Федеральное агентство по образованию

Санкт-Петербургский государственный
электротехнический университет «ЛЭТИ»

А. Ю. ДОРОГОВ

СИНХРОНИЗАЦИЯ И ВЗАИМОДЕЙСТВИЕ ПРОГРАММНЫХ ПОТОКОВ В ОПЕРАЦИОННОЙ СРЕДЕ РЕАЛЬНОГО ВРЕМЕНИ

Учебное пособие

Санкт-Петербург
Издательство СПбГЭТУ «ЛЭТИ»
2007

УДК 004.451+681.848.23(075)
ББК З 973.2 – 018.2я7
Д 69

Д 69 Дорогов А. Ю. Синхронизация и взаимодействие программных потоков в операционной среде реального времени: Учеб. пособие. СПб.: Изд-во СПбГЭТУ «ЛЭТИ», 2007. 64 с.

ISBN 5-7629-0774-0

Содержит описание механизмов синхронизации и управления программными потоками в операционных системах реального времени, удовлетворяющих стандарту POSIX и расширениям операционной системы QNX Neutrino.

Предназначено для студентов специальности 210100, а также может быть полезно инженерно-техническим работникам этой области знаний.

УДК 004.451+681.848.23(075)
ББК З 973.2 – 018.2я7

Рецензенты: ЗАО «СВД Софтвер» – Официальный дистрибьютор компании QSS; д-р. техн. наук, проф. Ю. И. Нечаев (ГМТУ).

Утверждено
редакционно-издательским советом университета
в качестве учебного пособия

Дорогов Александр Юрьевич

Синхронизация и взаимодействие программных потоков в операционной среде реального времени

Учебное пособие.
Редактор Э. К. Долгатов

Подписано в печать, 15.05.07. Формат $60\times84\ \frac{1}{16}$. Бумага офсетная.
Печать офсетная. Гарнитура «Arial». Печ. л. 4,0.
Тираж 120 экз. Заказ

Издательство СПбГЭТУ «ЛЭТИ»
197376, С.-Петербург, ул. Проф. Попова, 5

ISBN 5-7629-0774-0 © СПбГЭТУ «ЛЭТИ», 2007

Введение

Эффективность компьютерного управления технологическим объектом зависит от способности вычислительной системы обрабатывать множество событий одновременно. В современных операционных системах реального времени (ОСРВ) одновременность достигается разделением управляющей программы на параллельно работающие процессы, а процессов в свою очередь – на программные потоки. Многопоточная организация вычислительного процесса в настоящее время является общепринятым средством обеспечения высокой эффективности управляющих систем.

Создание многопоточных управляющих программ требует от разработчика глубоких знаний механизмов синхронизации, диспетчеризации и управления программными потоками. Синхронизация устанавливает правила совместного доступа потоков к критическим областям данных и условия одновременности исполнения программного кода. Проблема синхронизации заключается в том, что потоки в пределах процесса имеют общее адресное пространство, и если один поток пытается считать данные, в то время как другой их изменяет, то это может привести к катастрофическому отказу системы. В однопроцессорных системах потоки конкурируют за ресурсы процессора, фактически в этом случае они выполняются квазипараллельно и поэтому взаимно влияют друг на друга, порождая проблему эффективной диспетчеризации. Механизмы диспетчеризации потоков вместе с системой приоритетов устанавливают правила и условия переключения потоков. Дисциплина диспетчеризации непосредственно влияет на минимальное время реакции управляющей системы на внешние события. Механизмы управления потоками включают в себя средства их создания, уничтожения и динамического изменения атрибутов. Сервисы по синхронизации, диспетчеризации и управлению программными потоками возложены на операционную систему (ОС).

Для выполнения нескольких потоков одновременно в многозадачной операционной системе реального времени необходимо, чтобы ОС имела развитой механизм взаимодействия потоков между собой. Потоки отправляют и получают сообщения, а также отвечают на них. При этом изменяются состояния потоков. Определяя состояния потоков, ОС осуществляет их оптимальное планирование с максимально эффективным использованием ресурсов. Следовательно, механизм обмена сообщениями является основополагающим и постоянно действующим на всех уровнях ОС. Для приложений, работающих в режиме реального времени, требуется, чтобы меха-

низм межзадачного взаимодействия имел высокую степень надежности, поскольку процессы, на основе которых работают такие приложения, тесно связаны между собой.

Одной из общих проблем в области разработки приложений реального времени является обеспечение мобильности (переносимости, портабельности) программного обеспечения (ПО), т. е. возможности использования одного и того же программного кода в различных операционных системах. Это задача исключительной важности и сложности. Один из общепринятых способов повышения мобильности ПО – стандартизация окружения приложений, т. е. предоставляемых программных интерфейсов, утилит и т. п. На уровне системных сервисов подобные возможности определены международным стандартом POSIX (Portable Operating System Interface – мобильный интерфейс операционной системы). Название предложил известный специалист, основатель Фонда свободного программного обеспечения Ричард Столмэн. В данном учебном пособии рассматриваются сервисы синхронизации, диспетчеризации, управления и взаимодействия между потоками для современной версии стандарта, которую можно назвать «стандартом втройне», а именно: стандартом IEEE Std 1003.1, техническим стандартом Open Group и, что наиболее важно, международным стандартом ISO/IEC 9945. Информацию о текущей документации по стандартам POSIX можно найти в отчете Комитета по стандартам переносимых приложений (Portable Applications Standards Committee of the IEEE Computer Society) по интернет-адресу http://www.pasc.org/standing/sd11.html. Наиболее последовательно данный стандарт реализован в широкоизвестной операционной системе реального времени QNX Neutrino. Целевым назначением QNX Neutrino является реализация систем управления для широкого круга технологических приложений, начиная с небольших встроенных систем с ограниченными ресурсами и заканчивая крупными распределенными вычислительными средами. Данная ОСРВ поддерживает несколько семейств процессоров, в том числе x86, ARM, XScale, PowerPC, MIPS, SH-4. В настоящее время эта ОСРВ де-факто является полигоном для испытания и развития стандартов POSIX, поэтому далее большинство сервисов операционной системы описано в контексте QNX Neutrino.

1. Микроядро, потоки и процессы

Поток – это минимальная исполняемая единица операционной системы. Процесс – это контейнер потоков, имеющих общее адресное пространство. Любой процесс содержит по крайней мере один поток. Исполняющим

демоном потоков является микроядро операционной системы. Потоки могут взаимодействовать между собой обмениваясь сообщениями, сигналами, импульсами. В пределах процесса взаимодействие может также выполняться через совместно разделяемую память. Потоки подвергаются диспетчеризации микроядром вне зависимости от того, какому процессу они принадлежат, т. е. в системе QNX-Neutrino реализована потоковая модель исполнения программ.

Наглядной аналогией для модели процесса и его потоков может служить жилой дом и его обитатели. Дом реально представляет собой пассивный контейнер с некоторыми атрибутами (площадь, число комнат, бытовые удобства, состав семьи и. т. д.). Люди, живущие в доме, – активные объекты, они используют комнаты для жилья, готовят пищу, принимают душ, общаются между собой. Дом как пассивный контейнер аналогичен процессу, а обитатели дома подобны потокам. Если в доме живет только один человек, он в любой момент может использовать любые возможности дома: принять ванну, сготовить обед, включить телевизор и т. д. Ситуация в корне изменится, если в доме появится еще один человек, скажем, вы женитесь или выйдете замуж. Теперь уже вы не сможете попасть в душ в любой момент времени, не всегда вам удастся послушать любимую телепередачу или поесть то, что вам больше всего нравится. Каждому обитателю дома придется теперь всякий раз мысленно делать проверки на допустимость выполнения той или иной операции. Если вы ответственные люди, то будете уважать права другого обитателя дома, имея взамен общение, любовь, радость, ну а если нет, то взаимные обиды могут привести к трагедии и разрушению дома. Если ваша семья увеличится и в доме появятся дети, то станет еще интересней и сложней, возникнут определенные приоритеты в ваших поступках и привилегированные объекты, которых нужно кормить и оберегать. Из нашего повседневного опыта мы знаем, что семейная жизнь не так проста, как кажется. Институт семьи определен на уровне государства набором законов, дополнением к ним может служить брачный контракт, который как-то регламентируют взаимоотношения супругов, ну а еще, конечно, моральные и человеческие принципы взаимного уважения, которые заложены в нас воспитанием и уровнем развития общества. Совокупность законов и принципов определяет и набор действий, которые можно использовать в различных жизненных ситуациях.

В контексте операционной системы носителем законов и исполнительных средств является микроядро, которое обеспечивает все необходимые возможности жизнедеятельности процессов, включая их рождение и смерть.

1.1. Микроядро

Микроядро QNX-Neutrino инкапсулирует в себе системные свойства, необходимые для реализации встраиваемых систем реального времени, и включает в себя: сервис обмена сообщениями, синхронизацию и управление потоками и процессами, обработку внешних прерываний. Рис. 1.1 де-

монстрирует системный интерфейс и объекты, обслуживаемые ядром. Микроядро поддерживает системные вызовы, которые декларируют интерфейс управления:

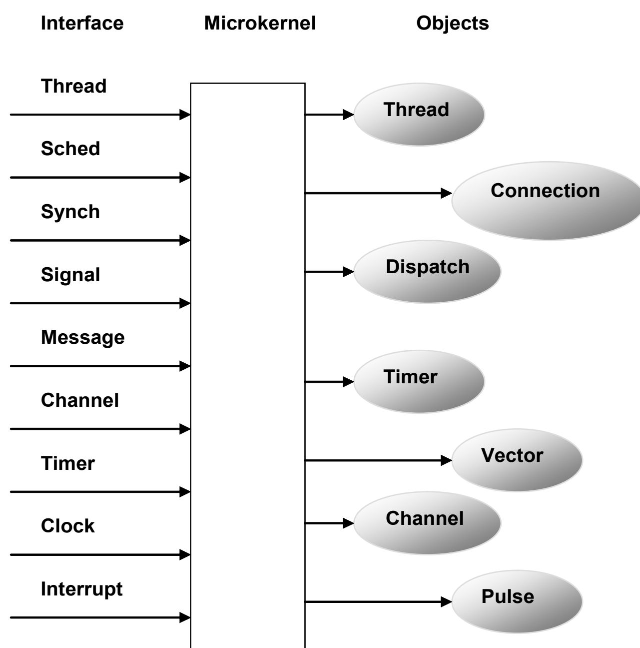

Рис. 1.1

- программными потоками (Thread);
- механизмами передачи сообщений (Message);
- механизмами передачи сигналов (Signal);
- каналами обмена данными (Channel);
- системными часами (Clock);
- таймерами (Timer);
- обработчиками прерываний (Interrupt);
- средствами синхронизации потоков (Synch);
- средствами диспетчеризации потоков (Sched).

Объекты ядра выполняют следующие функции:

- Потоки (Thread) инкапсулируют исполняемый программный код.
- Каналы (Channel) – это объект ядра, который используется потоком-сервером при реализации механизма передачи сообщений. Поток, который желает получить сообщение, создает канал (используя системный вызов ChannelCreate()), а затем получает сообщение из этого канала (используя вызов MsgReceive()).
- Системное соединение (Connection) – это объект ядра, который используется потоком-клиентом при реализации механизма передачи сообщений. Поток, который желает послать сообщение к потоку, создавшему канал, должен создать соединение с каналом (используя вызов ConnectAttach()), а затем переслать сообщение (используя вызов MsgSend()).
- Диспетчер сообщений (Dispatch) обеспечивает синхронизацию потоков при обмене сообщениями.
- Импульс (Pulse) – неблокируемое сообщение фиксированного размера.

• Таймер (Timer) – объект ядра, который периодически или однократно формирует события в заданные моменты времени.

• Векторные сообщения (Vector) – объект ядра, который используется при передаче длинных сообщений, имеющих предопределенную структуру. В векторном сообщении выделяется заголовок и составные компоненты.

1.2. Атрибуты потоков

Каждый поток имеет системную область памяти, предназначенную для размещения атрибутов потока. К атрибутам потока относятся:

• tid – идентификатор потока – это целое число, уникальное в пределах процесса;

• register context – область для хранения контекста состояния задачи, используемая при блокировках и прерываниях потока;

• stack – собственный стек – магазинная память для временных и рабочих переменных;

• signal mask – собственная маска для сигналов – запрещающая или, наоборот, разрешающая прием и обработку сигналов;

• cancellation handlers – функции, которые выполняются при завершении потока.

Системная область недоступна непосредственно из пользовательского кода, но может быть доступна через системные вызовы после предварительной процедуры связывания с конкретным атрибутом.

1.3. Состояния потока

Потоки создаются и уничтожаются динамически. Число потоков в процессе явно не ограничено. Любой поток может создать дочерний поток. При создании потока ему выделяются ресурсы памяти в пределах адресного пространства процесса. При завершении потока эти ресурсы освобождаются. Поток может находиться в трех состояниях: исполняться на процессоре (RUNNING), быть «готовым к исполнению» (READY), быть блокированным (BLOCKED) (рис. 1.2).

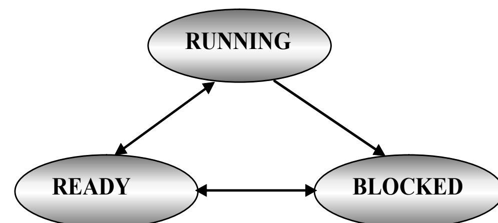

Рис. 1.2

Поток, «готовый к исполнению», не исполняется только потому, что процессор занят исполнением другого, возможно более приоритетного потока. Блокированный поток не может исполняться из-за ряда иных условий, к которым относятся:

• блокировка по условной переменной (CONDVAR);

• начата, но не закончена процедура завершения потока (DEAD);

• поток прерван другим более приоритетным потоком (INTERRUPT);

• поток блокирован из-за того, что некоторая общая критическая секция кода в данный момент принадлежит другому потоку (MUTEX);

• поток по своей инициативе «заснул» на некоторый интервал времени (NANOSLEEP);

• поток ждет ответа или подтверждения посылки при обмене сообщениями (REPLAY, NET_REPLAY, NET_SEND, RECEIVE);

• поток ожидает открытия семафора в заявленной группе потоков. Семафор ограничивает число потоков, одновременно «готовых к исполнению» (SEM);

• поток ожидает появления сигнала (SIGSUSPEND, SIGWAITINFO, STOPPED);

• поток ожидает освобождения сопроцессора, выполняющего операции с плавающей точкой (WAITCTX);

• поток ожидает распределения затребованной памяти (WAITPAGE);

• поток ожидает завершения дочернего потока (WAITTREAD);

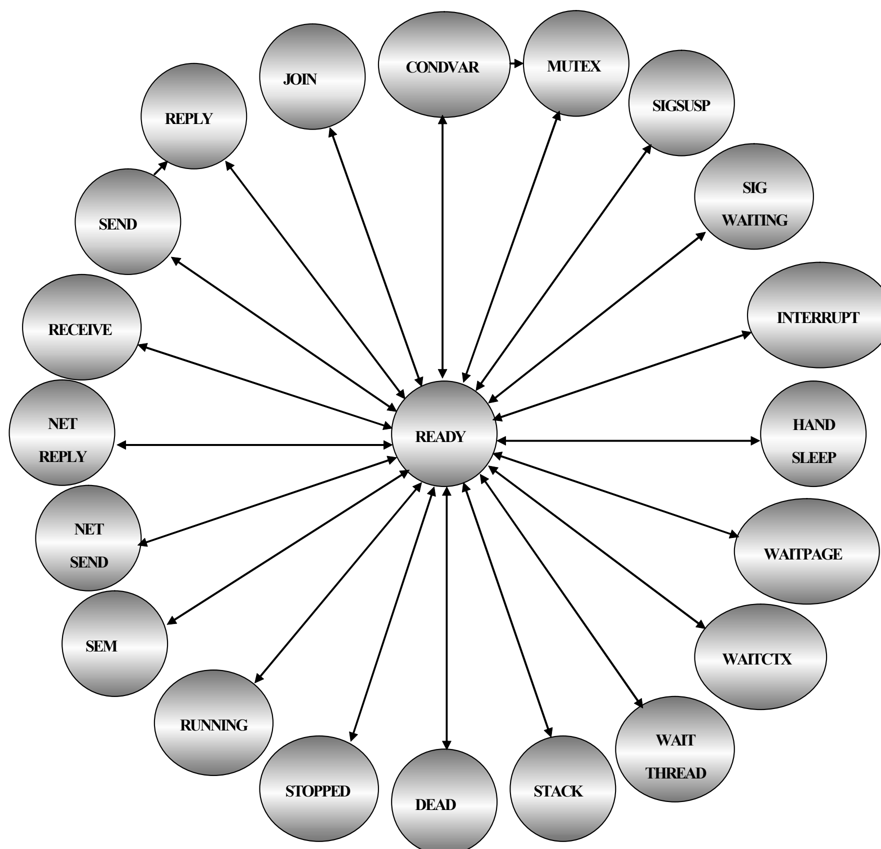

Рис. 1.3

• свободный поток ожидает присоединения к другому потоку, который становится его родительским потоком (JOIN).

Диаграмма на рис. 1.3 иллюстрирует возможные условия блокировки потоков. Важно помнить о том, что когда поток блокирован, то независимо от причины блокировки он не потребляет ресурсы процессора. Наоборот, единственным состоянием, в котором поток потребляет ресурсы процессора, является состояние выполнения (RUNNING).

При порождении потока по умолчанию устанавливается логическая связь нового потока с родительским. При наличии этой связи родительский поток «отслеживает» дочерний поток и фиксирует момент его завершения, получая от него «посмертную квитанцию» – код завершения. Если же родительский поток завершается, то завершаются и все дочерние потоки. Существует также возможность создать свободный поток, не обремененный какими-либо связями с родительским потоком.

1.4. Диспетчеризация потоков

Диспетчеризация – это процедура переключения процессора между потоками, «готовыми к исполнению». При переключении потоков контекст потока сохраняется в его системной области. Решение о переключении принимается ядром всякий раз, когда изменяется состояние исполняемого потока, при этом не имеет значения, какому процессу он принадлежит. Причинами изменения состояния потока могут быть:

- блокировка потока;
- изменение приоритета потока;
- собственное «желание» потока освободить процессор.

Среди потоков, готовых к исполнению, всегда исполняется тот поток,

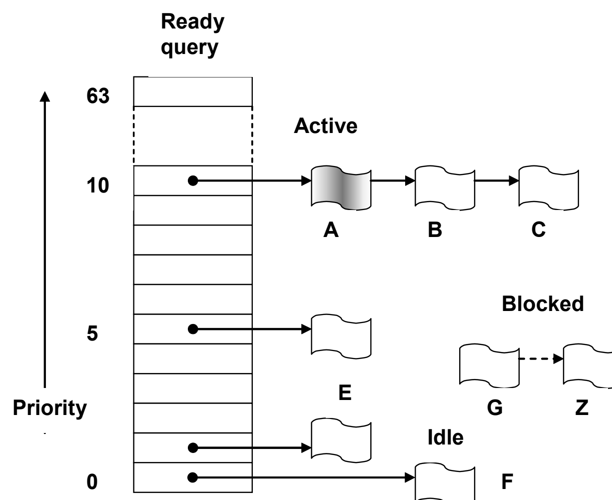

Рис. 1.4

который имеет наибольший приоритет. Если все потоки, готовые к исполне-

нию, имеют одинаковый приоритет, то они устанавливаются в очередь. Таким образом, можно считать, что для каждого уровня приоритета существует своя очередь (рис. 1.4). Перемещение потоков в очереди зависит от принятой дисциплины диспетчеризации. Операционная система поддерживает 256 уровней приоритета. В системе различаются обычные и привилегированные (корневые) потоки. Привилегированные потоки – это те, которые запущены привилегированным пользователем – обладающим правами администратора (его имя root). То, что поток привилегированный, определяется его идентификатором uid = 0, который при запуске наследуется от родительского потока. Привилегии явным образом не влияют на процедуру диспетчеризации, однако обычные потоки могут иметь значение приоритета от 1 до 63, а привилегированные – любые значения вплоть до 256. По умолчанию дочерние потоки наследуют свой приоритет от родительского потока. В системе существует поток, имеющий нулевой приоритет (idle-поток – пустой поток), он всегда готов к исполнению и заполняет паузу, когда системе делать нечего.

1.5. Дисциплины диспетчеризации

В системе QNX Neutrino поддерживаются 3 дисциплины диспетчеризации:

- FIFO (обычная очередь);
- циклическая (FIFO-очередь с установленными квантами времени исполнения);
- спорадическая (приоритет потока может изменяться во времени по определенным правилам).

Эти дисциплины применимы к потокам, которые имеют один и тот же приоритет. Если появляется поток с большим приоритетом, то он вытесняет все другие потоки и занимает процессор.

Дисциплина FIFO. Диаграмма на рис. 1.5, *а* иллюстрирует дисциплину диспетчеризации FIFO. Переключение потока может произойти, только когда поток самостоятельно передаст управление другому потоку (т. е. заблокируется), либо будет прерван более приоритетным потоком.

Циклическая дисциплина*.* Диаграмма на рис. 1.5, *б* иллюстрирует механизм циклической диспетчеризации. Передача управления другому потоку может произойти либо по инициативе активного потока, либо когда исчерпан отпущенный квант времени на исполнение (при этом поток будет поставлен в конец очереди), либо когда он будет прерван более приоритетным потоком.

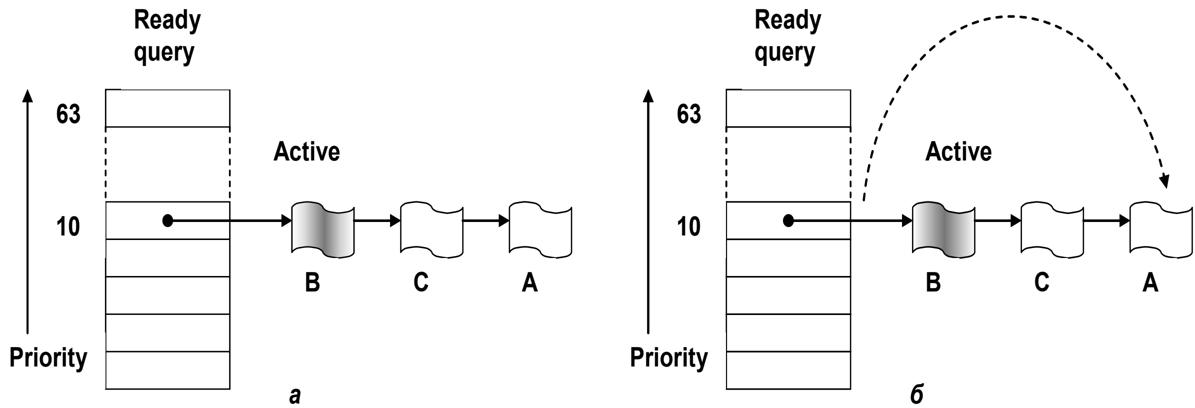

Рис. 1.5

Спорадическая дисциплина. Спорадическая диспетчеризация связана с изменением приоритета потока от заданного низкого уровня *L* до заданного нормального уровня *N* и обратно в зависимости от установленного бюджета времени *C*. Бюджет времени – это суммарный интервал времени, по истечении которого приоритет потока снижается от нормального *N* до низкого *L*. Кроме того, в алгоритме диспетчеризации используется параметр *T* – период восстановления. Этот период больше, чем значение начального бюджета *C*. Исчисление периода *T* начинается всякий раз, когда поток переходит в состояние *N*. С этого же момента открывается новый бюджет, который накладывается на предыдущий бюджет и существует с ним параллельно.

Следующая диаграмма (рис. 1.6) иллюстрирует функционирование механизма спорадической дисциплины диспетчеризации, когда поток не блокируется на нормальном уровне приоритета.

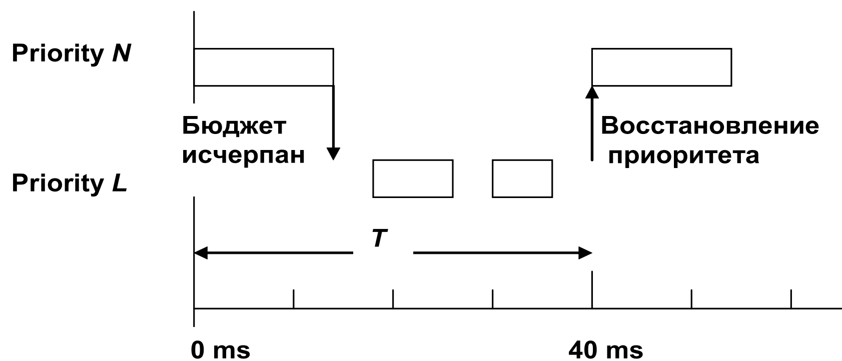

Рис. 1.6

На этой диаграмме показано, что приоритет потока падает до низкого уровня, после того как бюджет исчерпан. В этом состоянии он может получить или не получить шанс к исполнению, на диаграмме показан вариант, когда поток дважды получает возможность к исполнению на низком уровне приоритета. Приоритет потока вновь возвращается к нормальному уровню,

когда интервал восстановления *T* истечет. Восстановление приоритета до нормального происходит с интервалом *T* = 40 мс. Отношение *C/T* будет характеризовать средний уровень загрузки процессора данным потоком.

Если блокировка потока на уровне нормального приоритета *N* происходит многократно, то многократно запускается и период восстановления (рис. 1.7).

На приведенной диаграмме бюджет *C* = 10 мс, а период восстановления 40 мс.

Событие 1. Получив начальный бюджет 10 мс, после 3 мс исполнения поток блокируется.

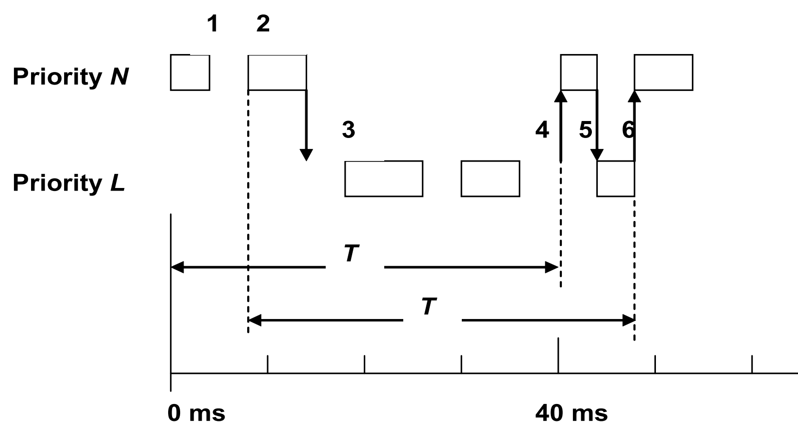

Рис. 1.7

Событие 2. Поток вновь получает возможность исполнения в момент времени 6 мс. Одновременно запускается новый интервал восстановления и открывается новый бюджет. Период работы потока длится 7 мс, поэтому он полностью исчерпывает свой первоначальный бюджет.

Событие 3. Приоритет потока падает до низкого уровня, поскольку весь первоначальный бюджет исчерпан.

Событие 4. В момент времени 40 мс происходит восстановление приоритета до нормального и запуск нового бюджета.

Событие 5. Поток на нормальном уровне работает в течение 3 мс до исчерпания прежнего бюджета, а затем его приоритет падает до низкого уровня.

Событие 6. В момент времени 40 + 6 = 46 мс истекает второй интервал восстановления и поэтому приоритет потока восстанавливается до нормального, при этом он имеет в текущем бюджете еще 7 мс и 10 мс в новом бюджете.

Восстановление приоритета в момент времени 46 мс считается ненормальным (отложенным), поскольку произошло раньше, чем ожидалось. Ядро подсчитывает число отложенных восстановлений и сравнивает его с

еще одним параметром спорадической дисциплины диспетчеризации – *K*. Как только число отложенных восстановлений достигнет значения *K*, приоритет потока будет снижен до значения *L* и поток будет находиться в этом состоянии до тех пор, пока не закончится период восстановления.

1.6. Управление приоритетами и дисциплиной диспетчеризации

В процессе своего исполнения приоритет потока может изменяться либо по инициативе самого потока, либо при получении сообщения от более приоритетного потока. Также может быть изменена дисциплина диспетчеризации. В табл. 1.1 приведены системные вызовы, позволяющие выполнить данные операции.

Таблица 1.1

POSIX-вызов	**Microkernel-вызов**	**Описание**
sched_getparam()	SchedGet()	Получить параметры диспетчеризации
sched_setparam()	SchedSet()	Установить параметры диспетчеризации
sched_getscheduler()	SchedGet()	Получить политику диспетчеризации
sched_setscheduler()	SchedSet()	Установить дисциплину диспетчеризации

В графе **POSIX-вызовы** представлены системные вызовы, соответствующие международному стандарту POSIX. В графе **Microkernel-вызовы** представлены системные вызовы ядра операционной системы QNX-Neutrino. Эти вызовы обладают обычно большими возможностями, и их можно рассматривать как расширения стандарта POSIX.

Существует такое понятие, как «масштаб диспетчеризации». POSIX определяет 2 масштаба: процесса и системы. При диспетчеризации в масштабе процесса ресурсы процессора делятся между процессами, а уже внутри процесса «разыгрываются» между потоками. При диспетчеризации в масштабе системы диспетчеризация выполняется на уровне потоков независимо от процессов, которым они принадлежат. В QNX реализован только масштаб системы.

1.7. Управление потоками

Под управлением потоком понимаются процедуры создания, уничтожения и изменения его атрибутов. В табл. 1.2 перечислены системные вызовы микроядра, которые используются для управления потоками.

Системный вызов создания потока имеет следующий формат:

```
int pthread_create( pthread_t* thread,          // идентификатор потока
          const pthread_attr_t* attr ,          // атрибуты потока
          void* (*start_routine)(void* ),       // программный код
          void* arg );                          // список параметров
```

В аргументе thread возвращается идентификатор нового потока (tid), который далее используется во всех вызовах управления потоком.

Аргумент (*start_routine) содержит указатель на функцию, содержащую программный код потока. Код потока должен быть предварительно создан программистом.

Таблица 1.2

POSIX-вызов	Microkernel-вызов	Описание
pthread_create()	ThreadCreate()	Создать новый поток
pthread_exit()	ThreadDestroy()	Уничтожить поток
pthread_detach()	ThreadDetach()	Отсоединить поток от родительского потока
pthread_join()	ThreadJoin()	Присоединить поток
pthread_cancel()	ThreadCancel()	Удалить поток на ближайшей точке завершения
N/A	ThreadCtl()	Изменить атрибуты потока

Аргумент void* arg является указателем на список параметров, передаваемых потоку. Эти параметры могут содержать какую-либо информацию, необходимую для настройки потока и его функционирования.

Аргумент attr указывает на структуру pthread_attr_t, содержащую атрибуты нового потока. Возможные атрибуты потоков и их значения по умолчанию представлены в табл. 1.3.

Таблица 1.3

Атрибут	Значение по умолчанию	Описание
detachstate	PTHREAD_CREATE_JOINABLE	Присоединить к родителю
schedpolicy	PTHREAD_INHERIT_SCHED	Наследовать дисциплину диспетчеризации
schedparam	Inherited from parent thread	Наследовать параметры диспетчеризации
contentionscope	PTHREAD_SCOPE_SYSTEM	Масштаб диспетчеризации
stacksize	4K bytes	Размер стека
stackaddr	NULL	Размещение стека
flags		Флаги

Атрибут detachstate. По умолчанию поток создается с атрибутом JOINABLE (присоединенный к родительскому). При завершении данного потока (с помощью вызова pthread_exit()) указатель value_ptr возвращается на определенную структуру данных, через которую дочерний поток может передать полезную информацию для родительского потока. Для получения данной структуры родительский поток (или любой другой поток) должен сделать системный вызов pthread_join(tid, value_ptr). До этого момента, несмотря на то что дочерний поток уничтожен, в памяти сохраняется его след «Зомби-поток». Если родительский поток сформировал вызов

pthread_join(tid, value_ptr) раньше, чем завершился дочерний поток, то он блокируется до завершения дочернего потока. При завершении родительского потока все связанные с ним дочерние потоки уничтожаются.

Если поток создан с атрибутом DETACHED, то, с одной стороны, он не сможет возвращать данные, а с другой – не будет уничтожен ядром при завершении его родителя.

Атрибут schedpolicy. По умолчанию задается жесткое наследование параметров и дисциплины диспетчеризации создаваемого потока от его родителя. Явным образом можно установить любую из трех допустимых дисциплин диспетчеризации. Дисциплина диспетчеризации и ее параметры задаются *атрибутом schedparam.*

Атрибут contentionscope (масштаб диспетчеризации). Значение по умолчанию PTHREAD_SCOPE_SYSTEM – диспетчеризация всех потоков вместе.

Атрибут stacksize (размер стека). По умолчанию 4 Кбайт.

Атрибут stackaddr (фактическое размещение стека). По умолчанию этот вопрос система решает самостоятельно.

Структура pthread_attr_t, содержащая атрибуты потока, инициализируется системным вызовом pthread_attr_init() и обеспечивает возможности, определенные стандартом POSIX. Кроме того, в QNX-Neutrino реализованы дополнительные возможности, несколько выходящие за пределы стандарта, определенные набором флагов в поле flags структуры pthread_attr_t. Флаги – это набор бит одного слова, которые складываются по «или».

В соответствии с POSIX при поступлении незамаскированного сигнала, для которого не определен обработчик, все потоки процесса должны быть уничтожены, однако при флаге PTHREAD_MULTISIG_DISALLOW будет уничтожен только поток, принявший сигнал. Остальные флаги предусмотрены POSIX и позволяют изменять поведение потока при завершении.

Если установлен флаг PTHREAD_CANCEL_DISABLE, то запрос на завершение потока не выполняется, но ставится в очередь и будет выполнен после сброса флага (т. е. установлен в значение PTHREAD_CANCEL_ENABLE).

Когда завершение разрешено, то анализируется еще один флаг, определяющий тип завершения – отсроченное, когда установлен флаг PTHREAD_CANCEL_DEFERED (по умолчанию), или асинхронное, когда установлен флаг PTHREAD_CANCEL_ASYNCHRONOUS. При асинхронном завершении поток будет уничтожен на первой же операции, которую он попытается выполнить. При отсроченном завершении поток будет выполнять-

ся, пока не достигнет ближайшей точки завершения (cancellation point). Точками завершения являются некоторые, как правило блокирующие, вызовы ядра и соответственно высокоуровневые функции, которые базируются на таких вызовах.

Кроме того, любой поток может зарегистрировать специальную функцию, которая будет вызываться, когда он завершается:

```
void pthread_cleanup_push( void (routine)(void*)        // указатель к функции
                          void* arg );                   // аргументы функции
```

Аргумент routine определяет имя функции, а arg – список ее входных параметров.

И последнее. При использовании многопроцессорной ЭВМ количество порождаемых для решения задачи потоков можно привязать к количеству процессоров. Это позволит оптимизировать использование ресурсов ЭВМ.

2. Механизмы синхронизации потоков

Потоки в пределах процесса имеют общее адресное пространство. Это означает, что если для процесса определены глобальные переменные, то любой поток может их «видеть» и изменять.

Таблица 2.1

Механизм синхронизации	Поддерживается между локальными процессами	Поддерживается между удаленными процессами
Мьютексы (Mutexes)	Yes	No
Условные переменные (Condvars)	Yes	No
Барьеры (Barriers)	No	No
Ждущие блокировки (Sleepon locks)	No	No
Блокировки Чтения/Записи (Reader/writer locks)	Yes	No
Семафоры (Semaphores)	Yes	Yes (только именованные)
FIFO-диспетчеризация (FIFO-scheduling)	Yes	No
Атомарные операции (Atomic operations)	Yes	No
Обмен сообщениями (Send/Receive/Reply)	Yes	Yes

Однако не так все просто. Может оказаться, что один поток читает данные в то время, когда другой поток их модифицирует. Это верный рецепт для провоцирования катастрофических ситуаций в системе. Для исключения подобных случаев и предназначены механизмы синхронизации, но служат они не только для этого. Часто возникает необходимость задать после-

довательность выполнения действий, например, когда результаты работы одного потока используются другим потоком. В этом случае второй поток должен подождать, пока первый поток закончит формирование промежуточного результата. Синхронизировать – это значит установить порядок или правила последовательного исполнения или доступа потоков (или процессов) к общим данным или критическим областям программы.

Мьютексы, семафоры и условные переменные являются примерами объектов, позволяющими разрешить указанные проблемы. Полный перечень механизмов синхронизации приведен в табл. 2.1.

Вторая графа таблицы определяет возможности использования указанных механизмов для синхронизации локальных процессов (в пределах одного узла). Третья графа определяет возможности сетевого использования данных механизмов. Объекты, поддерживающие механизмы синхронизации, могут быть созданы как статически (на этапе компиляции программы), так и динамически (при исполнении программы)

Таблица 2.2

POSIX вызов	Вызов микроядра	Описание
pthread_mutex_init()	SyncTypeCreate()	Создать мьютекс
pthread_mutex_destroy()	SyncDestroy()	Уничтожить мьютекс
pthread_mutex_lock()	SyncMutexLock()	Захватить мьютекс
pthread_mutex_trylock()	SyncMutexLock()	Условное блокирование на мьютексе
pthread_mutex_unlock()	SyncMutexUnlock()	Освободить мьютекс
pthread_cond_init()	SyncTypeCreate()	Создать условную переменную
pthread_cond_destroy()	SyncDestroy()	Уничтожить условную переменную
pthread_cond_wait()	SyncCondvarWait()	Ожидать на условной переменной
pthread_cond_signal()	SyncCondvarSignal()	Обращение к условной переменной
pthread_cond_broadcast()	SyncCondvarSignal()	Широковещательное обращение к условной переменной

В табл. 2.2 приведены вызовы ядра, используемые для динамического управления механизмами синхронизации.

2.1. Мьютексы

Mutex является сокращением фразы *Mutual exclusion locks* – взаимноисключающие блокировки. Этот механизм используется для обеспечения исключительного доступа к разделяемым данным. Этот объект подобен блокирующему замку в двери комнаты (критической секции): как только поток захватит мьютекс, никакой другой поток не сможет получить доступ к

критической секции, до тех пор пока замок не откроется. На рис. 2.1 показана мнемоническая схема функционирования мьютекса для потоков.

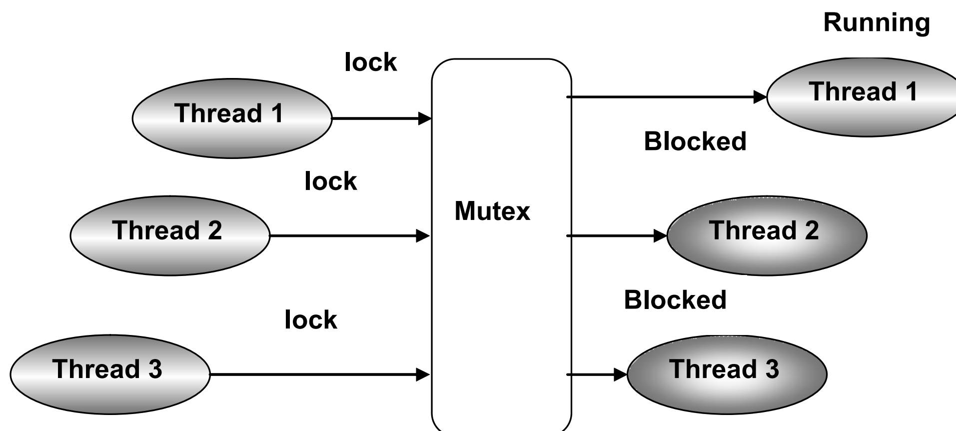

Рис. 2.1

Для реализации механизма используются 2 вызова, окружающие критическую секцию программного кода:

```
pthread_mutex_lock()        // захватить  мьютекс
pthread_mutex_unlock()      // освободить мьютекс
```

В любой момент времени только один поток может владеть мьютексом. Другие потоки, которые попытаются получить доступ к критической секции, будут блокированы, до тех пор пока владелец мьютекса не освободит его. Новым хозяином мьютекса станет конкурирующий поток, который имеет максимальный приоритет, а в случае равенства приоритетов – первый поток из очереди потоков, готовых к исполнению. В хорошей программе время исполнения критической секции должно быть минимальным. В примере 1 опущены системные вызовы создания мьютекса и программных потоков.

Существует также неблокирующий вызов (pthread_mutex_trylock()), который может быть использован для разовой проверки занятости мьютекса. В этом случае проверяющий поток не блокируется, но получает информацию о состоянии мьютекса.

ПРИМЕР 1

```
int count = 0;  /* Критическая область состоит из одной глобальной переменной count */
/****************************************************************/
void* function1( void* arg ) // поток 1
{
while( 1 ) { // бесконечный цикл
pthread_mutex_lock( &mutex ); /* захват мьютекса (неудача приводит к блокировке текущего потока) */
count++; // изменение критической переменной (увеличение на 1)
 pthread_mutex_unlock( &mutex );  // Освобождение мьютекса
   }
return 0;
}
/****************************************************************/
void* function2( void* arg ) // поток 2
```

```
{
while( 1 ) { // бесконечный цикл
pthread_mutex_lock( &mutex ); /* захват мьютекса (неудача приводит к блокировке те-
кущего потока) */
count--; // изменение критической переменной (уменьшение на 1)
pthread_mutex_unlock( &mutex ); // освобождение мьютекса
   }
return 0;
}
```

Если поток с приоритетом, большим чем приоритет владельца мьютекса, пытается захватить мьютекс, то эффективный приоритет текущего собственника будет увеличиваться до уровня максимального приоритета группы потоков, ожидающих мьютекс, что приводит к быстрому исполнению критической секции и освобождению мьютекса. Прежний собственник мьютекса после исполнения возвращается к своему реальному приоритету. Эта схема обеспечивает малое ожидание для высокопроизводительных процессов.

При инициализации мьютекса можно задать 2 его атрибута:

- protocol с возможными значениями:

– PTHREAD_PRIO_INHERIT (значение по умолчанию) – увеличивает приоритет текущего собственника мьютекса до уровня максимального приоритета группы потоков, ожидающих мьютекс;

– PTHREAD_PRIO_PROTECT – увеличивает приоритет текущего собственника мьютекса до максимума в группе всех потоков, инициализировавших данный мьютекс с атрибутом PTHREAD_PRIO_PROTECT в независимости от того, блокированы эти потоки или нет в текущий момент;

- recursive с возможными значениями:

– PTHREAD_RECURSIVE_ENABLE – собственник мьютекса может захватить его опять без блокировки;

– PTHREAD_RECURSIVE_DISABLE (значение по умолчанию) – любой поток, в том числе и собственник мьютекса, блокируется при попытке захвата мьютекса, если он уже захвачен.

2.2. Дополнительные сервисы QNX-Neutrino

Стандарт POSIX утверждает, что с мьютексом должны работать потоки одного и того же процесса, но позволяет в соответствующей реализации эту концепцию расширять. В QNX-Neutrino это расширение сводится к тому, что мьютекс может использоваться потоками различных процессов. Объясняется это тем, что ядро работает только с потоками и ему все равно какие потоки работают в каких процессах.

Другое существенное дополнение QNX-Neutrino – это пулы потоков. Пул – это несколько потоков-клонов, полностью идентичных друг другу. Пул

предназначен для того, чтобы увеличить скорость обслуживания запросов. Идея состоит в том, что если один из потоков пула занят обслуживанием и не может принять запрос на обработку, тогда другой поток из этого пула обслужит новый запрос. При необходимости размером пула можно управлять.

2.3. Условные переменные

Условные переменные, или CONDVAR, используются для блокировки потока до тех пор, пока не будет выполнено некоторое условие. Условие может быть произвольной сложности и не зависит от объекта CONDVAR. Объект CONDVAR должен быть всегда использован с мьютексом, для того чтобы выполнить безопасный мониторинг условия. Объект CONDVAR поддерживается тремя системными вызовами:

wait (pthread_cond_wait()) – **блокировка по условию**
signal (pthread_cond_signal()) – **деблокировка приоритетного потока**
broadcast (pthread_cond_broadcast()) – **деблокировка всех потоков**

Рассмотрим использование этих вызовов на примере. В примере 2 определены 2 конкурирующих потока consume (потребитель) и produce (производитель).

ПРИМЕР 2

```
int condition = 0; // условие, связанное с cond
int count = 0; // переменная критической секции
/**********************************************************/
int consume( void ) // поток-потребитель данных
{
  while( 1 )
  {
    pthread_mutex_lock( &mutex ); // захват мьютекса
    while( condition == 0 )
      pthread_cond_wait( &cond, &mutex ); /* блокировка потока по  CONDVAR cond */
    printf( "Consumed %d\n", count );
    condition = 0;
    pthread_cond_signal( &cond ); // деблокировка потока, ожидающего cond
    pthread_mutex_unlock( &mutex ); // освобождение мьютекса
  }
  return( 0 );
}
/**********************************************************/
void*  produce( void * arg ) // поток-производитель данных
{
  while( 1 )
  {
    pthread_mutex_lock( &mutex ); // захват мьютекса
    while( condition == 1 )
      pthread_cond_wait( &cond, &mutex ); /* блокировка потока по CONDVAR cond */
    printf( "Produced %d\n", count++ );
    condition = 1;
    pthread_cond_signal( &cond );       // деблокировка потока, ожидающего cond
    pthread_mutex_unlock( &mutex );   // освобождение мьютекса
  }
  return( 0 );
}
```

Каждый поток проверяет значение переменной condition. Мьютекс захватывается активным потоком, прежде чем выполняется мониторинг условия condition. В результате гарантируется, что только поток, захвативший мьютекс, имеет возможность проверить условие, логически связанное с CONDVAR (cond). Если это условие истинно, то выполняется вызов pthread_cond_wait(&cond, &mutex) и текущий поток блокируется, а связанный с ним мьютекс освобождается. Это дает возможность другому потоку захватить мьютекс и изменять критическую переменную count. Текущий поток будет блокирован до тех пор, пока другой поток не выполнит операцию signal или broadcast для условной переменной cond. В этом случае мьютекс вновь захватывается текущим потоком и освобождается только по вызову pthread_mutex_unlock(&mutex).

While-петля требуется по двум причинам. Во-первых, стандарт POSIX не гарантирует от ложных пробуждений в мультипроцессорных системах. Во-вторых, если другой поток изменил условие, то требуется отследить это изменение в текущем потоке.

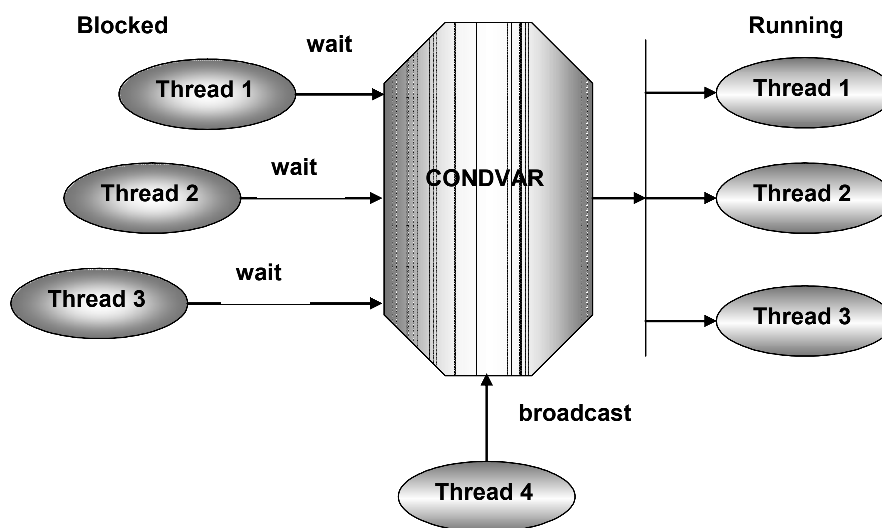

Рис. 2.2

Любой поток, который выполняет операцию signal, будет деблокировать наиболее приоритетный поток, стоящий в очереди к условной переменной. При выполнении любым потоком операции broadcast будут деблокированы все потоки, стоящие в очереди к условной переменной. Связанный мьютекс захватывается автоматически высокоприоритетным неблокированным потоком. Поток должен освободить мьютекс после обработки критической секции. На рис. 2.2 показана мнемоническая схема функционирования механизма синхронизации CONDVAR при выполнении операции broadcast.

Механизму условных переменных можно сопоставить следующую наглядную аналогию из спортивной жизни. Представьте себе олимпийский стадион, трибуны которого практически полностью заполнены в ожидании предстоящего матча. Стадион имеет несколько входов, которые осаждают болельщики. Каждый вход стадиона связан с собственной билетной кассой и независимым сектором на трибунах. Болельщики пытаются проникнуть на стадион, честно купив билет в кассе сектора, однако их желания блокируются ответом кассира, что билетов нет. Но у болельщиков каждого сектора есть один законный представитель на трибуне, с которым они могут общаться по телефону. Представитель по запросу проверяет фактическое наличие мест в секторе. Каждый болельщик пытается дозвониться своему представителю. Понятно, что в каждый момент времени только один болельщик может говорить с представителем сектора. По требованию болельщика представитель контролирует сектор, и предположим, что он обнаружил одно свободное место. Тогда он сообщает, что есть одно свободное место, и в кассе появляется один билет. Наибольшим приоритетом в получении билета обладает болельщик, который стоит в очереди первым, скорей всего он и получит заветный билет (если, конечно, в толпе ожидающих болельщиков нет VIP-персоны). Может оказаться (хотя на мировом первенстве это маловероятно!), что представитель обнаружил несколько мест и это число больше или равно числу ожидающих болельщиков на входе данного сектора, тогда они все смогут приобрести билеты и попасть на стадион.

В данной ситуации болельщики соответствуют программным потокам, различные входы на стадион с билетным кассиром – различным объектам CONDVAR, представитель с последовательным доступом соответствует потоку-поставщику информации, связанному с конкретным объектом CONDVAR. Критические ресурсы, охраняемые мьютексом, – это разговор по телефону, а проверяемым условием является фактическое наличие мест в секторе. Представитель является источником либо вызова signal, когда проходит один болельщик, либо broadcast – когда пропускают всех болельщиков с данного входа. Ситуацию, когда в кассу поступило несколько билетов, но меньше, чем ожидающее число болельщиков, для чистоты аналогии исключаем из рассмотрения. Болельщики, попавшие на трибуну, занимают свободные места, а представитель вновь начинает поиск свободных мест по запросам болельщиков.

Существует другой вид операции ожидания условной переменной (pthread_cond_timedwait()) который позволяет установить тайм-аут. По окончании этого периода ожидающий поток может быть разблокирован.

2.4. Ждущие блокировки

Типовая ситуация в многопоточных программах – это потребность заставить поток «ждать чего-либо», например готовности данных от внешнего устройства, заданной позиции конвейерной ленты, доступа к диску и т. д., т. е. ждать освобождения какого-либо ресурса. Причем одного и того же события могут ожидать несколько потоков. Для таких целей можно было бы использовать условную переменную (CONDVAR), но гораздо проще использовать ждущую блокировку (SLEEPON-LOCK). В действительности ждущие

блокировки – это надстройка над механизмом условных переменных. Для применения ждущих блокировок необходимо выполнить несколько операций. Системные вызовы, обслуживающие этот механизм синхронизации, представлены в табл. 2.3.

Таблица 2.3

Функция	Описание
pthread_sleepon_lock()	Установить механизм синхронизации
pthread_sleepon_unlock()	Снять механизм синхронизации
pthread_sleepon_broadcast()	Деблокировка всех потоков, ожидающих событие
pthread_sleepon_signal()	Деблокировка наиболее приоритетного потока, ожидающего событие
pthread_sleepon_wait()	Блокировать поток на ожидании события

Прежде всего поток, которому требуется ресурс, должен проверить, надо ли ему ждать. С этой целью он должен проконтролировать некоторый флаг готовности ресурса, и если ресурс доступен, то при его использовании сбросить флаг готовности. Но такие проверки и модификации могут быть одновременно от нескольких потоков, поэтому понадобится некоторая форма монопольного доступа к флагу, чтобы в любой момент времени он был доступен только одному потоку.

Метод, который применяется в данном случае, – это мьютекс, но это внутренний мьютекс библиотеки ждущих блокировок, так что обращаться к нему можно только с помощью двух функций: pthread_sleepon_lock() и pthread_sleepon_unlock(). В примере 3 флагом доступа является переменная data_ready. В примере определено 2 потока: Consume («потребитель») и Producer («производитель»). Поток Consumer выполняет установку и снятие ждущей блокировки. Это означает, что потребитель может теперь надежно проверять и модифицировать флаг data_ready, не опасаясь гонок.

Пример из спортивной жизни с механизмом ждущих блокировок трансформируется так, что остается один представитель болельщиков на весь стадион, а число работающих входов динамически изменяется: увеличивается – по настоянию болельщиков или уменьшается – по желанию администрации.

ПРИМЕР 3

```
Consumer () // поток-потребитель
{
  while (1) {
    pthread_sleepon_lock();
    while (!data_ready) { /* разрешить проверку наличия события,  если  data_ready=0 */
                pthread_sleepon_wait((&data_ready);  // ждать событие
          }
                // обработать данные
                data_ready=0;
                pthread_sleepon_unlock();
```

```
            }
    }
    /*********************************************************/
    Producer () // поток-производитель
    {
       while (1) {
                    // ждать прерывания от оборудования
                    pthread_sleepon_lock();
                    data_ready=1;
                    pthread_sleepon_signal(&data_ready)
                    pthread_sleepon_unlock()
                }
    }
```

Функция pthread_sleepon_wait() в действительности выполняет 3 действия:

- разблокирует мьютекс библиотеки ждущих блокировок;
- выполняет собственно операцию ожидания события;
- снова блокирует мьютекс библиотеки ждущих блокировок.

Причина обязательной разблокировки/блокировки мьютекса библиотеки проста: необходимо запретить потоку-производителю изменять флаг. Но если не разблокировать флаг впоследствии, то поток-производитель не сможет его установить, чтобы сообщить о доступности данных.

Поток-производитель также блокирует мьютекс, чтобы получить монопольный доступ к флагу data_ready перед его установкой. Производитель вызывает pthread_sleepon_signal(), «пробуждая» наиболее приоритетного потребителя данных, и деблокирует мьютекс. Приоритетный потребитель, блокированный по вызову pthread_sleepon_wait(), возвращается из состояния ожидания события и снова захватывает мьютекс. Флаг data_ready в данном примере служит для двух целей:

- Он является флагом состояния – посредником между «Потребителем» и «Производителем», указывающим на состояние системы. Если флаг установлен в состояние 1, это означает, что данные доступны для обработки; если этот флаг установлен в состояние 0, это означает, что данных нет и поток должен быть заблокирован.
- Он выполняет функцию «места, где происходит синхронизация со ждущей блокировкой». Другими словами, адрес переменной data_ready используется как уникальный идентификатор объекта, по которому осуществляется ждущая блокировка. Все равно, что это за идентификатор, лишь бы он был уникален и корректно использовался. Использование же в качестве идентификатора адреса переменной есть надежный способ сгенерировать уникальный номер, поскольку не бывает же двух переменных с одинаковым адресом!

2.5. Ждущие блокировки в сравнении с условными переменными

Ждущие блокировки имеют одно основное преимущество в сравнении с условными переменными. Предположим, что необходимо синхронизировать множество объектов. При использовании условных переменных с каждым объектом ассоциируется отдельная условная переменная, т. е. для *M* объектов необходимо определить *M* условных переменных. При применении же ждущих блокировок соответствующие им условные переменные создаются динамически по мере постановки потоков на ожидание, поэтому в данном случае на *M* объектов и *N* блокированных потоков потребовалось бы максимум *N*, а не *M* условных переменных. Однако условные переменные более универсальны, чем ждущие блокировки, поскольку:

1) ждущие блокировки в любом случае основаны на условных переменных;
2) мьютексы ждущих блокировок скрыты в библиотеке.

Второй пункт имеет еще и практический смысл. Если мьютекс скрыт в библиотеке, он может быть только один на процесс, независимо от числа потоков в этом процессе или от количества переменных. Данный фактор может стать сильно ограничивающим. Гораздо лучше применять несколько мьютексов – по одному на каждый набор данных – и явно сопоставлять им условные переменные по мере необходимости.

2.6. Барьеры

Барьер – это механизм синхронизации, который позволяет логически интегрировать результаты работы нескольких потоков, заставляя их ждать

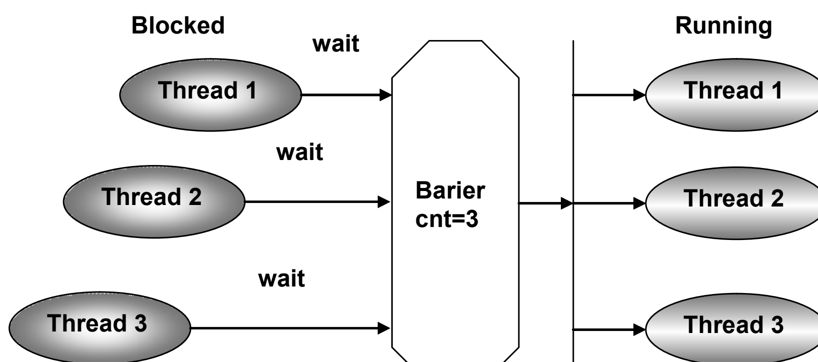

Рис. 2.3

на определенной точке до тех пор, пока число заблокированных потоков не достигнет установленного значения. Только после этого все потоки деблокируются и могут продолжить свое исполнение. Таким образом, барьер – это некоторое рандеву для потоков в определенной точке. Когда специфицированный набор потоков достигнет барьера, все потоки деблокируются и

продолжают исполнение. Мнемоническая схема барьера показана на рис. 2.3.

> Аналогией из городской жизни является ситуация на конечной остановке маршрутного такси. Строгий водитель-частник, подогнав микроавтобус, ждет, пока все места будут заняты. Пассажиры недовольны, они спешат домой и по другим личным делам, но вынуждены ждать полного заполнения маршрутного такси, поскольку оно частное и правилам общественного транспорта не подчиняется. И водителю, и пассажирам совершенно все равно, кто займет места, лишь бы это случилось как можно быстрей. Только когда все места заняты, маршрутное такси трогается с места. В этой аналогии число мест микроавтобуса соответствует счетчику барьера, а запросы wait – ожиданиям пассажиров уже занявших места в микроавтобусе.

Объект *барьер* может быть создан вызовом:

```
int pthread_barrier_init(pthread_barrier_t *barrier,          // указатель к барьеру
          const pthread_barrierattr_t *attr,                  // атрибуты барьера
          unsigned int count);                                // счетчик барьера.
```

Счетчик count определяет число потоков, которые должны вызывать функцию int pthread_barrier_wait(pthread_barrier_t *barrier);

После создания барьера каждый поток будет вызывать эту функцию для указания на завершение этапа вычислений. При вызове данной функции поток блокируется до тех пор, пока число блокированных потоков не достигнет значения счетчика. После этого все потоки будут деблокированы. Для обслуживания механизма синхронизации *барьер* используются следующие системные вызовы (табл. 2.4).

Таблица 2.4

Функция	**Описание**
pthread_barrierattr_getpshared()	Получить текущее значение атрибута разделения барьера
pthread_barrierattr_destroy()	Уничтожить атрибутный объект барьера
pthread_barrierattr_init()	Инициализировать атрибутный объект барьера
pthread_barrierattr_setpshared()	Установить значение атрибута разделения барьера
pthread_barrier_destroy()	Уничтожить барьер
pthread_barrier_init()	Инициализировать барьер
pthread_barrier_wait()	Синхронизировать поток с барьером

При создании барьера формируется указатель на объект атрибутов, который далее должен быть инициализирован. Атрибут разделения барьера может иметь 2 значения:

- PTHREAD_PROCESS_SHARED – разрешает барьеру синхронизировать потоки различных процессов;
- PTHREAD_PROCESS_PRIVATE (значение по умолчанию) – разрешает барьеру синхронизировать потоки в пределах одного процесса.

2.7. Семафоры

Семафор является другой общей формой синхронизации, которая позволяет потокам объявлять свою активность и ждать на семафоре возможности исполнения.

> Для пояснения принципа действия семафора рассмотрим аналогию из повседневной жизни. Предположим, что на входной двери дома висит несколько ключей. Человек берет ключ, входит в дом и закрывает дверь изнутри. При этом количество ключей снаружи уменьшается на единицу. Если человек покидает дом, то он закрывает дверь и вешает ключ снаружи, количество ключей снаружи при этом увеличивается на единицу. Максимальное число людей, которые могут одновременно находиться в доме, очевидно, равно числу существующих ключей. Если ключ один, то только один человек сможет находиться в доме.

Создание семафора эквивалентно объявлению о числе существующих ключей. Для инициализации семафора используется системный вызов

```
int sem_init( sem_t * sem,        // указатель к объекту семафор
        int pshared,              // сфера действия семафора
        unsigned value );         // число пропускаемых потоков
```

Если pshared ≠ 0, то семафор разделяемый, он управляет потоками вне зависимости от их принадлежности к процессам.

Если pshared = 0, то семафор управляет потоками только в пределах одного процесса. Каждый поток может взаимодействовать с семафором, используя две функции:

```
int sem_post( sem_t * sem );
int sem_wait( sem_t * sem );
```

Функция post увеличивает значение счетчика семафора на единицу, а функция wait декрементирует его. Если значение семафора не положитель-

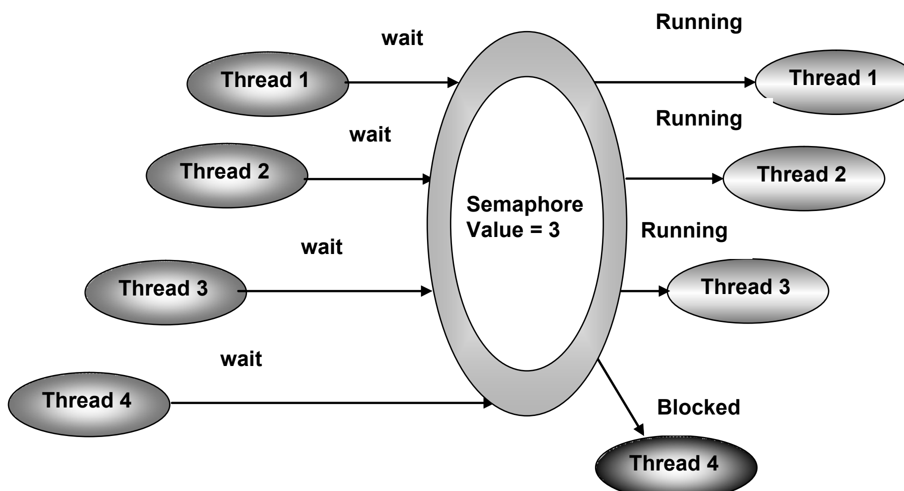

Рис. 2.4

но, то очередной поток блокируется по функции wait. Мнемоническая схема работы семафора представлена на рис. 2.4.

По сравнению с другими механизмами синхронизации семафоры более безопасны в асинхронных приложениях. Другое полезное свойство семафоров в том, что стандартом POSIX они определены не только для синхронизации потоков в пределах одного процесса, но и для синхронизации потоков различных процессов.

Таблица 2.5

Функция	Описание
sem_init()	Инициализировать объект *семафор*
int sem_post()	Увеличить текущий счетчик
int sem_wait()	Уменьшить текущий счетчик
sem_trywait()	Уменьшить текущий счетчик, если его значение больше нуля
sem_destroy()	Уничтожить семафор

Существуют также именованные семафоры, которые используются при взаимодействиях через сеть. В табл. 2.5 представлен полный набор функций семафоров.

2.8. Блокировки чтения/записи

Блокировки чтения/записи используются, когда область данных доступна нескольким потокам по чтению и одному потоку по записи. Эта ситуация возникает достаточно часто при совместном использовании структуры данных группой потоков. Иначе этот механизм называется «много читате-

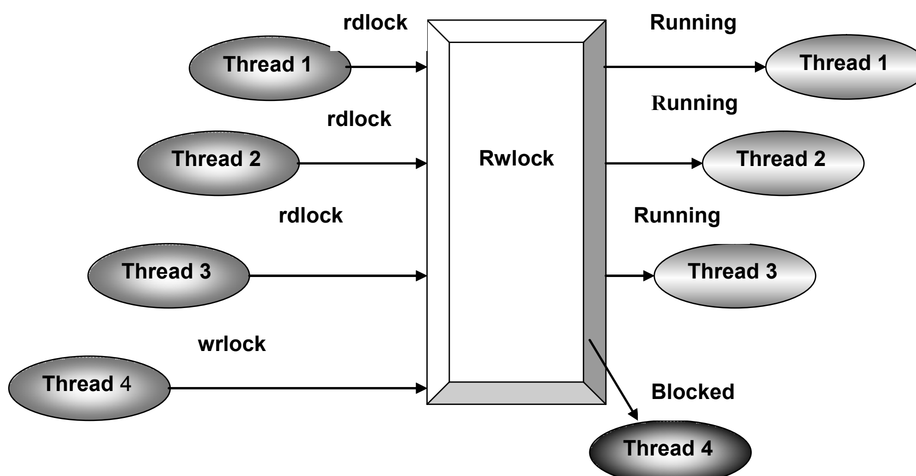

Рис. 2.5

лей, один писатель». Поскольку считывание области данных – неразрушающая операция, любое число потоков может считывать данные (даже если ту же часть данных считывает другой поток). Принципиально важным моментом здесь является то, что никто не должен производить запись в об-

ласть данных, из которой в этот момент ведется чтение. В противном случае может нарушиться целостность данных. Блокировка создается вызовом

```
int pthread_rwlock_init(
        pthread_rwlock_t * rwl,              // объект блокировки
        const pthread_rwlockattr_t * attr ); // атрибуты
```

где rwl – указатель к объекту *блокировка чтения/записи*, attr – указатель к набору атрибутов. Блокировка разрешает всем потокам доступ по чтению к некоторой области данных с помощью вызова:

```
pthread_rwlock_rdlock( pthread_rwlock_t* rwl );
```

но когда поток, желающий выполнить запись, обращается с вызовом

```
pthread_rwlock_wrlock(pthread_rwlock_t* rwl ),
```

то требование записи блокируется до тех пор, пока все читающие потоки завершат чтение области данных. Множество пишущих потоков упорядочивается в очередь (с учетом приоритетов) и ожидает возможности выполнить запись к защищенной структуре данных. Приоритет читающих потоков не учитывается.

На рис. 2.5 показана мнемоническая схема функционирования механизма rwlock, когда активны потоки, выполняющие операции чтения. На рис. 2.6 показано функционирование механизма rwlock, когда активен пишущий поток.

Имеются также специальные системные вызовы

```
pthread_rwlock_tryrdlock(pthread_rwlock_t* rwl );
pthread_rwlock_trywrlock(pthread_rwlock_t* rwl ),
```

позволяющие потокам проверить на допустимость выполнения действия без блокировки потока. Важно отметить, что «проверочные версии» захватывают объект rwlock, если он доступен. Это необходимо для того, чтобы

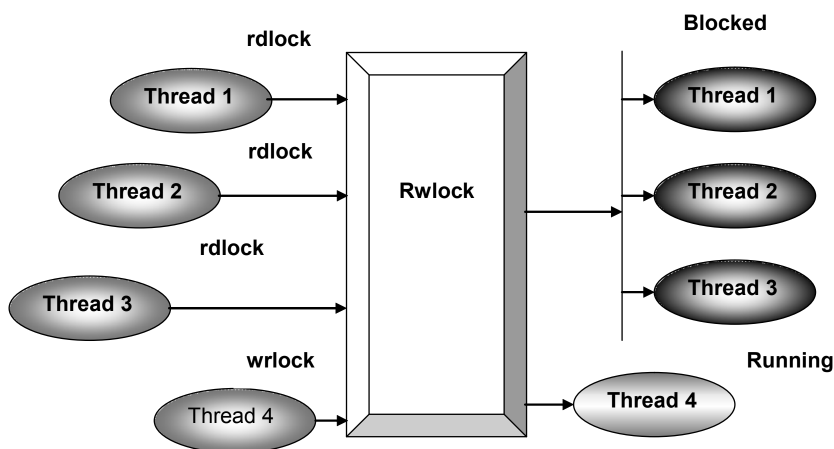

Рис. 2.6

исключить опасную паузу между проверкой и доступом, поскольку в пределах этой паузы может произойти модификация данных.

Отметим, что невозможно реализовать такую форму синхронизации с помощью мьютекса, поскольку он рассчитан только на один поток, что было бы хорошо в случае записи, но мьютекс не способен разрешить доступ к разделяемой области данных нескольким потокам в случае чтения. Семафор также бесполезен, потому что не позволяет различить 2 режима доступа – применение семафора могло бы обеспечить доступ нескольких «читателей», но при попытке «писателя» завладеть семафором его вызов ничем бы не отличался от вызовов «читателей», что привело бы к ситуации с множеством неблокированных «писателей».

2.9. Синхронизация через диспетчеризующий механизм

Выбор дисциплины диспетчеризации типа FIFO («первый вошел – первый вышел») гарантирует, что никакие 2 потока с одинаковым приоритетом не смогут одновременно выполнять критическую секцию, а будут делать это только последовательно, когда один поток передаст процессор другому потоку. При практической реализации этого механизма необходимо:

- обеспечить условия, при которых все потоки, имеющие доступ к критической секции, имели бы одинаковый приоритет;
- тщательно документировать программный продукт, организационно исключив возможность изменения дисциплины диспетчеризации FIFO;
- не применять данный механизм для многопроцессорных платформ.

2.10. Синхронизация через атомарные функции

Термин *атомарный* означает неделимый. В некоторых случаях может потребоваться выполнить короткую математическую операцию с гарантией, что она не будет прервана другим потоком или процедурой прерывания. В QNX-Neutrino этот механизм обеспечивается атомарными операциями, представленными в табл. 2.6.

Таблица 2.6

Функция	Описание
atomic_add(), atomic_add_value()	Добавление значения
atomic_clr(), atomic_clr_value()	Сброс бит
atomic_set(), atomic_set_value()	Установка бит
atomic_sub(), atomic_sub_value()	Вычитание значения
atomic_toggle(), atomic_toggle_value()	Переключение бит

Все операции выполняются над одной переменной. Версии операций с добавкой _value возвращают, кроме того, прежние значения переменной.

Атомарные операции могут быть использованы где угодно, но особенно полезны они в двух случаях:

- между процедурой обработки прерывания (*ISR*) и потоком;
- между двумя потоками.

Так как ISR может вытеснить поток в любой точке, то естественный путь защитить поток от прерываний – запретить все прерывания. Для систем реального времени это плохо, так как ведет не только к снижению производительности, но и к возможной потере внешних событий. Использование атомарных функций позволяет решить данную проблему.

2.11. Синхронизация через сообщения

Сообщение представляет собой группу байт, значение которых имеет смысл только для взаимодействующих потоков и не имеет какого-либо абсолютного значения для ядра операционной системы.

Таблица 2.7

Функция	Описание
MsgSend()	Послать сообщение и блокироваться до получения ответа
MsgReceive()	Ожидать приема сообщения
MsgReply()	Ответить на сообщение
MsgError()	Возвратить код ошибки без передачи ответного сообщения

Потоки получают и отправляют сообщения, а также отвечают на них. При этом состояния потоков изменяются. Передача сообщений между пото-

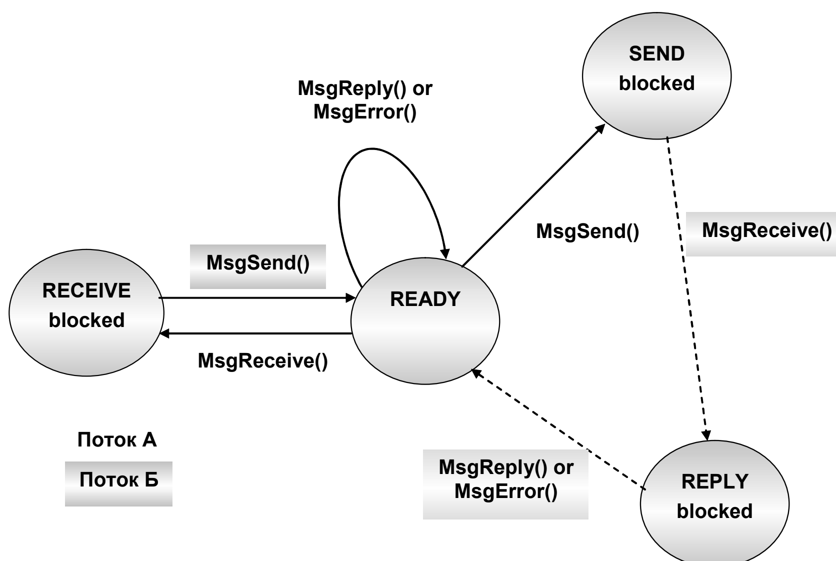

Рис. 2.7

ками связана с блокировками, т. е. механизм синхронизации встроен в про-

цедуры передачи сообщений. Базовый механизм передачи сообщений обеспечивается системными вызовами, представленными в табл. 2.7.

Поток (клиент), который выполняет вызов MsgSend() передачи сообщения (рис. 2.7) к другому потоку, SEND-блокируется до тех пор, пока принимающий поток (сервер) не выполнит вызов MsgReceive(), после чего передающий поток переходит в REPLY-блокированное состояние и ждет ответа или кода ошибки. Принимающий поток обработает сообщение и сформирует ответ с помощью вызова MsgReply() или код ошибки с помощью вызова MsgError(). Далее происходит деблокировка передающего потока. Если принимающий поток выполняет вызов MsgReceive() до передачи сообщения, то он RECEIVE-блокируется до тех пор, пока передающий поток не выполнит вызов MsgSend().

2.12. Тайм-ауты ядра

Тайм-аут (timeout) – это системное средство, позволяющее ограничить время пребывания потока в блокированном состоянии. Наиболее часто такая потребность возникает при обмене сообщениями: клиент, посылая сообщение серверу, не желает ждать ответа вечно. В этом случае удобно использовать вызовы ядра, устанавливающие тайм-ауты на состояния блокировки. Тайм-аут также полезен в сочетании с функцией pthread_join: завершения потока тоже не всегда хочется ждать. Ответственной за формирование тайм-аутов ядра является функция timer_timeout(), представленная в следующем фрагменте.

```
#include <time.h>
extern int timer_timeout(
        clockid_t id,                   // тип системных часов
        int flags,                      // флаги, определяющие область действия тайм-аута
        const struct sigevent* notify,  // уведомление о событии
        const struct timespec* ntime,   // новое время тайм-аута
        struct timespec* otime );       // старое время тайм-аута
```

Аргумент id указывает тип системных часов. Стандарт POSIX определяет 3 типа системных часов:

- CLOCK_REALTIME – подстраиваемые системные часы для нормального режима работы процессора и режима работы при сниженном энергопотреблении. Необходимость подстройки часов обычно возникает при сетевых приложениях;
- CLOCK_SOFTIME – подобны CLOCK_REALTIME при нормальном режиме работы, не действуют в режиме сниженного энергопотребления;
- CLOCK_MONOTONIC – эти часы всегда инкрементируются с постоянной скоростью и не могут быть подстроены.

Аргумент flags определяет подмножество условий блокировки, на которое распространяется действие тайм-аута. Каждое условие задается битовой константой. Константы логически объединяются по «или». В табл. 2.8 представлены символические обозначения констант и их описания.

Аргумент notify является указателем на структуру, устанавливающую тип уведомления о наступлении тайм-аута. Рекомендуемые типы уведомлений представлены в табл. 2.9.

Таблица 2.8

Константа	Блокирующие состояния	Порождающие вызовы
_NTO_TIMEOUT_CONDVAR	STATE_CONDVAR.	SyncCondvarWait()
_NTO_TIMEOUT_JOIN	STATE_JOIN.	ThreadJoin()
_NTO_TIMEOUT_INTR	STATE_INTR.	InterruptWait()
_NTO_TIMEOUT_MUTEX	STATE_MUTEX.	SyncMutexLock()
_NTO_TIMEOUT_RECEIVE	STATE_RECEIVE.	MsgReceive()
_NTO_TIMEOUT_REPLY	STATE_REPLY.	MsgSend()
_NTO_TIMEOUT_SEM	STATE_SEM.	SyncSemWait()
_NTO_TIMEOUT_SEND	STATE_SEND.	MsgSend()
_NTO_TIMEOUT_SIGSUSPEND	STATE_SIGSUSPEND.	SignalSuspend()
_NTO_TIMEOUT_SIGWAITINFO	STATE_SIGWAITINFO.	SignalWaitinfo()

Только SIGEV_UNBLOCK гарантирует, что наступит деблокировка потока. Остальные виды уведомлений лишь информируют об истечении интервала тайм-аута. Если поле события имеет значение NULL, то это эквивалентно SIGEV_UNBLOCK. Деблокируемый по тайм-ауту вызов возвращает ошибку с кодом ETIMEDOUT.

Таблица 2.9

Варианты уведомлений	Описание
SIGEV_INTR	Прерывание
SIGEV_PULSE	Импульс
SIGEV_SIGNAL	Сигнал
SIGEV_SIGNAL_CODE	Сигнал реального времени
SIGEV_SIGNAL_THREAD	Сигнал к потоку
SIGEV_UNBLOCK	Деблокировка потока

Тайм-ауты запускаются по входу в одно из состояний блокировки, указанное в аргументе flags. Сбрасывается тайм-аут при возврате из любого системного вызова. Это означает, что необходимо заново «взводить» тайм-аут перед каждым системным вызовом, к которому он применяется. Если вместо параметра ntime указан NULL, то это предписывает ядру не блоки-

ровать поток в данном состоянии. Этот прием можно использовать для организации программного опроса, периодически проверяя, выполнено ли условие деблокировки. Если нет, то можно сделать что-то другое, но это не очень хорошая практика. Следующий фрагмент программы демонстрирует пример использования тайм-аута при ожидании завершения дочернего потока.

```
include <sys/neutrino.h>
#define SEC_NSEC 1000000000LL  // в 1 с содержится 1 000 000 000 нс
int main (void) // игнорировать аргументы
{
   uint64_t       timeout;
   struct sigevent event;
   int            rval;
   ...
// следующая макрокоманда устанавливает event.sigev_notify = SIGEV_UNBLOCK:
   SIGEV_UNBLOCK_INIT (&event);
   timeout = 10LL * SEC_NSEC;  // интервал тайм-аута равен 10 с
   TimerTimeout (CLOCK_REALTIME, _NTO_TIMEOUT_JOIN,
             &event, &timeout, NULL); // установить тайм-аут
   rval = pthread_join (thread_id, NULL);
   if (rval == ETIMEDOUT) {
      printf ("Поток %d работает больше 10 с!\n", thread_id);
   }
...
```

В этом фрагменте тайм-аут на ожидание завершения потока устанавливается равным 10 с. Уведомление SIGEV_UNBLOCK устанавливается с помощью макрокоманды. Если дочерний поток продолжает работать за пределами установленной временной границы, то в родительском потоке выполняется возврат из вызова pthread_join() с кодом ошибки ETIMEDOUT.

Следующий пример демонстрирует использование тайм-аута при обмене сообщениями.

```
event.sigev_notify = SIGEV_UNBLOCK; // прямая установка типа уведомления

timeout.tv_sec  = 10;
timeout.tv_nsec =  0;

timer_timeout( CLOCK_REALTIME,
           _NTO_TIMEOUT_SEND | _NTO_TIMEOUT_REPLY,
           &event, &timeout, NULL );
MsgSendv( coid, NULL, 0, NULL, 0 );
...
```

Тайм-аут длительностью в 10 с устанавливается перед вызовом передачи сообщения и действует на SEND-блокированное и REPLY-блокированное состояния передающего потока.

3. Механизмы взаимодействия программных потоков

Согласно стандарту POSIX основной формой взаимодействия программных потоков определена передача сообщений. Механизм обмена со-

общениями является базовым средством для реализации современной концепции проектирования прикладных программ в виде набора взаимодействующих процессов.

QNX была первой коммерческой системой, в которой механизм передачи сообщений использовался не только для прикладных программ, но и для объединения различных компонентов системных служб в единое целое. В 2.11 было дано описание механизма синхронизации потоков при передаче сообщений, в данном разделе будут детально представлены сервисные примитивы, поддерживающие эту форму взаимодействия между потоками.

Таблица 3.1

Формы взаимодействия	Область реализации
Обмен синхронными сообщениями	Ядро
Импульсы	Ядро
Сигналы	Ядро
Обмен асинхронными сообщениями	Ядро
POSIX очереди сообщений	Внешний процесс
Разделяемая память	Администратор процессов
Неименованные программные каналы (PIPEs)	Внешний процесс
Именованные программные каналы (FIFOs)	Внешний процесс

Кроме передачи сообщений в QNX Neutrino используются несколько других форм взаимодействия потоков (табл. 3.1). В большинстве случаев другие формы взаимодействия надстраиваются поверх механизма обмена сообщениями.

3.1. Синхронный обмен сообщениями

В QNX Neutrino программные потоки взаимодействуют не напрямую друг с другом, а через специальные объекты, называемые каналом (Channel) и соединением (Connection) (рис. 3.1). Поток сервера (Server), который желает получить сообщение, сначала создает канал; другой поток – клиент (Client), который желает послать сообщение к потоку сервера, должен создать соединение с каналом. Каналы и соединения после своего создания получают идентифицирующие номера – малые целые числа, которые берутся из пространства файловых дескрипторов. Последнее означает, что сообщение может быть послано через файловый дескриптор независимо от того, как он был получен.

С соединением связаны 2 идентификатора – coid на стороне клиента и scoid на стороне сервера. Для передачи сообщений используется только

идентификатор клиента – coid, идентификатор scoid – это внутренний идентификатор, используемый ядром для маршрутизации ответного сообщения от сервера к клиенту. В табл. 3.2 представлены системные функции, используемые для управления каналами и соединениями.

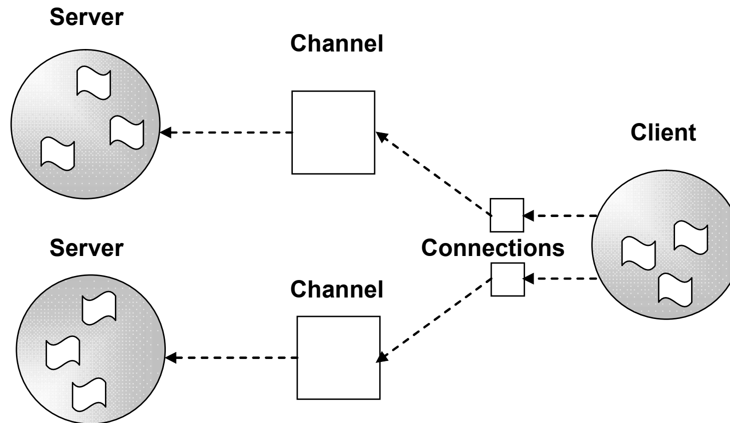

Рис. 3.1

По умолчанию приоритет потока, получающего сообщения через канал, устанавливается равным приоритету потока, пославшего сообщение. Это приоритетное наследование предотвращает инверсию приоритетов. Если сообщение поступает в канал и нет потока, который его может принять, то система приоритетно форсирует все потоки в процессе, которые в прошлом получали сообщение из этого канала (ассоциированные потоки). Это форсирование предотвращает приоритетную инверсию клиента, пославшего сообщение, в случае, когда все потоки непрерывно работают в интересах других клиентов, возможно с более низким приоритетом. Поток ассоциируется с каналом, когда он первый раз выдает вызов MsgReceivev(). Если поток работает с несколькими каналами, то он всегда ассоциируется с последним каналом, по которому он принял сообщение.

Таблица 3.2

Функция	Описание
ChannelCreate()	Создать канал для приема сообщений
ChannelDestroy()	Уничтожить канал
ConnectAttach()	Создать соединение для передачи сообщения
ConnectDetach()	Уничтожить соединение

Свойства канала определяются словом флагов, которое является единственным аргументом функции ChannelCreate(). Некоторые из канальных флагов изменяют поведение канала, другие требуют от ядра формирования уведомляющих импульсов в случае нарушения работы канала (использование импульсов для передачи информации между потоками рас-

смотрено в 3.5). В следующем фрагменте представлен набор флагов, которые используются при создании канала.

_NTO_CHF_COID_DISCONNECT – доставляет импульс к этому каналу по каждому связанному с ним соединению, когда канал уничтожается. Только один канал в процессе может иметь такой флаг установленным. Импульс характеризуется кодом (Pulse code) и значением (Pulse value). Для данного флага импульс имеет характеристики:

Pulse code: _PULSE_CODE_COIDDEATH.

Pulse value: идентификатор соединения клиента (coid), который был прикреплен к уничтоженному каналу.

_NTO_CHF_DISCONNECT – доставляет импульс к этому каналу, когда разорвано последнее из имевшихся соединений сервера с клиентами. Если процесс завершается без предварительного разрыва всех его соединений, ядро уничтожает соединения автоматически. Когда этот флаг установлен, сервер должен вызвать ConnectDetach(scoid), где scoid есть идентификатор серверного соединения в импульсном сообщении. Неудача приводит к тому, что номер идентификатора серверного соединения не может быть использован заново. В этой ситуации через некоторое время сервер может исчерпать приемлемые идентификаторы. Если этот флаг не установлен, ядро удаляет серверное соединение автоматически, делая его идентификатор приемлемым для повторного использования. Характеристики импульса:

Pulse code: _PULSE_CODE_DISCONNECT.

Pulse value: None.

_NTO_CHF_FIXED_PRIORITY – подавляет приоритетное наследование при получении сообщений. Если флаг установлен, получающий поток не будет повышать свой приоритет до приоритета потока, посылающего сообщение.

_NTO_CHF_REPLY_LEN – требует от ядра, чтобы длина ответного сообщения была установлена в поле dstmsglen в структуре _msg_info (описание структуры см. далее). Это поле приемлемо, если данный флаг был установлен при создании канала.

_NTO_CHF_SENDER_LEN – требует от ядра, чтобы длина клиентского сообщения была установлена в поле srcmsglen структуры _msg_info. Это поле приемлемо, если данный флаг был установлен при создании канала.

_NTO_CHF_THREAD_DEATH – ядро посылает импульс всякий раз, когда блокированный на этом канале поток завершается. Только один канал серверного процесса может иметь этот флаг установленным. Характеристики импульса:

Pulse code: _PULSE_CODE_THREADDEATH.

Pulse value: Thread ID – идентификатор потока (tid).

_NTO_CHF_UNBLOCK – ядро посылает импульс всякий раз, когда поток клиента пытается разблокироваться. Этот флаг скажет ядру: «Сообщи мне импульсом, когда клиент попробует разблокироваться, но не позволяй ему этого делать! Я разблокирую его сам». Характеристики импульса:

Pulse code: _PULSE_CODE_UNBLOCK.

Pulse value: Receive ID (rcvid).

С каналом связаны 3 очереди (рис. 3.2):

- очередь для потоков, ожидающих сообщение;
- очередь для потоков, которые послали сообщения, но они еще не приняты;
- очередь для потоков, которые послали сообщения. Эти сообщения приняты, но ответы еще не переданы.

Службы обмена сообщениями в QNX Neutrino копируют сообщение непосредственно из адресного пространства одного потока в адресное про-

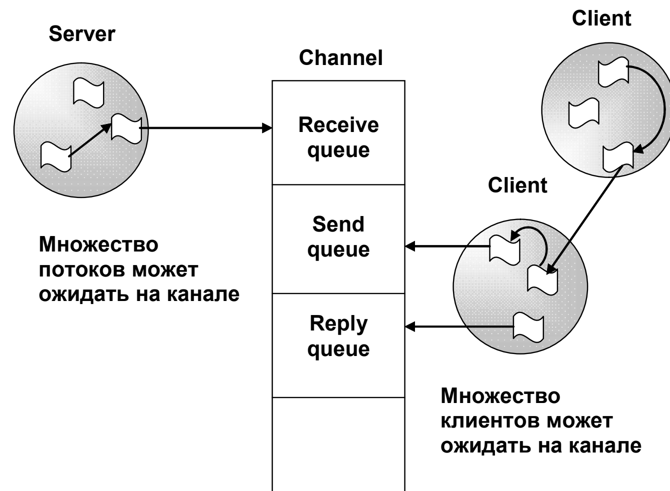

Рис. 3.2

странство другого без промежуточной буферизации. Скорость обмена сообщениями определяется производительностью памяти и используемым оборудованием. Следующий программный фрагмент демонстрирует пример использования примитивов передачи сообщений.

Пример. Обмен сообщениями.

```
#include <pthread.h>
#include <stdlib.h>
#include <string.h>
#include <sys/neutrino.h>
int chid;  // идентификатор канала
/*********************************************************************************/
void  *server() // поток сервера
{
int rcvd;  // идентификатор принятого сообщения
char receive_buf[25];    // буфер для приема сообщения
char reply_buf[25];      // буфер для приема ответа
chid=ChannelCreate(0); // создание канала
sleep(1);
rcvd=MsgReceive(chid, &receive_buf, sizeof(receive_buf), NULL); // прием сообщения
printf ("Server thread: message <%s> has received \n", &receive_buf);
strcpy(reply_buf, "Strong answer from Server");
MsgReply(rcvd, 1500052, &reply_buf, sizeof(reply_buf)); // передача ответа
ChannelDestroy(chid);  // уничтожить канал
pthread_exit(NULL);
}
/*********************************************************************************/
void  *client() // поток клиента
{
int coid;                 // идентификатор соединения
int status;       // статус ответного сообщения ( в данном случае 1500052)
pid_t PID;
char send_buf[25];        // буфер для передаваемого сообщения
char reply_buf[25];       // буфер для ответного сообщения
PID=getpid();             // получить программный идентификатор процесса
coid=ConnectAttach(chid,PID,1,0,0);    // создать соединение
strcpy(send_buf, "It is very simple example");
```

```
/* передача сообщения */
status=MsgSend(coid,&send_buf, sizeof(send_buf), &reply_buf, sizeof(reply_buf));
printf ("Client thread: message <%s> has received \n", &reply_buf);
ConnectDetach(coid);   // уничтожить соединение
pthread_exit(NULL);
}
/*****************************************************************************/
int  main() { // главный поток
pthread_t server_tid, client_tid;
pthread_create(&server_tid, NULL, &server, NULL); // создать поток сервера
pthread_create(&client_tid, NULL, & client, NULL);       // создать поток клиента
pthread_join(client_tid,NULL);
return 0;
}
```

Последним параметром функции MsgReceive() является указатель на структуру _msg_info. Эта структура позволяет серверу получить добавочную информацию о клиенте, пославшем сообщение. Структура имеет следующий формат:

```
struct _msg_info {
        uint32_t        nd;             /* дескриптор сетевого узла клиента */
        uint32_t        srcnd;          /* дескриптор сетевого узла сервера */
        pid_t           pid;            /* pid клиента */
        int32_t         tid;            /* tid клиента */
        int32_t         chid;           /* идентификатор канала */
        int32_t         scoid;          /* идентификатор серверного соединения* /
        int32_t         coid;           /* идентификатор клиентского соединения */
        int32_t         msglen;         /* число полученных байт */
        int32_t         srcmsglen;      /* длина посланного сообщения в байтах */
        int32_t         dstmsglen;      /* длина буфера клиента для ответа */
        int16_t         priority;       /* приоритет посылающего потока */
        int16_t         flags;          /* флаги */
        uint32_t        reserved;       /* зарезервировано */
};
```

Поле flags может содержать следующий битовый флаг:

_NTO_MI_UNBLOCK_REQ – клиент пытается выполнить деблокировку Send-блокированного состояния, но не может этого сделать, поскольку для канала установлен флаг _NTO_CHF_UNBLOCK.

Если в качестве последнего параметра в вызове MsgReceive() указано NULL, то информацию о клиенте можно получить позже с помощью вызова MsgInfo() – т. е. она не теряется. Функция MsgReceive() возвращает идентификатор отправителя (rcvd) – это целое число, действующее как «жетон», по которому возвращается ответное сообщение. Функция MsgReply() принимает в качестве параметров идентификатор отправителя, код возврата, указатель на ответное сообщение и размер этого сообщения. Код возврата указывает, какой код статуса должна возвратить функция MsgSend() на стороне клиента. Ответное сообщение может иметь нулевую длину. Единственная цель ответа – разблокировать клиента. Как вариант вместо функции MsgReply() можно использовать функцию MsgError(), которая не передает ответное сообщение, а возвращает только код ошибки (в глобальной переменной errno). После вызова функции MsgReply() идентификатор отправи-

теля теряет смысл (за исключением одного частного случая с применением функции MsgDeliverEvent(), которую рассмотрим в 3.7).

3.2. Векторные сообщения

В сервисном наборе существуют примитивы, поддерживающие передачу сообщений, состоящих из нескольких частей. В этом случае сообщение, доставляемое от одного адресного пространства к другому, не требует непрерывного буфера. Вместо этого как передающий, так и принимающий потоки могут специфицировать векторную таблицу, которая указывает, в каких фрагментах памяти размещаются части сообщения (рис. 3.3).

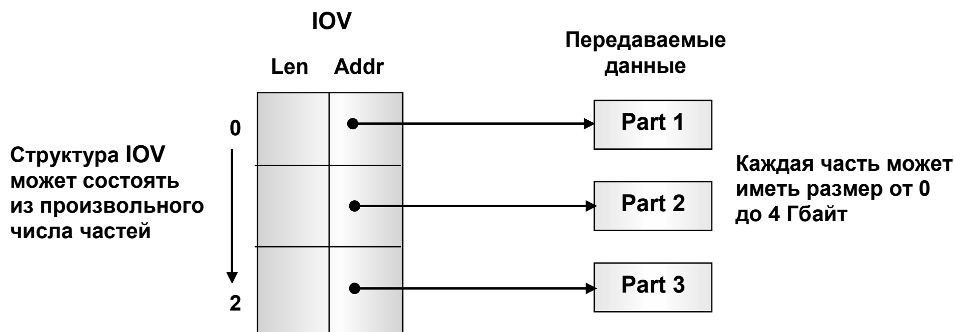

Рис. 3.3

Заметим, что размеры различных частей сообщения могут отличаться для передающего и приемного потоков. Более того, примитивы обмена позволяют принимать векторное сообщение как неделимое и, наоборот, при приеме разделять одиночное сообщение на составные части. В табл. 3.3 перечислены примитивы для обмена векторными сообщениями.

Таблица 3.3

Функция	Посылаемое сообщение	Принимаемое сообщение
MsgSend()	Одиночное	Одиночное
MsgSendsv()	Одиночное	Векторное
MsgSendvs()	Векторное	Одиночное
MsgSendv()	Векторное	Векторное

Функции векторного обмена подобны MsqSend() и MsgReceive(), но в поле аргументов векторных вызовов вместо буфера указывается массив структур IOV с полями Len и Addr (длина и размещение составных частей векторного сообщения соответственно). Вместо размера буфера указывается число элементов в массиве IOV.

3.3. Многошаговый обмен сообщениями

В операционной системе при обмене сообщениями между ее компонентами (например, при обмене с файловой системой) используются сооб-

щения предопределенного формата. Для этого в сообщении выделяется заголовок (обычно длиной 12 байт), который содержит сведения о том, что находится внутри сообщения. Например, подобный заголовок автоматически добавляет системный вызов write() библиотеки языка Си. Компонент операционной системы должен уметь сначала прочитать заголовок, а затем в зависимости от полученной информации принять решение о том, как поступить с остальными частями сообщения. В QNX Neutrino для этой цели реализовано несколько дополнительных функций, представленных в табл. 3.4.

Таблица 3.4

Для векторных сообщений	**Для одиночных сообщений**
MsgReadv()	MsgRead()
MsgWritev()	MsgWrite()

Эти функции имеют дополнительный аргумент offset (смещение) и используются следующим образом:

1. Клиент передает полное сообщение, имеющее заголовок.

2. Сначала сервер принимает сообщение с помощью MsgReceive(), при этом длина приемного буфера устанавливается равной размеру передаваемого заголовка.

3. Клиент переходит в Reply-блокированное состояние, а сервер тем временем анализирует заголовок сообщения. Заголовок содержит информацию о размещении составных частей сообщения.

4. Сервер считывает сообщение с помощью нескольких вызовов функции MsgRead(), в качестве аргументов функции указывается идентификатор отправителя полученного сообщения (возвращаемый функцией MsgReceive()), адрес и размер буфера, в который следует записать полученные данные, и смещение offset, с которого надо начинать считывать данные из буфера клиента. Клиент все это время остается Reply-блокированным.

5. Сервер подготавливает ответное сообщение. Он может его отправить единовременно, вызвав MsgReply(), или может записать ответ порциями в буфер клиента, используя вызов MsgWrite() и смещения offset. Но в этом случае, записав данные, сервер должен для разблокировки клиента обязательно вызвать MsgReply() или MsgError() с нулевым размером буфера.

3.4. Локализация сервера синхронного канала

Для того чтобы передать сообщение от клиента к серверу, клиентский поток должен создать соединение с каналом сервера, а для этого ему необходимо знать номер канала, созданного сервером. Когда передающий и принимающий потоки находятся в одном процессе, эта задача не представляет особого труда: достаточно объявить номер канала как глобальную переменную процесса, и она будет доступна всем процессам в потоке. Ситуация резко усложняется, когда потоки находятся в разных не связанных между собой процессах. Существует несколько способов передать номер канала и/или непосредственно образовать соединение с каналом. Наиболее радикальный способ – построить серверный процесс по канонам администратора ресурсов. В этом случае администратор регистрирует некоторую область полномочий в пространстве имен, и номер соединения может быть получен при выполнении стандартной библиотечной функции open(). При вызове этой функции в качестве аргумента указывается имя из зарегистрированной области полномочий. Разработка администратора ресурсов в полном объеме – сложная задача и имеет смысл только для достаточно больших приложений. Далее будет рассмотрена более простая альтернатива, основанная на регистрации имен процессов (так называемый неполный администратор).

Для работы с именами процессов используются системные вызовы, представленные в табл. 3.5.

Таблица 3.5

Функция	Описание
name_attach()	Регистрировать имя процесса и создать канал
name_open()	Локализовать сервер и получить идентификатор соединения
name_close()	Уничтожить соединение с каналом сервера
name_detach()	Удалить имя процесса из пространства имен и созданный канал

Процесс сервера, который желает получать сообщения, регистрирует символическое имя. С этой целью используется системный вызов name_attach().

```
name_attach_t * name_attach( dispatch_t * dpp,      // dispatch-драйвер
                    const char * path,              // имя процесса
                    unsigned flags );               // флаги
```

Аргументы:

- dpp – указывает на функцию dispatch-драйвер, но обычно имеет значение NULL, в этом случае необходимые структуры и функции строятся автоматически;
- path – имя процесса, которое необходимо зарегистрировать. Список локальных зарегистрированных имен процессов можно посмотреть в директории ./dev/name/local, список глобальных зарегистрированных имен доступен в директории ./dev/name/global. Локальные имена известны в пределах одного узла, а глобальные – в пределах всей сети;

• flags – флаги. Существует только один флаг NAME_FLAG_ATTACH_GLOBAL – регистрировать глобальное имя. В противном случае имя регистрируется как локальное.

Вызов возвращает указатель к структуре name_attach, которая имеет вид:

```
typedef struct _name_attach {
   dispatch_t* dpp;       // dispatch-драйвер
   int     chid;          // идентификатор канала
   int     mntid;         // идентификатор монтирования
   int     zero[2];       // нулевые значения
} name_attach_t;
```

При выполнении функции создается канал посредством внутреннего вызова ChannelCreate() с атрибутами: _NTO_CHF_UNBLOCK, _NTO_CHF_DISCONNECT, _NTO_CHF_COID_DISCONNECT. В результате сервер может получать сообщения и импульсы. Набор флагов устанавливает, что сервер получит служебные импульсы при попытке деблокировки клиента и при разрыве соединений. Ядро поддерживает взаимно-однозначное соответствие между именем процесса и его программным идентификатором (pid), поэтому именованный процесс не может иметь копий в оперативной памяти. На стороне клиента используется функция name_open().

```
int name_open( const char * name,      // имя сервера
            int flags );               // флаги
```

Аргументы:

• name – имя сервера, которое необходимо открыть;

• flags – единственный используемый флаг NANE_FLAG_ATTACH_GLOBAL – искать глобальное имя. При нулевом слове флагов – искать локальное имя.

Эта функция открывает имя для серверного соединения, при этом ядро с помощью вызова MsgSend() посылает серверу служебное сообщение _IO_CONNECT с 4-байтовым заголовком. Серверное приложение должно быть готовым принять это сообщение и ответить вызовом MsgReply(). При удачном выполнении функции name_open() возвращается идентификатор соединения (coid), а при неудачном значение –1.

3.5. Импульсы

При передаче сообщений с помощью вызова MsgSend() передающий поток блокируется. Однако возможны ситуации, когда отправитель не может позволить себе находиться в блокированном состоянии. Механизм, который обеспечивает отправку коротких сообщений предопределенного формата без блокирования отправителя, называют импульсным обменом. Импульс (Pulse) – это миниатюрное сообщение, которое несет 40 бит полезной информации: 8 бит кода (Pulse code) и 32 бит данных (Pulse value). Посылающий поток задает приоритет импульса, импульсы с одинаковым приоритетом ставятся в очередь. Импульс может быть принят функцией MsgReceive() как и обычное синхронное сообщение, но более удобной является функция MsgReceivePulse(). При приеме импульса с помощью

MsgReceive() функцией возвращается нулевой идентификатор отправителя (rcvd), что и позволяет отличить импульс от обычного сообщения. Формат принимаемого сообщения определен структурой _pulse:

```
struct _pulse {
   uint16_t          type;
   uint16_t          subtype;
   int8_t            code;     // код импульса
   uint8_t          zero[3];
   union sigval      value;    // значение импульса
   int32_t          scoid;     // идентификатор серверного соединения
};
```

Импульс погружен в принятое сообщение. Поля type и subtype равны нулю (еще один признак импульсного сообщения). Поля code и value принадлежат импульсу. Код импульса обычно указывает причину, по которой импульс был отправлен. Ядро резервирует отрицательные значения кода, оставляя 127 положительных для программиста. Другие поля структуры _pulse пользователем не настраиваются. Резервируемые системой QNX значения кода и соответствующие им значения импульсов представлены в табл. 3.6.

Таблица 3.6

Code	Value	Назначение
_PULSE_CODE_UNBLOCK	Идентификатор отправителя (rcvd)	Служебные сообщения канала
_PULSE_CODE_DISCONNECT	–	Служебные сообщения канала
_PULSE_CODE_THREADDEATH	Идентификатор завершенного потока	Служебные сообщения канала
_PULSE_CODE_COIDDEATH	Идентификатор соединения	Служебные сообщения канала
_PULSE_CODE_NET_ACK, _PULSE_CODE_NET_UNBLOCK, _PULSE_CODE_NET_DETACH	–	Резервированы для сетевых взаимодействий

В отличие от MsgReceive() функция MsgReceivePulse() выполняет прием только импульсов. При удачном приеме возвращается нуль, при неудачном – код ошибки. Набор аргументов для этой функции такой же, как и для MsgReceive(). Отправить импульс можно с помощью функции:

```
int MsgSendPulse ( int coid,      // идентификатор соединения
                   int priority,  // приоритет импульса
                   int code,      // 8-битный код
                   int value );   // 32-битное значение
```

3.6. Сигналы

Сигналы являются асинхронными сообщениями предопределенного формата и используются для передачи уведомлений о происшедших событиях. Формат сигналов аналогичен формату импульсов (1 байт кода + 4

байт данных), и аналогично импульсам сигналы могут ставиться в очередь. Однако в отличие от импульса сигналы способны прерывать исполнение программного потока, и в этом качестве они подобны аппаратным прерываниям. Системные вызовы для обмена сигналами представлены в табл. 3.7.

Таблица 3.7

Вызовы микроядра	POSIX-вызовы	Описание
SignalKill()	kill(), pthread_kill(), raise(), sigqueue()	Послать сигнал к потоку, процессу или группе процессов
SignalAction()	sigaction(), signal()	Определить действие, которое нужно выполнить при приеме сигнала
SignalProcmask()	sigprocmask(), pthread_sigmask()	Изменить маску блокировки сигналов для потока
SignalSuspend()	sigsuspend(), pause()	Блокировать поток, пока сигнал обрабатывается
SignalWaitinfo()	sigwaitinfo()	Ожидать сигнал и возвратить информацию при приеме сигнала

Спецификация POSIX первоначально определяла порядок обработки сигналов только для процессов. Процесс может определить следующие варианты поведения при получении сигнала:

1. Игнорировать сигнал. Поступивший сигнал не обрабатывается и теряется. Необходимо отметить, что 3 сигнала не могут быть проигнорированы – это SIGSTOP, SIGCONT, SIGTERM (остановить процесс, продолжить процесс, завершить процесс).
2. Блокировать сигнал (другой равноправно используемый термин – это «маскировать сигнал»). Ядро поддерживает для потока атрибут, называемый маской сигналов. Действие сигнальной маски подобно действию регистра маски контроллера прерываний. Если сигнал замаскирован, то он не может быть обработан, но он не теряется и обрабатывается после снятия маски.
3. Вызвать собственный обработчик сигнала, который был предварительно зарегистрирован.
4. Использовать обработчик по умолчанию. Обычно обработчик по умолчанию уничтожает процесс.

Для многопоточных приложений стандарт POSIX в дальнейшем дополнен следующими правилами:

- Действие сигнала, направленного к потоку, распространяется на весь процесс.
- Каждый поток имеет свою сигнальную маску.

- Неигнорируемый сигнал, предназначенный для какого-либо потока, доставляется только этому потоку.
- Неигнорируемый сигнал, предназначенный для процесса, передается первому потоку, который не имеет блокировки для сигнала.
- Если все потоки имеют блокировки для сигнала, сигнал будет поставлен в очередь к процессу и будет находиться там до тех пор, пока какой-либо поток не проигнорирует или не деблокирует сигнал. При игнорировании сигнал будет удален. При деблокировании сигнал будет направлен к потоку, который его деблокировал.

На рис. 3.4 представлена схема доставки сигнала в многопоточном приложении. При доставке сигнала прежде всего проверяется, не принадлежит ли он множеству игнорируемых сигналов (Signal ignore), и если это

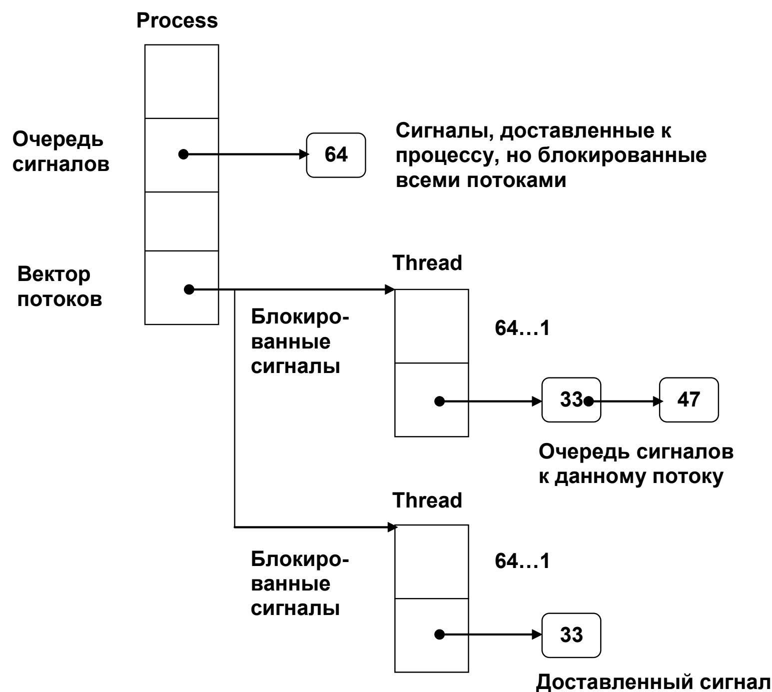

Рис. 3.4

так, то он отбрасывается. Далее просматривается таблица потоков и ищется поток, который не блокирует сигнал. Стандартная практика для большинства многопоточных процессов – это маскировать сигнал во всех потоках, кроме одного, который ответствен за его обработку. Для увеличения эффективности процедуры доставки сигнала ядро будет кэшировать последний поток, принявший сигнал, и будет всегда пытаться доставить сигнал вначале к этому потоку. Если сигнал не принадлежит множеству игнорируемых сигналов, но блокирован всеми потоками, то он ставится в очередь к процессу (Process Signal queue). Сигналы, которые направлены к

конкретному потоку, при необходимости ставятся в очередь на обработку (Thread Signal queue). Такие сигналы называются захваченными.

POSIX разделяет сигналы на две группы: стандартные сигналы (включая 32 UNIX-сигнала) и сигналы реального времени. Механизм очередности поддерживается только для сигналов реального времени. При отсутствии очередности новый сигнал замещает висящий необработанный сигнал, который теряется. Для стандартных сигналов не существуют также понятия кода и значения сигнала. В отличие от POSIX QNX Neutrino поддерживает опциональную очередность для всех сигналов, и каждый сигнал имеет 8-битный код Code и 32-битное значение Value. Сигналы с положительным значением кода используются для системных целей, а для пользователя определен диапазон кодов от –128 до 0. Всего поддерживается 64 сигнала. Все сигналы пронумерованы, многие из них имеют символическое обозначение, отражающее системное назначение сигнала. Специализация сигналов в зависимости от номера показана в табл. 3.8.

Таблица 3.8

Номер сигнала	**Назначение**
1...57	57 POSIX-сигналов (включая традиционные UNIX-сигналы)
41...56	16 POSIX-сигналов реального времени (от SIGRTMIN до SIGRTMAX)
57...64	8 сигналов для специальных целей QNX Neutrino

Номер сигнала определяет приоритет его доставки: чем меньше номер сигнала, тем выше его приоритет. Восемь сигналов для специальных целей не могут быть игнорированы или захвачены. Любая попытка вызова функции signal() или sigaction(), или SignalAction() будет терпеть неудачу. Более того, эти сигналы всегда блокированы, попытка деблокирования через sigpromask() или SignalProcmask() будет игнорироваться. Специальные сигналы всегда упорядочиваются в очередь и могут быть приняты только блокирующей функцией sigwaitinfo(). Эти сигналы используются для передачи уведомления клиенту с помощью вызова MsgDeliverEvent() (вызов будет рассмотрен далее), а также для поддержки механизма событий в графической системе Photon и уведомлений ввода/вывода в множественных серверах.

При обработке сигнала с номером signo программа обработки маскирует сигнал signo, тем самым предотвращая вложенность обработки для сигналов того же самого номера. Когда выполняется возврат из обработчика, прежняя маска восстанавливается и любые висящие и немаскированные сигналы становятся активными. Реакция на сигнал должна быть задана

предварительно, вызовом функций SignalAction() (QNX) или sigaction() (POSIX). Ниже представлен синтаксис функции sigaction().

```
int sigaction( int sig,
               const struct sigaction * act,
               struct sigaction * oact );
```

Аргументы:

- sig – номер сигнала (как правило, задается символической константой);
- act – NULL или указатель к структуре sigaction, которая специфицирует новую реакцию на сигнал;
- oact – NULL или указатель, по которому должна быть сохранена прежняя реакция сигнала, определенная структурой sigaction.

Структура sigaction содержит следующие поля:

- void (*sa_handler)(); адрес функции обработчика для сигналов без поддержки очередности;
- void (*sa_sigaction)(int signo, siginfo_t* info, void* other); адрес функции обработчика для сигналов с поддержкой очередности;
- sigset_t sa_mask – набор сигналов, которые должны быть дополнительно маскированы при выполнении обработчика;
- int sa_flags – специальные флаги, определяющие поведение сигнала:

- SA_NOCLDSTOP – используется только для сигнала SIGCHLD и запрещает передачу этого сигнала дочерним процессом к родительскому процессу, если дочерний процесс остановлен по сигналу SIGSTOP,

- SA_SIGINFO – определяет, что для данного сигнала будет поддерживаться очередь. По умолчанию очередь для всех сигналов не поддерживается.

Функции обработчиков sa_handler и sa_sigaction выполняются как компоненты union. Оба обработчика разделяют одно и то же адресное пространство и отличаются только прототипами. Через эти поля можно также определить стандартные обработчики, используя символические константы:

- SIG_DFL – определяет следующие действия для специфицированных сигналов:

- сигналы SIGCHLD, SIGIO, SIGURG и SIGWINCH игнорируются,
- сигнал SIGSTOP – останавливает процесс,
- сигнал SIGCONT – продолжает программу,
- все остальные сигналы убивают процесс;

- SIG_IGN – игнорирует специфицированный сигнал. Если специфицирован сигнал SIGCHLD, то дочерний процесс не переходит в состояние зомби, а родительский не может получить от него код завершения, используя вызовы wait() или waitpid().

Возвращаемая структура siginfo_t функции обработчика (*sa_sigaction) содержит следующие поля:

- int si_signo – номер сигнала, который должен совпадать с аргументом signo;
- int si_code – код сигнала. Ядро резервирует следующие значения кодов:
  – SI_USER – сигнал сгенерирован kill() функцией,
  – SI_QUEUE – сигнал сгенерирован sigqueue() функцией,
  – SI_TIMER – сигнал сгенерирован таймером,
  – SI_ASYNCIO – сигнал сгенерирован асинхронной операцией ввода/вывода,
  – SI_MESGQ – сигнал, сгенерированный POSIX-очередью сообщений.

Обработчик сигналов может завершаться вызовами longjmp() или siglongjmp(). В первом случае после возврата из обработчика сигнал остается маскированным. Во втором случае восстанавливается маска, предварительно сохраненная вызовом sigsetjmp().

Сигнал можно сгенерировать, используя POSIX-вызов kill(), синтаксис которого представлен в следующем фрагменте.

```
int kill( pid_t pid,          // идентификатор процесса или группы процессов
        int sig );            // номер сигнала
```

Если pid > 0, то он указывает идентификатор одного процесса.

Если pid = 0, сигнал посылается к группе процессов, которые имеют групповой идентификатор, совпадающий с групповым идентификатором процесса, посылающего сигнал.

Если pid < 0, то сигнал посылается к группе процессов, для которых gid=abs(pid).

Для того чтобы процесс мог послать сигнал другому процессу, его эффективные идентификаторы UID и GID должны либо совпадать с идентификаторами принимающего процесса, либо быть нулевыми. В последнем случае посылающий процесс принадлежит привилегированному пользователю. QNX-вызов SignalKill() расширяет возможности стандарта POSIX.

```
int SignalKill( uint32_t nd,      // дескриптор узла сети или 0 для локального узла
            pid_t pid,            // pid процесса
            int tid,              // tid – идентификатор потока
            int signo,            // номер сигнала
            int code,             // код сигнала
            int value );          // значение сигнала
```

С помощью этого вызова сигналы можно пересылать по сети и направлять к конкретному потоку в процессе. Согласно стандарту POSIX, если поток, принявший сигнал, завершается, то завершаются все потоки процесса. Это условие действует по умолчанию, но его можно изменить, определив при создании потока флаг PTHREAD_MULTISIG_DISALLOW. При установленном флаге будет завершаться только поток, принявший сигнал, остальные потоки сохраняют свою активность. В следующем фрагменте приведен пример использования сигналов.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

int main( void )
 {
  extern void handler();
  struct sigaction act;
  sigset_t set;
  sigemptyset( &set );
  sigaddset( &set, SIGUSR1 );
  sigaddset( &set, SIGUSR2 );

  act.sa_flags = 0;
  act.sa_mask = set;             // установить маску сигналов при вызове обработчика
  act.sa_handler = &handler;     // установить обработчик сигнала
  sigaction( SIGUSR1, &act, NULL );   // связать обработчик с сигналом SIGUSR1

  kill( getpid(), SIGUSR1 ); // послать сигнал SIGUSR1

  /* программа будет завершаться по SIGUSR2 */
  return EXIT_SUCCESS;
 }
```

```
void handler( signo ) // обработчик сигнала SIGUSR1
 {
   static int first = 1;
   if( first ) {
     first = 0;
     kill( getpid(), SIGUSR1 );  // маскированный сигнал
     kill( getpid(), SIGUSR2 );  // маскированный сигнал
   }
 }
```

В данном примере сигнал SIGUSR1, посланный из потока main(), вызывает обработчик handler(). Обработчик генерирует еще 2 сигнала SIGUSR1 и SIGUSR2, однако эти сигналы маскированы, до тех пор пока не произойдет возврат к main(). После возврата вновь вызывается обработчик для висящего сигнала SIGUSR1. Сигнал обрабатывается, но для висящего сигнала SIGUSR2 обработчик не определен, что приводит к завершению процесса.

Таблица 3.9

Функция	Назначение
sigemptyset()	Очистить список обрабатываемых сигналов
sigaddset()	Добавить сигнал в список
sigdelset()	Удалить сигнал из списка
sigfillset()	Добавить в список все сигналы
sigismember()	Проверить наличие сигнала в списке

В примере использованы сервисные функции для формирования списка обрабатываемых сигналов. Полный набор сервисных функций приведен в табл. 3.9.

3.7. Иерархический принцип обмена синхронными сообщениями

Используемый принцип синхронизации сообщений (через вызовы MsgSend(), MsgReceive(), MsgReply()) заставляет следовать строгой иерархии при организации взаимодействия между потоками. В частности, это означает, что 2 потока никогда не должны посылать сообщения друг другу, так как это с большой вероятностью приводит к взаимной блокировке. В таком состоянии оба потока будут ожидать друг от друга ответ на посланные сообщения, но не смогут дать его, поскольку оба блокированы. Для исключения подобной ситуации обмен должен быть организован так, чтобы каждый поток занимал свой уровень иерархии и все потоки данного уровня посылали сообщения только потокам более высокого уровня, а не своего или низшего (рис. 3.5).

Способ назначения уровней иерархии заключается в том, чтобы в цепочке доступа разместить наиболее удаленного клиента на самом нижнем

уровне иерархии. При этом передачи MsgSend() будут направлены от клиента нижнего уровня к серверу верхнего уровня, а ответы на сообщения будут иметь встречное направление.

Однако существуют ситуации, когда приходится нарушать естественное направление передач, например, когда для формирования ответа серверу требуется значительное время, а клиент, пославший сообщение, не может позволить себе блокироваться на длительное время из-за опасности потери событий.

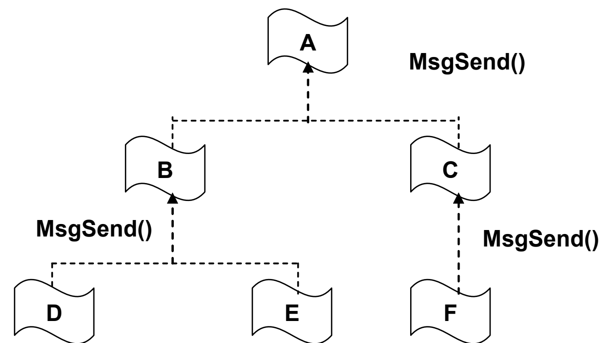

Рис. 3.5

Такому клиенту необходимо, чтобы сервер ответил сразу чем-то вроде «заказ принят». Получив такой «формальный ответ», клиент способен продолжать работу, а тем временем сервер отрабатывает запрос и после выполнения «заказа» должен как-то сказать клиенту, что «заказ выполнен». Использовать MsgSend() сервер не может из-за опасности взаимной блокировки. Для разрешения подобной ситуации эффективно используется механизм доставки событий с помощью системного вызова MsgDeliverEvent(). Эта составная операция работает следующим образом (рис. 3.6).

1. Клиент создает структуру типа struct sigevent, в которой указывает, какое уведомление о завершении работы сервером он желает получить.

2. Клиент посылает сообщение, которое содержит структуру sigevent (как заголовок) и данные для обработки сервером.

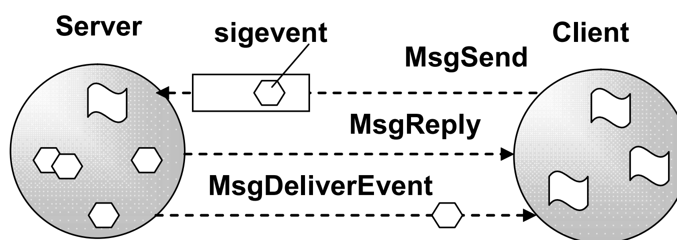

Рис. 3.6

3. Сервер принимает сообщение, сохраняет структуру sigevent и идентификатор отправителя rcvd и немедленно отвечает клиенту.

4. Теперь клиент деблокирован и выполняется – так же как и сервер.

Когда сервер завершает работу, он использует функцию MsgDeliverEvent(), чтобы сообщить об этом клиенту.

Функция доставки события имеет следующий синтаксис:

```
int MsgDeliverEvent( int rcvd,                    // идентификатор отправителя
          const struct sigevent* event );    // уведомление о событии
```

Сервер никогда не изменяет и даже не читает содержимое структуры sigevent. Уведомление о событии формирует ядро, анализируя аргумент event в вызове MsgDeliverEvent(). Это позволяет серверу доставлять события вне зависимости от выбранного клиентом типа. Тип события указывается в поле event.sigev_notify структуры sigevent. Прочие поля структуры подчинены типу события, все возможные варианты заполнения полей структуры sigevent представлены в табл. 3.10.

Таблица 3.10

sigev_notify	sigev_signo	sigev_coid	sigev_priority	sigev_code	sigev_value
SIGEV_INTR	–	–	–	–	–
SIGEV_NONE	–	–	–	–	–
SIGEV_PULSE	–	Connection	Priority	Code	Value
SIGEV_SIGNAL	Signal	–	–	–	–
SIGEV_SIGNAL_CODE	Signal	–	–	Code	Value
SIGEV_SIGNAL_THREAD	Signal	–	–	Code	Value
SIGEV_THREAD	–	–	–	–	Value
SIGEV_UNBLOCK	–	–	–	–	–

Пустые позиции указывают, что соответствующие поля структуры не используются для данного типа уведомления. Символические обозначения типа события расшифровываются следующим образом:

- SIGEV_INTR – уведомить прерыванием (предназначено для обработчиков прерываний и не используется для вызова MsgDeliverEvent());
- SIGEV_NONE – не посылать уведомления (возможно, клиенту оно не требуется!);
- SIGEV_PULSE – уведомить импульсом с кодом Code и значением Value. Импульс посылается клиенту по соединению Connection с приоритетом Priority;
- SIGEV_SIGNAL – уведомить сигналом с номером Signal;
- SIGEV_SIGNAL_CODE – уведомить сигналом реального времени с номером Signal, кодом Code и значением Value;
- SIGEV_SIGNAL_THREAD – послать сигнал реального времени к специфицированному потоку;
- SIGEV_THREAD – создать новый поток и передать ему параметр Value;
- SIGEV_UNBLOCK – деблокировать поток (используется для формирования реакции на тайм-ауты ядра).

Аргумент rcvd в вызове MsgDeliverEvent() определяет доставочный адрес для пересылки уведомления. Сервер хранит значение этого идентификатора после вызова функции MsgReceive().

3.8. Асинхронный обмен сообщениями

Механизм асинхронного обмена определен как экспериментальный в версии 6.3.0. QNX Neutrino и не входит в стандарт POSIX. Асинхронные сообщения являются коммуникационной моделью, которая основана на способе передачи данных с промежуточным хранением. В отличие от механизма синхронных сообщений этот способ обеспечивает доставку сообщений, даже если получатель временно отсутствует, занят или недоступен. При асинхронном обмене посылающий поток не нуждается в ответе от потока получателя. Это обеспечивает большую гибкость и расширяемость, так как передающие и принимающие потоки теперь не связаны в пары и не требуют жесткой взаимной синхронизации.

В системах реального времени нередко встречаются ситуации, когда клиент не желает ждать в блокирующем состоянии, пока сервер закончит обслуживать его требование. Вместо этого он может улучшить свою эффективность, выполняя в это время некоторые задачи. Множество асинхронных сообщений может быть также послано клиентом в пределах этого периода. Буферизация асинхронных сообщений и пересылка их в виде пакетов позволяют снизить частоту обращения к ядру и улучшить тем самым системную эффективность.

Механизм передачи асинхронных сообщений обеспечивает событий-

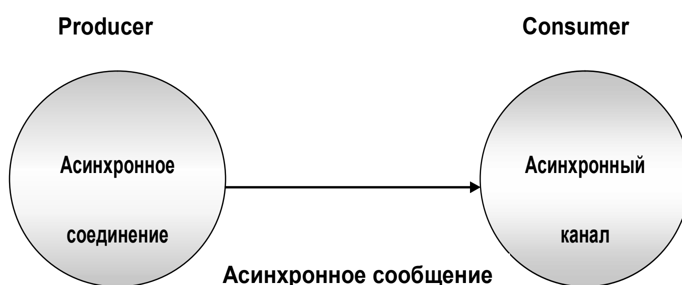

Рис. 3.7

ную системную модель, которая хорошо дополняет традиционный синхронный механизм передачи сообщений. Ее главным свойством является высокая производительность и отсутствие блокировок при передаче сообщений. Эта системная модель позволяет разработать системы, в которых QNX Neutrino сможет своевременно реагировать на внешние и внутренние асинхронно генерируемые события, а также обеспечивает удобный и эф-

фективный механизм доставки асинхронных событий и связанных с ними данных к системным компонентам.

Рис. 3.7 иллюстрирует прикладной процесс (Producer), который собирает данные от некоторых входных устройств (сенсоров), буферизует их и посылает к другому прикладному процессу (Consumer) для обработки. Базовый принцип при асинхронной передаче сообщений заключается в пакетной доставке сообщений, которая увеличивает пропускную способность и снижает накладные расходы. Данные посылаются как сообщения с определенными границами.

Таблица 3.11

Функция	Назначение
asyncmsg_channel_create()	Создать канал для передачи асинхронных сообщений
asyncmsg_channel_destroy()	Уничтожить канал для передачи асинхронных сообщений
asyncmsg_connect_attach()	Установить соединение с каналом
asyncmsg_connect_detach()	Уничтожить соединение с каналом
asyncmsg_flush()	Объединить все сообщения в пакет
asyncmsg_connect_attr()	Получить атрибуты соединения
asyncmsg_put()/asyncmsg_putv()	Передать сообщение(я) одиночное или векторное
asyncmsg_get()	Получить асинхронные сообщения из канала
asyncmsg_free()	Освободить буфер сообщений
asyncmsg_malloc()	Поместить сообщение в буфер для пересылки

QNX Neutrino обеспечивает несколько системных функций для реализации асинхронного обмена, представленных в табл. 3.11. Данные функции поддерживают создание и управление асинхронным каналом, создание и управление соединениями, пакетную буферизацию и обмен сообщениями. Функция создания асинхронного канала имеет следующий формат:

```
int asyncmsg_channel_create(unsigned flags, mode_t mode, size_t buffer_size,
        unsigned max_num_buffer, const struct sigevent *ev
            int (*recvbuf_callback) (size_t bufsize, unsigned num_bufs,
              void *bufs[], int flags));
```

Аргументы:

- flags – флаги канала, принимают те же значения что и для функции Channel_create();
- mode – режим доступа к каналу, подобен режиму доступа к файлу;
- buffer_size – размер каждого буфера, используемого для хранения сообщений;
- ev – NULL или указатель к структуре, которая должна быть послана как уведомление о событии. Уведомление доставляется, если очередь была предварительно пуста;
- recvbuf_callback – функция, используемая библиотекой, чтобы создать буфер для входящих сообщений или освобождения буфера, когда канал освобождается. Если значение этого аргумента NULL, библиотека будет использовать функции malloc() и free().

Следующий программный фрагмент демонстрирует пересылку нескольких сообщений и прием всего пакета сообщений с помощью однократного вызова функции asyncmsg_get(). Источником и приемником сооб-

щений является один и тот же поток main(). Для формирования пакета необходимо при образовании соединения с каналом при помощи вызова asyncmsg_connect_attach() установить размер пакета. В данном примере размер пакета устанавливается к значению 3.

```
#include <stdio.h>
#include <process.h>
#include <unistd.h>
#include <sys/asyncmsg.h>

int callback(int err, void *cmsg, unsigned handle)
{
   printf("Callback: err = %d, msg = %p, handle = %d\n",
         err, cmsg, handle);
   return 0;
}
/*********************************************************************************/
int main()
{
   int chid, coid, i;
   struct _asyncmsg_connection_attr aca;
   struct _asyncmsg_get_header *agh, *agh1;
   char msg[3][80]; // массив сообщений

// создать канал
   if ((chid = asyncmsg_channel_create(_NTO_CHF_SENDER_LEN,
      0666, 2048, 5, NULL, NULL)) == -1)
   {
      perror("channel_create");
      return -1;
   }
      memset(&aca, 0, sizeof(aca));
      aca.buffer_size = 2048;     // размер буфера сообщения
      aca.max_num_buffer = 5;  // число буферов сообщений
      aca.trigger_num_msg = 3; // число сообщений в пакете

// создать соединение
      if ((coid = asyncmsg_connect_attach(0, 0, chid, 0, 0, &aca)) == -1)
      {
         perror("connect_attach");
         return -1;
      }
// сформировать и передать  пакет тестовых сообщений
      for (i = 0; i < 3; i++)
      {
          sprintf(msg[i], "Async Message Passing (msgid %d)\n", i);
          if ((asyncmsg_put(coid, msg[i], strlen(msg[i]) + 1,
             1234, callback)) == -1)
          {
             perror("put");
             return -1;
          }
      }
// получить пакет сообщений
      if ((agh = asyncmsg_get(chid)) == NULL)
      {
         perror("get");
         return -1;
```

```
        }
    // разобрать пакет и отобразить сообщения на дисплее
        printf("Got message(s): \n\n");
        while (agh1 = agh)
        {
           agh = agh1->next;
           printf("from process: %d (%d)\n", agh1->info.pid, getpid());
           printf("msglen: %d (%d)\n", agh1->info.msglen, strlen(msg) + 1);
           printf("srclen: %d\n", agh1->info.srcmsglen);
           printf("err: %d\n", agh1->err);
           printf("parts: %d\n", agh1->parts);
           printf("msg: %s\n\n", (char *)agh1->iov->iov_base);
           asyncmsg_free(agh1);
        }
           sleep(1);
         if (asyncmsg_connect_detach(coid) == -1) // уничтожить соединение
        {
           perror("connect_detach");
           return -1;
        }
        if (asyncmsg_channel_destroy(chid) == -1) // уничтожить канал
        {
           perror("channel_detach");
           return -1;
        }
        return 0;
    }
```

3.9. Очереди сообщений

Стандарт POSIX определяет набор неблокирующих средств передачи сообщений, известных как «POSIX-очереди сообщений». Для того чтобы использовать этот механизм обмена данными, необходимо запустить менеджер очередей mqueue. Ядро QNX Neutrino непосредственно не участвует в поддержке этого механизма. POSIX-очереди сообщений обеспечивают интерфейс, хорошо известный многим программистам. Они подобны почтовым ящикам, которые можно обнаружить во многих исполняющих системах реального времени. POSIX-очереди сообщений реализуют механизм приоритетного перемещения данных, при котором отправитель не блокируется и может иметь много не доставленных сообщений, упорядоченных в очередь. Фактически существует несколько очередей с разным уровнем приоритета. POSIX-очереди сообщений существуют независимо от процессов, которые их используют. Это более медленный механизм, чем естественные сообщения ядра, однако он обладает более широкими функциональными возможностями, что может в ряде случаев компенсировать недостаток быстродействия.

Очереди сообщений имеют сходство с файлами, во всяком случае, в том, что касается их интерфейса. Любая очередь сообщений, как и файл, имеет имя. Открыть очередь сообщений можно с помощью вызова

mq_open(), закрыть – с помощью mq_close() и уничтожить сообщения – с помощью mq_unlink(). Для того чтобы поместить данные в очередь сообщений и извлечь данные из очереди, используются вызовы mq_send() и mq_receive(). При точном соответствии со стандартом POSIX в имени очереди может присутствовать только один слеш (/). Однако в QNX стандарт имеет расширение и поддерживает имена, содержащие много слешей. Это позволяет, например, некоторой компании иметь все очереди сообщений, в которых слешами выделено имя компании, что дает возможность исключить конфликты при совместном использовании данного механизма обмена несколькими компаниями в рамках одной сети. В QNX Neutrino все очереди сообщений с ведущим символом (/) будут отображаться как имена файлов в директории /dev/mqueue. Эти имена можно отобразить на дисплее с помощью команды интерпретатора ls –Rl /dev/mqueue. Очереди сообщений с другими именами локализуются в текущем директории процесса, создающего очередь. Очередь сообщений может быть открыта и на другом узле, в этом случае спецификация имени должна включать предваряющий префикс /net/node/, где node – имя узла. Для управления очередью используются вызовы, представленные в табл. 3.12.

Таблица 3.12

Функция	Описание
mq_open()	Открыть очередь сообщений
mq_close()	Закрыть очередь сообщений
mq_unlink()	Удалить очередь сообщений
mq_send()	Добавить сообщение к очереди
mq_receive()	Получить сообщение из очереди
mq_notify()	Уведомить вызывающий процесс, что сообщение находится в очереди
mq_setattr()	Установить атрибуты очереди сообщений
mq_getattr()	Получить атрибуты очереди сообщений
mq_timedreceive()	Получить наиболее старое сообщение с наивысшим приоритетом
mq_timedsend()	Добавить в очередь сообщение с заданным временем доставки

Рассмотрим синтаксис основных вызовов. Функция mq_open() открывает очередь сообщений по имени и возвращает дескриптор очереди, подобный файловому дескриптору. При неудаче возвращается значение –1. Все последующие операции с очередью используют только этот дескриптор.

```
mqd_t mq_open( const char * name,   // имя очереди
        int oflag,                  // флаги
        int mode,                   // режим доступа (опциональный аргумент)
        int mq_attr,)               // атрибуты (опциональный аргумент)
Аргументы:
```

• oflag – флаги, определяющие режим доступа. Вы должны специфицировать O_RDONLY (только для чтения), O_WRONLY (только для записи) или O_RDWR (для чтения и записи). Кроме того, для получения дополнительных эффектов, к этим флагам по «ИЛИ» можно добавить следующие константы:

- O_CREAT – инструктирует сервер создать новую очередь, если имя, указанное в аргументе name, не существует,

- O_EXCL – используется вместе с флагом O_CREAT. Если очередь с указанным именем существует, то вызов не исполняется и возвращается ошибка в глобальной переменной errno с кодом EEXIST. При одиночном использовании данного флага он игнорируется,

- O_NONBLOCK – при сброшенном флаге вызовы mq_send() или mq_receive() могут быть заблокированы, если очередь полна или, наоборот, пуста. Если флаг установлен, то вызовы никогда не блокируются, но происходит возврат с установленной ошибкой EAGAIN в глобальной переменной errno;

• mode – аргумент присутствует в вызове mq_open(), только если установлен флаг O_CREAT, и определяет для новой очереди файловые разрешения для типовых категорий пользователей: собственник, член группы и прочие. Разрешения задаются с помощью символьных констант представленных в табл. 3.13;

• mq_attr – опциональный аргумент присутствует в вызове mq_open(), только если установлен флаг O_CREAT. Указывает на структуру mq_attr, которая содержит атрибуты для новой очереди сообщений. Структура имеет следующий набор полей:

- long mq_flags – флаги очереди,
- long mq_maxmsg – максимальное число сообщений в очереди,
- long mq_msgsize – максимальный размер каждого сообщения,
- long mq_curmsgs – число сообщений, непрерывно сохраняемых в данной очереди,
- long mq_sendwait – число потоков, непрерывно ожидающих пересылки сообщения. Ненулевое значение в этом поле означает, что очередь заполнена полностью,
- long mq_recvwait – число потоков, непрерывно ожидающих приема сообщения. Ненулевое значение в этом поле означает, что очередь пуста.

Значения полей структуры mq_attr для существующей очереди можно получить с помощью вызова mq_getattr() и установить с помощью вызова mq_setattr(). Если аргумент mq_attr имеет значение NULL, то создается новая очередь с атрибутами:

- mq_maxmsg: 1024 // **число сообщений в очереди**,
- mq_msgsize: 4096 // **максимальный размер сообщений**,
- mq_flags: 0 // **флаги**.

Таблица 3.13

Собственник	**Член группы**	**Прочие**	**Разрешения**
S_IRUSR	S_IRGRP	S_IROTH	Чтение
S_IRWXU	S_IRWXG	S_IRWXO	Чтение/запись/исполнение (поиск)
S_IWUSR	S_IWGRP	S_IWOTH	Запись
S_IXUSR	S_IXGRP	S_IXOTH	Исполнение (поиск)

Для передачи и приема сообщения используются вызовы mq_send() и mq_receive(). Синтаксис этих вызовов представлен в следующем фрагменте. Эти вызовы не блокируют поток, а в случае неудачи возвращают значение –1 и код ошибки в глобальной переменной errno.

```
int mq_send (mqd_t mqdes,      // дескриптор очереди
        const char * msg_ptr,  // указатель к сообщению
        size_t msg_len,        // размер сообщения
        unsigned int msg_prio ); // приоритет сообщения
```

Приоритет для сообщений может быть установлен в диапазоне от 0 до (MQ_PRIO_MAX-1). Константа MQ_PRIO_MAX устанавливается при запуске сервера mqueue. Вместо функции

mq_send() может быть использован вызов write(), он эквивалентен mq_send() со значением msg_prio=0. При неудаче вызов mq_send() возвращает значение —-1, а при успешном завершении – любое положительное значение.

```
ssize_t mq_receive( mqd_t mqdes,       // дескриптор очереди
                    char* msg_ptr,     // указатель к буферу сообщения
                    size_t msg_len,    // размер сообщения
                    unsigned int* msg_prio );   // приоритет сообщения
```

Если указатель msg_prio установлен в NULL, значение приоритета принятого сообщения не сохраняется. Вызов read() с дескриптором mqdes аналогичен вызову mq_receive() с аргументом msg_prio= NULL.

Сообщения, подобно файлам, существуют, даже если процессы, их создавшие, завершены. Если все процессы отсоединены от очереди вызовами mq_unlink() и последний связанный с ней процесс выполняет вызов mq_close(), то очередь уничтожается со всем ее содержимым. Вызов close() с дескриптором очереди имеет тот же самый эффект, что и вызов mq_close(). Для уведомления процессов о поступлении сообщений в очередь может быть использован вызов mq_notify(). Синтаксис этого вызова представлен в следующем фрагменте.

```
int mq_notify(mqd_t mqdes,                  // дескриптор очереди
    const struct sigevent* notification ); // уведомление
```

Тип уведомления notification устанавливается структурой sigevent. Рекомендуются следующие типы уведомлений: SIGEV_SIGNAL, SIGEV_SIGNAL_CODE, SIGEV_SIGNAL_THREAD, SIGEV_PULSE, SIGEV_INTR (см. табл. 3.10).

Если аргумент notification не равен NULL, то вызов просит сервер уведомить вызывающий процесс, когда в пустую очередь поступит хотя бы одно сообщение. После пересылки уведомление удаляется. Только один процесс на интервале ожидания имеет право зарегистрировать уведомление. Если другой процесс в это время пытается выполнить регистрацию уведомления, вызов отклоняется с ошибкой EBUSY в переменной errno. Если некоторый процесс зарегистрировал уведомление, а другой процесс блокирован по mq_receive(), тогда при поступлении сообщения в очередь производится деблокировка ожидающего потока. Уведомление при этом не передается, поскольку считается, что очередь осталась пустой. Если аргумент notification равен NULL и вызывается процессом, который ранее зарегистрировал уведомление, то существующая регистрация удаляется.

4. Дополнение к механизмам синхронизации

4.1. Именованные семафоры

Методы синхронизации потоков, описанные в разд. 1, работают только в пределах одного узла. Единственным исключением является механизм именованных семафоров, которые, как и очереди сообщений, поддерживаются менеджером mqueue. Именованные семафоры имеют то же самое назначение и функции управления, что и обычные семафоры, но отличаются наличием имени, которое регистрируется менеджером mqueue. Благодаря имени они становятся известными не только в пределах локального узла, но и во всей сети. Следующие системные вызовы (табл. 4.1) поддерживают функциональность именованных семафоров.

Таблица 4.1

Функция	Описание
sem_close()	Закрыть именованный семафор
sem_getvalue()	Получить значение семафора
sem_open()	Создать или получить доступ к именованному семафору
sem_post()	Увеличить значение счетчика
sem_trywait()	Проверить семафор без блокировки
sem_unlink()	Уничтожить именованный семафор
sem_wait()	Уменьшить значение счетчика семафора

Синтаксис некоторых функций представлен в следующем фрагменте.

```
sem_t * sem_open( const char * sem_name,    // имя семафора
                  int oflags,               // флаги семафора
                  ... );
```

Функция возвращает дескриптор семафора, который далее используется во всех вызовах.

Аргументы:

• sem_name – имя семафора, это последовательность символов, разделенная слешами (/). Если имя имеет ведущий слеш, то оно регистрируется как имя файла в директории /dev/sem. Все другие имена регистрируются в текущем директории процесса, создавшего семафор;

• oflags – флаги семафора, используются при его создании и могут иметь только значения O_CREAT и O_EXCL. Назначение данных флагов и их использование точно такое же, как и для очереди сообщений.

Если установлен флаг O_CREAT, означающий создание нового семафора, то вызов дополняется еще двумя аргументами:

• mode_t mode – режим доступа к семафору, соответствует режиму доступа для очередей сообщений (см. табл. 3.13). Для нормального функционирования следует установить доступ по чтению, записи и исполнению, например следующим образом:

- S_IRWXG для доступа членов группы,
- S_IRWXO для прочих,
- S_IRWXU для собственника;

• unsigned int value – это начальное значение счетчика семафора. При функционировании семафора положительное значение счетчика (т. е. значение больше нуля) указывает на неблокированный семафор, а значение нуль и меньше нуля – на блокированный. Начальное значение счетчика не должно превышать константу SEM_VALUE_MAX, устанавливаемую при запуске менеджера.

```
int sem_init( sem_t * sem,        // дескриптор семафора
              int pshared,        // признак разделения в памяти
              unsigned value );   // значение счетчика
```

Вызов используется для динамической инициализации семафора. Нулевое значение признака pshared указывает, что семафор будет использоваться процессами через механизм разделяемой памяти. Это приемлемо только в пределах одного узла. Функция возвращает нуль при удачном завершении и –1 при ошибке.

Остальные функции полностью подобны функциям обычного семафора.

4.2. Таймеры

Таймеры являются источниками событий, которые служат для синхронизации вычислительного процесса во времени. QNX Neutrino обеспечивает полный набор таймерных возможностей, определяемых стандартом POSIX. POSIX-модель позволяет иметь:

- абсолютную дату;
- относительную дату;
- циклический режим.

Циклический режим наиболее важен, поскольку в большинстве случаев периодический источник событий подталкивает программный поток к выполнению какого-либо действия. Поскольку таймеры являются источниками событий, то они также используют процедуру доставки событий, подобную MsgDeliverEvent() со структурой sigevent.

Таймер – это объект ядра, который обрабатывает прерывания, поступающие от аппаратного генератора синхроимпульсов, размещенного на материнской плате или в кристалле процессора. Для того чтобы работать с таймером, необходимо:

1) создать объект типа «таймер»;
2) выбрать тип уведомления о событии;
3) выбрать нужный тип таймера (абсолютный, относительный, периодический, однократный);
4) запустить таймер.

Первый этап (создание таймера) выполняется с помощью функции timer_create():

```
int TimerCreate( clockid_t id,                  // временная база
        const struct sigevent *event );         // уведомление о событии
```

Функция возвращает идентификатор таймера, который дальше используется во всех вызовах.

Аргументы:

- id – временная база системных часов;
- event – указатель к структуре sigevent, которая содержит тип события, генерируемого при срабатывании таймера.

POSIX-стандарт утверждает, что на различных платформах можно использовать различные типы временных базисов, но любая платформа должна, по меньшей мере, поддерживать базис CLOCK_REALTIME. В QNX Neutrino есть 3 временных базиса:

- CLOCK_REALTIME – таймеры, основанные на этой временной базе, будут пробуждать процессор в любых режимах. Часы, основанные на этой временной базе, допускают подстройку, которую можно выполнить с помощью функции ClockAdjust();

- CLOCK_SOFTTIME – эта временная база подобна CLOCK_REALTIME, за исключением режима сниженного энергопотребления, где ее не следует использовать;

- CLOCK_MONOTONIC – это временная база, которая приемлема для всех режимов, но не допускает подстройку.

В структуре sigevent рекомендуется указывать следующие типы уведомлений для таймерных событий: SIGEV_SIGNAL, SIGEV_SIGNAL_CODE, SIGEV_SIGNAL_THREAD, SIGEV_PULSE (см. табл. 3.10). Создав таймер, необходимо выбрать его вид. Это осуществляется комбинированием аргументов функции timer_settime(), которая обычно применяется для запуска таймера:

```
int TimerSettime( timer_t id,                 // идентификатор таймера
                  int flags,                  // флаги
                  const struct _itimer * itime,  // текущие установки
                  struct _itimer * oitime );  // прежние установки
```

Аргументы:

- id – идентификатор таймера, полученный при вызове TimerCreate();
- flags – единственный поддерживаемый флаг – это TIMER_ABSTIME. Если он установлен, то поддерживается абсолютное время, начиная с нуля часов 1 января 1970 г. Когда флаг сброшен, то таймер ведет отсчет относительного времени с момента запуска;
- itime – указатель к структуре _itimer, которая специфицирует время первого срабатывания таймера и период срабатывания. Если указанное время первого срабатывания уже истекло, то событие, связанное с таймером, доставляется немедленно;
- oitime – это NULL или указатель к структуре _itimer, где сохраняются прежние установки.

Структура _itimer состоит из двух полей:

```
– uint64_t nsec            // время срабатывания таймера в наносекундах
– uint64_t interval_nsec   // интервал срабатывания таймера в наносекундах
```

Если поле nsec = 0, то таймер запрещен. Ненулевое значение определяет момент первого срабатывания таймера. Если поле interval_nsec =0, то таймер сработает только однократно. Ненулевое значение определяет период срабатывания таймера, начиная с момента первого срабатывания.

Если на момент вызова TimerSettime() таймер уже был запущен, то этот вызов сбросит старые установки и заменит их на новые. Другие системные вызовы, связанные с временной синхронизацией, представлены в табл. 4.2.

Вызов TimerAlarm() предназначен для формирования сигнала SIGALRM («тревога») при срабатывании таймера:

```
int TimerAlarm( clockid_t id,                  // временная база
                const struct _itimer * itime,  // текущие установки
                struct _itimer * otime );      // прежние установки
```

Аргументы функции подобны аргументам вызова TimerSettime(). Сигнал SIGALRM доставляется к ожидающему потоку, который издал вызов TimerAlarm(). Доставка сигналов не поддерживает очередности. Новый сигнал заменяет старый, если старый не был обработан.

Таблица 4.2

Функция микроядра	POSIX-вызов	Описание
TimerAlarm()	alarm()	Послать сигнал SIGALRM при срабатывании таймера
TimerCreate()	timer_create()	Создать интервальный таймер
TimerDestroy()	timer_delete()	Уничтожить интервальный таймер
TimerGettime()	timer_gettime()	Получить время до момента срабатывания таймера
TimerGetoverrun()	timer_getoverrun()	Получить число перезапусков интервального таймера
TimerSettime()	timer_settime()	Запустить интервальный таймер.
TimerInfo(),		Получить полную информацию о таймере
TimerTimeout()	sleep(), nanosleep(), sigtimedwait(), pthread_cond_timedwait(), pthread_mutex_trylock(), intr_timed_wait()	Установить тайм-аут для блокирующего состояния

Если новый вызов TimerAlarm() вызывается до момента срабатывания прежнего, то временной счет таймера перезапускается с новыми значениями.

Заключение

В данном учебном пособии представлены ключевые особенности операционных систем реального времени, которые нашли свое отражение в современном международном стандарте POSIX. Хотя изложение выполнено в контексте операционной системы QNX Neutrino, вызовы, соответствующие стандарту POSIX, представлены достаточно полно. Специфические вызовы QNX Neutrino дополняют возможности стандарта и не противоречат ему. Пособие не требует глубокого понимания языка Си, но знание его определенно дает преимущество, поскольку в тексте есть примеры программ и системных вызовов. Целью автора было показать функциональные особенности систем реального времени на примере конкретной операционной системы, детализируя изложение материала только в той мере, которая достаточна для понимания основных механизмов. Пособие ни в коем случае не подменяет техническую документацию QNX Neutrino, но может служить средством, способствующим ее глубокому пониманию.

Список рекомендуемой литературы

Операционная система реального времени QNX Neutrino 6.3. Системная архитектура / Пер. с англ. - СПб.: БХВ-Петербург, 2005. - 336 с.

Зыль С. Н. QNX Momentix: основы применения. - СПб.: БХВ-Петербург, 2005. - 256 с.

Зыль С. Н. Операционная система реального времени QNX от теории к практике. 2-е изд. - СПб.: БХВ-Петербург, 2004. - 192 с.

Кёртен Р. Введение в QNX Neutrino 2. Руководство по программированию приложений реального времени / Пер. с англ. - СПб.: Петрополис, 2001. - 479 с.

Оглавление

ВВЕДЕНИЕ **3**

1. МИКРОЯДРО, ПОТОКИ И ПРОЦЕССЫ **4**

1.1. Микроядро 5
1.2. Атрибуты потоков 7
1.3. Состояния потока 7
1.4. Диспетчеризация потоков 9
1.5. Дисциплины диспетчеризации 10
1.6. Управление приоритетами и дисциплиной диспетчеризации 13
1.7. Управление потоками 13

2. МЕХАНИЗМЫ СИНХРОНИЗАЦИИ ПОТОКОВ **16**

2.1. Мьютексы 17
2.2. Дополнительные сервисы QNX-Neutrino 19
2.3. Условные переменные 20
2.4. Ждущие блокировки 22
2.5. Ждущие блокировки в сравнении с условными переменными 25
2.6. Барьеры 25
2.7. Семафоры 27
2.8. Блокировки чтения/записи 28
2.9. Синхронизация через диспетчеризующий механизм 30
2.10. Синхронизация через атомарные функции 30
2.11. Синхронизация через сообщения 31
2.12. Тайм-ауты ядра 32

3. МЕХАНИЗМЫ ВЗАИМОДЕЙСТВИЯ ПРОГРАММНЫХ ПОТОКОВ **34**

3.1. Синхронный обмен сообщениями 35
3.2. Векторные сообщения 40
3.3. Многошаговый обмен сообщениями 40
3.4. Локализация сервера синхронного канала 42
3.5. Импульсы 43
3.6. Сигналы 44
3.7. Иерархический принцип обмена синхронными сообщениями 50
3.8. Асинхронный обмен сообщениями 53
3.9. Очереди сообщений 56

4. ДОПОЛНЕНИЕ К МЕХАНИЗМАМ СИНХРОНИЗАЦИИ **59**

4.1. Именованные семафоры 59
4.2. Таймеры 61

ЗАКЛЮЧЕНИЕ **63**

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ **64**