами процессы, а потоки, заключенные в адресных пространствах процессов.

---

[1] Здесь используется терминология [7]; терминология и аббревиатуры для различных клонов UNIX несколько различаются между собой в описывающих их литературных источниках.

Использование командного интерпретатора

Самый простой способ – запустить из программного кода дочернюю копию командного интерпретатора, которому затем передать команду запуска процесса. Для этого используется вызов:

```
int system( const char * command );
```

где `command` – текстовая строка, содержащая команду, которую предполагается выполнить ровно в том виде, в котором мы вводим ее командному интерпретатору с консоли.

Примечание

Функция имеет еще одну специфическую форму вызова, когда в качестве `command` задается `NULL`. По коду возврата это позволяет выяснить, присутствует ли (и доступен ли) командный интерпретатор в системе (возвращается 0, если интерпретатор доступен).

На время выполнения вызова `system()` вызывающий процесс приостанавливается. После завершения порожденного процесса функция возвращает код завершения вновь созданной копии интерпретатора (или –1, если сам интерпретатор не может быть выполнен), то есть младшие 8 бит возвращаемого значения содержат код завершения выполняемого процесса. Возврат вызова `system()` может анализироваться макросом `WEXITSTATUS()`, определенным в файле `<sys/wait.h>`. Например:

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

int main( void ) {
   int rc = system( "ls" );
   if( rc == -1 ) cout << "shell could not be run" << endl;
   else
      cout << "result of running command is "
           << WEXITSTATUS( rc ) << endl;
    return EXIT_SUCCESS;
};
```

Примечание

Эта функция использует вызов `spawnlp()` для загрузки новой копии командного интерпретатора, то есть «внутреннее устройство» должно быть в общем виде вам понятно. Особенностью QNX-реализации является то, что `spawnlp()` всегда использует вызов `/bin/sh`, независимо от конкретного вида интерпретатора, устанавливаемого переменной окружения SHELL (ksh, bash…). Это обеспечивает независимость поведения родительского приложения от конкретных установок системы, в которой это приложение выполняется.

Вызов `system()` является не только простым, но и очень наглядным, делающим код легко читаемым. Программисты часто относятся к нему