Усовершенствованные методы синхронизации потоков

В предыдущей главе были описаны проблемы производительности, возникающие в Windows, и способы их преодоления в реалистичных ситуациях. В главе 8 обсуждался ряд простых задач, требующих привлечения объектов синхронизации. В настоящей главе на основании идей, изложенные в главах 8 и 9, решаются задачи, которые также встречаются в реальной практике, но отличаются большей сложностью.

Первое, что нам предстоит сделать — это объединить два или более объекта синхронизации вместе с данными для создания сложного объекта синхронизации. Наиболее полезной комбинацией такого рода является модель переменных условий (condition variable model), включающая мьютекс и одно или несколько событий. Указанная модель играет весьма существенную роль в самых различных практических ситуациях, поскольку многие серьезные программные дефекты, обусловленные влиянием состязательности, проявляются именно тогда, когда объекты синхронизации Windows, особенно события, используются программистами неправильно. События имеют сложную природу и ведут себя по-разному в зависимости от того, какой именно из описанных в табл. 8.1 вариантов используется, и поэтому следует придерживаться определенных правил, устанавливаемых хорошо изученными моделями.

В последующих разделах показано, как систематизировать управление запуском и отменой выполнения каждого из совместно работающих потоков при помощи асинхронного вызова процедур (Asynchronous Procedure Calls, APC).

Другие проблемы производительности обсуждаются по мере необходимости.

Модель переменных условий и свойства безопасности

Многопоточные программы намного легче разрабатывать, делать их более понятными и сопровождать, если использовать известные, хорошо разработанные методики и модели. Эти вопросы уже обсуждались в главе 7, в которой для создания полезной концептуальной основы, позволяющей понять принципы работы многопоточных программ, были введены модель "хозяин/рабочий" ("boss/worker") и модель рабочей группы (work crew model). Понятие критических участков кода (critical sections) играет существенную роль при использовании мьютексов, а определение инвариантов (invariants) используемых структур данных также может принести немалую пользу. Наконец, даже для дефектов существуют свои модели, как это было показано на примере взаимной блокировки (deadlock) потоков.

Примечание

Компания Microsoft разработала собственный набор моделей, таких как апартаментная модель (apartment model) или модель свободных потоков (free

threading). Эта терминология чаще всего встречается в технологии COM и кратко обсуждается в конце главы 11.

Совместное использование событий и мьютексов

Далее показано, как обеспечить совместное использование мьютексов и событий путем обобщения программы 8.2, представляющей описанную ниже ситуацию, с которой нам еще не раз предстоит столкнуться. *Примечание.* Это обсуждение в равной степени применимо как к мьютексам, так и к объектам CRITICAL_SECTION.

- Как мьютекс, так и событие связываются с блоком сообщений или иной структурой данных.
- Мьютекс определяет критический участок кода для доступа к объекту структуры данных.
- Событие используется для того, чтобы сигнализировать о появлении нового сообщения.
- Обобщая, можно утверждать, что мьютекс обеспечивает соблюдение условий, определяемых инвариантами объекта (то есть поддерживает свойства безопасности), а событие сигнализирует о том, что состояние объекта изменилось (например, сообщение было добавлено в буфер сообщений или удалено из него), и он мог перейти в известное состояние (например, в буфере сообщений присутствует, по крайней мере, одно сообщение).
- Один поток (в программе 8.2 — поток производителя) блокирует структуру данных, изменяет состояние объекта путем создания нового сообщения и применяет функции SetEvent или PulseEvent к событию, связанному с появлением нового сообщения.
- По крайней мере, один поток из числа остальных (в данном примере — поток потребителя) ожидает наступления события, сигнализирующего о том, что объект достиг требуемого состояния. Ожидание должно выполняться за пределами критического участка кода, чтобы поток производителя мог иметь доступ к объекту.
- Кроме того, поток потребителя может блокировать мьютекс, проверить состояние объекта (например, поступило ли в буфер новое сообщение) и отказаться от ожидания события, если объект уже находится в требуемом состоянии.

Модель переменных условий

А теперь давайте объединим все это в едином фрагменте кода, представляющем то, что мы будем называть *моделью переменных условий* (condition variable model, CV model), которая может существовать в виде *сигнальной* (signal) и *широковещательной* (broadcast) моделей. В первых примерах будет использована широковещательная модель. Результат представляет собой программную модель, с которой мы будем работать еще

не один раз, и которая может быть использована для решения широкого круга задач синхронизации. Для удобства изложения примеры сформулированы в терминах задачи производителя и потребителя.

Обсуждение может показаться вам несколько абстрактным, однако, поняв суть методики, вы сможете решать многие задачи синхронизации, справиться с которыми без наличия хорошей модели было бы очень трудно.

В упомянутом фрагменте кода имеется несколько ключевых элементов.

• Структура данных типа STATE_TYPE, в которой содержатся все данные, или *переменные состояний* (state variables), такие как сообщения, контрольные суммы и счетчики, используемые в программе 8.2.

• Мьютекс и одно или более событий, связанных с этой структурой данных и обычно входящих в ее состав.

• Одна или несколько булевых функций, предназначенных для вычисления *предикатов переменных условий* (condition variable predicates), представляющих собой условия (состояния), наступления которых может ожидать поток. Например, в качестве предикатов могут использоваться следующие условия: "готово новое сообщение", "имеется свободное место в буфере", "очередь не пуста". Можно связывать с каждым предикатом переменной условия отдельное событие, но возможно также использование одного события для представления изменения состояния или же для представления комбинации нескольких предикатов (получаемой посредством применения операции логического "или"). В последнем случае для определения фактического состояния должны проверяться возвращаемые значения отдельных предикативных функций при блокированном мьютексе. Если предикат (логическое выражение) является простым, необходимость в использовании отдельной функции отпадает.

Эти принципы используются потоками производителя и потребителя в приведенном ниже фрагменте кода, включающем единственное событие и предикат переменной условия (реализованный с помощью функции cvp, которая здесь не представлена). В данном примере принимается, что если поток производителя сигнализирует о достижении требуемого состояния, то должны быть освобождены сразу несколько потоков, откуда следует, что сигнал должен рассылаться всем ожидающим потокам потребителя. Так, сигналом, соответствующим созданию потоком производителя нескольких сообщений, может служить увеличение значения счетчика сообщений. Во многих случаях вам может требоваться освобождение только одного потока, что обсуждается после приведенного ниже фрагмента кода.

Этот код ориентирован на работу под управлением Windows 9x и всех версий Windows NT. Для упрощения решения впоследствии в нем будет использована функция SignalObjectAndWait.

Примечание и предостережение

В данном примере намеренно и вполне осознанно используется функция PulseEvent, хотя некоторые авторы, а кое-где и документация Microsoft (см. замечания в соответствующем разделе MSDN), этого делать не рекомендуют. Причины нашего выбора будут ясны из последующего обсуждения и

подкреплены примерами, а читателю предлагается решить (корректно) эту задачу, используя функцию SetEvent.

```
typedef struct _state_t {
 HANDLE Guard; /* Мьютекс, защищающий объект. */
 HANDLE CvpSet; /* Вручную сбрасываемое событие — выполняется условие, определяемое предикатом cvp(). */
 … другие переменные условий …
 /* Структура состояния, содержащая счетчики, контрольные суммы и прочее. */
 struct STATE_VAR_TYPE StateVar;
 } STATE_TYPE State;
…
/* Инициализировать состояние, создавая мьютекс и событие. */
…
/* Поток ПРОИЗВОДИТЕЛЯ, который изменяет состояние. */
WaitForSingleObject(State.Guard, INFINITE);
/* Изменить состояние таким образом, чтобы выполнялось условие, */
/* определяемое предикатом CV. */
/* Пример: к данному моменту подготовлено одно или несколько сообщений.*/
State.StateVar.MsgCount += N;
PulseEvent(State.CvpSet);
ReleaseMutex(State.Guard);
/* Конец интересующей нас части кода потока производителя. */
…
/* Ожидание определенного состояния функцией потока ПОТРЕБИТЕЛЯ. */
WaitForSingleObject(State.Guard, INFINITE);
while (!cvp(&State)) {
 ReleaseMutex(State.Guard);
 WaitForSingleObject(State.CvpSet, TimeOut);
 WaitForSingleObject(State.Guard, INFINITE);
}
/* Теперь этот поток владеет мьютексом, и выполняется условие, */
/* определяемое предикатом cvp(&State). */
/* Предпринять соответствующее действие, возможно, изменяя состояние.*/
…
ReleaseMutex(State.Guard);
/* Конец интересующей нас части кода потока потребителя. */
```

Комментарии по поводу модели переменных условий

В приведенном выше фрагменте кода очень важная роль принадлежит циклу в той части кода, которая соответствует потребителю. Этот цикл

включает три операции: 1) освобождение мьютекса, заблокированного до входа в цикл; 2) ожидание события; 3) повторное блокирование мьютекса. Как будет показано далее, *использование конечного интервала ожидания события является весьма существенным.*

Потоки Pthreads в том виде, в каком они реализованы во многих системах UNIX и других системах, сочетают эти три операции в одной функции — pthread_cond_wait, объединяющей мьютекс и переменную условия (которая аналогична, но не идентична событиям Windows). Именно поэтому и используется термин *модель переменных условий*. Существует также версия этой функции, допускающая использование конечных интервалов ожидания событий.

Что немаловажно, в Pthreads первые две операции (освобождение мьютекса и ожидание события) реализуются посредством вызова одной функции как одна атомарная операция, так что никакой другой поток не сможет вклиниться раньше, чем начнется выполнения вызывающим потоком функции ожидания наступления события (или выполнения условия).

Проектировщики Pthreads сделали мудрый выбор: единственный способ организовать ожидания выполнения условия, определенного для переменной условия, — это использование одной из двух указанных выше функций (с конечным и неопределенным интервалами ожидания), так что переменная условия должна всегда использоваться вместе с мьютексом. Windows вынуждает вас использовать для этой цели два или три отдельных вызова функций, и вы сами должны проследить за тем, чтобы все было сделано правильно, иначе вам не избежать проблем.

Помимо того, что это упрощает разработку программ и является существенно необходимым в случае использования потоков Pthreads, есть еще одна причина, по которой следует изучать модель CV, заключающаяся в том, что именно эта модель используется рядом сторонних производителей для реализации классов потоков и объектов синхронизации, не зависящих от ОС. Владея изложенным в этой книге материалом, вы сможете очень быстро разобраться в особенностях этих реализаций.

Примечание

В версии Windows NT 4.0 была введена новая функция — SignalObjectAndWait (SOAW), которая выполняет упомянутые два шага атомарным образом. В дальнейших примерах программ предполагается, что эта функция доступна, и она будет использоваться, а это означает, что под управлением Windows 9x такие программы выполняться не смогут. Тем не менее, на стадии ознакомления с моделью CV функция SOAW не применяется, чтобы сделать более понятной мотивировку необходимости ее использования впоследствии, а на Web-сайте книги приведены альтернативные варианты реализации некоторых примеров, в которых вместо мьютексов используются объекты CS. (Функцию SOAW нельзя применять вместе с объектами CS.) О значительных преимуществах функции SignalObjectAndWait в отношении производительности свидетельствуют данные, представленные в приложении B (табл. B.5).

Использование модели переменных условий

Модель переменных условий при правильной ее реализации работает следующим образом:

• Поток производителя блокирует мьютекс, изменяет состояние, применяет к событию функцию PulseEvent, когда это необходимо, и разблокирует мьютекс. Например, функция PulseEvent может вызываться в случае готовности одного или нескольких сообщений.

• Функция PulseEvent должна применяться к событию при блокированном мьютексе, чтобы никакой другой поток не мог изменить объект, что могло бы сделать недействительным условие, определенное предикатом.

• Поток потребителя тестирует предикат переменной условия при блокированном мьютексе. Если условие, выраженное предикатом, выполняется, выполнять функцию ожидания нет никакой необходимости.

• Если же условие, выраженное предикатом, не выполняется, поток потребителя должен разблокировать мьютекс до выполнения ожидания события. Если этого не сделать, то никакой поток вообще не сможет изменить состояние и установить событие.

• Интервал ожидания события должен быть конечным, чтобы обеспечить правильную обработку в том случае, если поток производителя применит к событию функцию PulseEvent в промежутке времени между освобождением мьютекса (шаг 1) и выполнением ожидания события (шаг 2). Таким образом, без использования *конечного* интервала ожидания сигнал мог бы потеряться, что является еще одним примером проявления проблемы состязательности. К потере сигналов могут приводить и асинхронные вызовы процедур, описанные далее в этой главе. Используемый в приведенном выше фрагменте кода интервал ожидания является настраиваемым параметром. (С комментариями по поводу оптимальных значений этого параметра вы можете ознакомиться, обратившись к приложению В.)

• По завершении ожидания события поток потребителя всегда повторно проверяет выполнение условия, определенного предикатом. Среди прочих других причин, это необходимо делать с учетом того, что интервал ожидания может просто исчерпаться. Кроме того, за это время состояние также могло измениться. Например, поток производителя мог сгенерировать два сообщения, а затем освободить три ожидающих потока потребителя, в результате чего один из потребителей проверит состояние, определит, что сообщения отсутствуют, и продолжит выполнение ожидания. Наконец, повторная проверка предиката необходима для защиты от ложного пробуждения потоков, которое могло бы произойти в результате того, что поток установит событие в сигнальное состояние или применит к нему функцию PulseEvent без предварительного блокирования мьютекса.

• После выхода из цикла поток потребителя всегда сохраняет за собой право владения мьютексом, независимо от того, выполнялось или не выполнялось тело цикла.

Разновидности модели переменных условий

Прежде всего, обратите внимание на то, что в предшествующем фрагменте кода используется сбрасываемое вручную событие и вызывается функция PulseEvent, а не функция SetEvent. Является ли такой выбор корректным и возможен ли иной способ использования события? Ответ на оба эти вопросы является положительным.

Вернувшись к табл. 8.1, можно увидеть, что сбрасываемые вручную события характеризуются освобождением *нескольких потоков.* Это именно так в случае нашего примера, в котором генерируются несколько сообщений и существует несколько потоков потребителя, и все они должны быть оповещены о произошедших изменениях. В то же время, если поток производителя создает всего лишь одно сообщение и имеется несколько потоков потребителя, то событие должно быть автоматически сбрасываемым, а поток производителя должен вызывать функцию SetEvent, чтобы обеспечить освобождение только одного потока. В этом случае мы имеем дело не с сигнальной разновидностью модели CV, а с широковещательной. При этом по-прежнему остается существенным, чтобы освобожденный поток потребителя, который приобретает права владения мьютексом, изменил объект для указания того, что доступные сообщения отсутствуют (то есть, что условие, определяемое предикатом переменной условия, уже не выполняется).

Из четырех возможных комбинаций, указанных в табл. 8.1, для модели переменных условий важны только две. Что касается двух других комбинаций, то в силу конечности интервала ожидания эффект комбинации "автоматически сбрасываемое событие/PulseEvent" будет тем же, что и комбинации "автоматически сбрасываемое событие/SetEvent" (сигнальная модель CV), однако зависимость от длительности интервала ожидания приведет к снижению характеристик реактивности.

Использование же комбинации "вручную сбрасываемое событие/PulseEvent" приведет к появлению ложных сигналов (от которых, правда, можно защититься проверкой предикатов переменных условий), поскольку событие должно быть сброшено каким-либо из потоков, а до сброса события потоки будут состязаться между собой.

Подводя итоги, можно сделать вывод, что комбинация "автоматически сбрасываемое событие/SetEvent" представляет собой сигнальную модель CV, в которой освобождается единственный из ожидающих потоков, а комбинация "вручную сбрасываемое событие/PulseEvent" — широковещательную модель CV, в которой освобождаются все ожидающие потоки. Для потоков Pthreads существуют те же различия, но использование конечных интервалов ожидания событий для широковещательной модели в данном случае не требуется, тогда как в Windows этот фактор является весьма существенным, поскольку освобождение мьютекса и ожидание события не выполняются атомарно, то есть за одну операцию. В то же время, введение функции SignalObjectAndWait меняет эту ситуацию.

Пример предиката переменной условия

Рассмотрим следующий предикат переменной условия:
State.StateVar.Count >= K;

В данном случае поток потребителя будет ожидать до тех пор, пока значение счетчика не станет достаточно большим, и поток производителя может увеличивать это значение на произвольную величину. Отсюда, например, становится понятным, как можно реализовать сложные семафоры; вспомните, что обычные семафоры не допускают атомарного выполнения нескольких функций ожидания. В данном же случае поток потребителя может просто уменьшить значение счетчика на К единиц после выхода из цикла, но перед тем, как освободить мьютекс.

Заметьте, что в данном случае подходит широковещательная модель CV, поскольку один поток производителя может увеличить значение счетчика и тем самым разрешить выполнение нескольким, но не всем ожидающим потокам потребителя.

Семафоры и модель переменных условий

В некоторых случаях уместнее использовать не события, а семафоры, преимущество которых заключается в том, что они позволяют указывать точное количество потоков, которые необходимо освободить. Например, если бы было известно, что каждый из потоков потребителя может получить только одно сообщение, то поток производителя мог бы вызвать функцию ReleaseSemaphore, используя в качестве параметра точное количество сгенерированных сообщений. Однако в общем случае потоку производителя ничего не известно о том, каким образом отдельные потоки потребителя изменят структуру переменной состояния, и поэтому модель переменных условий применима для решения более широкого круга задач.

Модель CV обладает достаточно мощными возможностями, которых хватает для реализации семафоров. Как уже отмечалось ранее, в основе этого метода лежит определение предиката, эквивалентного утверждению: "значение счетчика является ненулевым", и создание структуры состояния, содержащей текущее значение счетчика и его максимально допустимое значение. В упражнении 10.11 представлено завершенное решение, позволяющее манипулировать функциями ожидания путем изменения значений счетчика на несколько единиц одной операцией. Создание семафоров для потоков Pthreads не предусмотрено, поскольку переменные условий предоставляют достаточно широкие возможности.

Использование функции SignalObjectAndWait

Цикл, выполняемый потоком потребителя в предшествующем фрагменте кода, играет очень важную роль в модели CV, поскольку в нем выполняется ожидание изменения состояния, а затем проверяется, является ли состояние

именно тем, какое требуется. Последнее условие может не выдерживаться, если событие оказывается слишком *обобщенным,* указывая, например, только на сам факт изменения состояния, а не на характеристики такого изменения. К тому же, другие потоки могут дополнительно изменить состояние, например, очистить буфер сообщений. Упомянутый цикл требовал выполнения двух функций ожидания и одной функции освобождения мьютекса, как показано ниже.

```
while (!cvp(&State)) {
 ReleaseMutex(State.Guard);
 WaitForSingleObject(State.CvpSet, TimeOut);
 WaitForSingleObject(State.Guard, INFINITE);
}
```

Использование конечного интервала ожидания (time-out) при выполнении первой функции ожидания (ожидание события) требуется здесь для того, чтобы избежать потери сигналов или возникновения других вероятных проблем. Этот код будет работать как под управлением Windows 9x, так и под управлением Windows NT 3.5 (еще одна устаревшая версия Windows), а предыдущий фрагмент кода сохранит свою работоспособность и в том случае, если мьютексы заменить объектами CS.

Однако в случае Windows NT 5.x (XP, 2000 и Server 2003) и даже Windows NT 4.0 мы можем использовать функцию SignalObjectAndWait — важный элемент усовершенствования, который избавляет от необходимости применения конечных интервалов ожидания и объединяет освобождение мьютекса и ожидание события. При этом кроме явного упрощения программы, производительность в общем случае повышается, что объясняется устранением системного вызова и отсутствием необходимости в настройке длительности интервала ожидания.

```
DWORD SignalObjectAndWait(HANDLE hObjectToSignal, HANDLE
hObjectToWaitOn, DWORD dwMilliseconds, BOOL bAlertable)
```

Эта функция, при вызове которой используются дескрипторы, указывающие соответственно на мьютекс и событие, упрощает цикл потребителя. Интервал ожидания здесь отсутствует, поскольку вызывающий поток переходит к ожиданию второго дескриптора *сразу же* после того, как первый дескриптор переходит в сигнальное состояние (что в данном случае означает освобождение мьютекса). Перевод объекта в сигнальное состояние и переход к ожиданию осуществляются атомарным образом, то есть за одну операцию, так что никакой другой поток не может сигнализировать о наступлении события в течение промежутка времени между освобождением мьютекса вызывающим потоком и ожидания потоком события, на которое указывает второй дескриптор. Тогда упрощенный цикл потребителя приобретает следующий вид:

```
while (!cvp(&State)) {
 SignalObjectAndWait(State.Guard, State.CvpSet, INFINITE, FALSE);
 WaitForSingleObject (State.Guard, INFINITE);
}
```

Значением последнего аргумента этой функции, bAlertable, в данном случае является FALSE, однако в последующих разделах, посвященных рассмотрению APC, он будет полагаться равным TRUE.

Вообще говоря, оба дескриптора могут указывать на любые подходящие объекты синхронизации. В то же время, использовать объект CRITICAL_SECTION в качестве объекта, сигнальное состояние которого отслеживается, нельзя, поскольку допустимыми являются только объекты ядра.

Функция SignalObjectAndWait применяется во всех примерах программ, представленных как в книге, так и на Web-сайте, хотя на Web-сайте находятся и другие варианты решений, о которых будет говориться в тексте. Если программа должна выполняться под управлением Windows 9x, то следует заменить эту функцию парой функций "сигнал/ожидание", как в первоначально приведенном фрагменте кода, и обязательно использовать конечный интервал ожидания.

В разделе, посвященном APC, представлены различные методы отправки сигналов ожидающим потокам, обеспечивающие получение сигналов только определенными потоками, тогда как в случае событий простых способов, позволяющих контролировать, каким потокам направляются сигналы, не существует.

Пример: объект порогового барьера

Предположим, вам необходимо, чтобы рабочие потоки оставались в состоянии ожидания и не выполнялись до тех пор, пока количество таких потоков не станет достаточным для образования рабочей группы, способной выполнить нужную работу. Как только количество потоков достигает порогового значения, все ожидающие рабочие потоки начинают выполняться, а появляющиеся впоследствии дополнительные рабочие потоки будут выполняться без ожидания. Эту задачу можно решить путем создания сложного объекта порогового барьера (threshold barrier compound object).

В программах 10.1 и 10.2 представлена реализация трех функций, поддерживающих сложный объект барьера. Две из этих функций, CreateThresholdBarrier и CloseThresholdBarrier, управляют переменными THB_HANDLE, аналогичными дескрипторам, которые на протяжении всего времени применялись нами вместе с объектами ядра. Пороговое количество потоков является параметром функции CreateThresholdBarrier.

Программа 10.1 представляет соответствующую часть заголовочного файла, SynchObj.h, тогда как программа 10.2 — реализацию трех упомянутых функций. Обратите внимание, что объект барьера содержит мьютекс, событие, счетчик и пороговое значение. Предикат переменной условия документирован в заголовочном файле, а именно, событие должно устанавливаться только тогда, когда значение счетчика достигает или становится больше порогового значения.

Программа 10.1. SynchObj.h: часть 1 — объявления объекта порогового барьера

```
/* Глава 10. Сложные объекты синхронизации. */
#define CV_TIMEOUT 50 /* Настраиваемый параметр для модели CV. */
/* ОБЪЕКТ ПОРОГОВОГО БАРЬЕРА — ОПРЕДЕЛЕНИЕ ТИПА И ПРОТОТИПЫ ФУНКЦИЙ. */
typedef struct THRESHOLD_BARRIER_TAG { /* Пороговый барьер. */
 HANDLE b_guard; /* Мьютекс для объекта. */
 HANDLE b_broadcast; /* Вручную сбрасываемое событие: b_count >= b_threshold.*/
 volatile DWORD b_destroyed; /* Установить после закрытия. */
 volatile DWORD b_count; /* Количество потоков до достижения барьера. */
 volatile DWORD b_threshold; /* Пороговый барьер. */
} THRESHOLD_BARRIER, *THB_HANDLE;

/* Коды ошибок. */
#define SYNCH_OBJ_NOMEM 1 /* Невозможно выделить ресурсы. */
#define SYNCH_OBJ_BUSY 2 /* Объект используется и не может быть закрыт. */
#define SYNCH_OBJ_INVALID 3 /* Объект более не является действительным. */
DWORD CreateThresholdBarrier(THB_HANDLE *, DWORD /* Порог. */);
DWORD WaitThresholdBarrier(THB_HANDLE);
DWORD CloseThresholdBarrier(THB_HANDLE);
```

Рассмотрим теперь предложенную в программе 10.2 реализацию трех функций. На Web-сайте книги находится тестовая программа testTHB. Обратите внимание на уже знакомый вам цикл проверки переменной условия в функции ожидания WaitThresholdBarrier. Кроме того, эта функция не только ожидает наступления события, но и переводит объект события в сигнальное состояние с помощью функции PulseEvent. Предыдущее обсуждение модели "производитель/потребитель" предполагало использование отдельных функций потоков.

Наконец, в данном случае предикат переменной условия обладает последействием. Как только условие выполнилось, оно будет выполняться и в дальнейшем, что исключает возможность перевода объекта события в сигнальное состояние более одного раза.

Программа 10.2. ThbObject.c: реализация объекта порогового барьера

```
/* Глава 10. Программа 10.2. */
/* Библиотека сложных объектов синхронизации на основе порогового барьера.*/
#include "EvryThng.h"
#include "synchobj.h"
```

```
/**********************************/
/*   ОБЪЕКТЫ ПОРОГОВОГО БАРЬЕРА   */
/**********************************/
DWORD CreateThresholdBarrier(THB_HANDLE *pthb, DWORD b_value) {
 THB_HANDLE hthb;
 /* Инициализация объекта барьера. Вариант программы с полной проверкой ошибок находится на Web-сайте. */
 hthb = malloc(sizeof(THRESHOLD_BARRIER));
 hthb->b_guard = CreateMutex(NULL, FALSE, NULL);
 hthb->b_broadcast = CreateEvent(NULL, FALSE /* Автоматически сбрасываемое событие. */, FALSE, NULL);
 hthb->b_threshold = b_value;
 hthb->b_count = 0;
 hthb->b_destroyed = 0;
 *pthb = hthb;
 return 0;
}

DWORD WaitThresholdBarrier(THB_HANDLE thb) {
 /* Ожидать, пока заданное количество потоков не достигнет порога, а затем установить событие. */
 if (thb->b_destroyed == 1) return SYNCH_OBJ_INVALID;
 WaitForSingleObject(thb->b_guard, INFINITE);
 thb->b_count++; /* Появился новый поток. */
 while (thb->b_count < thb->b_threshold) {
  SignalObjectAndWait(thb->b_guard, thb->b_broadcast, INFINITE, FALSE);
  WaitForSingleObject(thb->b_guard, INFINITE);
 }
 PulseEvent(thb->b_broadcast);
 /* Широковещательная модель CV, освобождение всех ожидающих потоков. */
 ReleaseMutex(thb->b_guard);
 return 0;
}

DWORD CloseThresholdBarrier(THB_HANDLE thb) {
 /* Уничтожить мьютекс и событие объекта барьера. */
 /* Убедиться в отсутствии потоков, ожидающих объект. */
 if (thb->b_destroyed == 1) return SYNCH_OBJ_INVALID;
 WaitForSingleObject(thb->b_guard, INFINITE);
 if (thb->b_count < thb->b_threshold) {
  ReleaseMutex(thb->b_guard);
  return SYNCH_OBJ_BUSY;
 }
 ReleaseMutex(thb->b_guard);
```

```
    CloseHandle(thb->b_guard);
    CloseHandle(thb->b_broadcast);
    free(thb);
    return 0;
    }
```

Комментарии по поводу реализации объекта порогового барьера

Возможности реализованного выше объекта порогового барьера в интересах простоты были намеренно ограничены. Вообще говоря, было бы желательно эмулировать объекты Windows следующим образом:

- Разрешив объектам иметь атрибуты защиты (глава 15).
- Разрешив присвоение имен объектам.
- Допуская наличие у одного объекта нескольких "дескрипторов" и не уничтожая их до тех пор, пока счетчик ссылок не станет равным 0.
- Разрешив совместное использование объекта несколькими процессами.

На Web-сайте доступна полная реализация одного из таких объектов — сложного (multiple-wait) семафора, допускающего изменение счетчика семафора сразу на несколько единиц, которая использует методы, применимые по отношению к любому из объектов, рассматриваемых в данной главе.

Объект очереди

До сих пор мы связывали с каждым мьютексом только одно событие, но в общем случае могут существовать несколько предикатов переменных условий. Например, в случае очереди, действующей по принципу "первым пришел, первым ушел" (first in first out, FIFO), поток, который пытается удалить элемент из очереди, должен дождаться события, указывающего на то, что очередь не является пустой, а поток, помещающий элемент в очередь, должен дождаться события, указывающего на то, что очередь не является заполненной. Решение заключается в предоставлении двух событий — по одному для каждого условия.

В программе 10.3 представлены необходимые объявления объекта очереди и его функций. В объявлениях намеренно применяется стиль, отличающийся от того, который принят в Windows и который мы использовали до сих пор. Эта программа была получена преобразованием ее первоначального варианта, реализованного в UNIX на основе потоков Pthreads, чем и объясняется происхождение использованного нами стиля. Точно так же и вы можете наследовать тот или иной стиль или определить собственный, который соответствует вашему вкусу или принятым в вашей организации требованиям. В упражнении 10.7 вам предлагается преобразовать приведенный стиль к стилю Windows.

Программы 10.4 и 10.5 представляют функции очереди и программу, которая их использует.

Программа 10.3. SynchObj.h: часть 2 — объявления объекта очереди

```
/* Объявления структуры обычной ограниченной синхронизированной очереди.*/
/* Очереди закольцованы и реализованы в виде массивов с индексацией */
/* последнего и первого сообщений. */
/* Кроме того, каждая очередь содержит защитный мьютекс и */
/* переменные условий "очередь не пуста" и "очередь не заполнена". */
/* Наконец, имеется указатель массива сообщений произвольного типа. */
typedef struct queue_tag { /* Универсальная очередь. */
 HANDLE q_guard; /* Защита блока сообщения. */
 HANDLE q_ne; /* Очередь не пуста. Вручную сбрасываемое событие. (Автоматически сбрасываемое событие для "сигнальной модели".) */
 HANDLE q_nf; /* Очередь не заполнена. Вручную сбрасываемое событие. (Автоматически сбрасываемое событие для "сигнальной модели".) */
 volatile DWORD q_size; /* Максимальный размер очереди. */
 volatile DWORD q_first; /* Индекс первого сообщения. */
 volatile DWORD q_last; /* Индекс последнего сообщения. */
 volatile DWORD q_destroyed; /* Получатель сообщений очереди завершил выполнение. */
 PVOID msg_array; /* Массив q_size сообщений. */
} queue_t;

/* Функции управления очередью. */
DWORD q_initialize(queue_t  *, DWORD, DWORD);
DWORD q_destroy(queue_t *);
DWORD q_destroyed(queue_t *);
DWORD q_empty(queue_t *);
DWORD q_full(queue_t  *);
DWORD q_get(queue_t *, PVOID, DWORD, DWORD);
DWORD q_put(queue_t *, PVOID, DWORD, DWORD);
DWORD q_remove(queue_t *, PVOID, DWORD);
DWORD q_insert(queue_t *, PVOID, DWORD);
```

В программе 10.4 представлены такие функции, как q_initialize и q_get, прототипы которых описаны в конце программы 10.3. Обратите внимание, что функции q_get и q_put обеспечивают синхронизацию доступа, а функции q_remove и q_insert, которые вызываются первыми двумя функциями, сами по себе не являются синхронизированными и могут быть использованы в однонитевых программах. В первых двух функциях предусмотрена возможность использования конечных интервалов ожидания, что требует незначительного расширения модели переменных условий.

q_empty и q_full — две другие важные функции, которые используются для реализации предикатов переменных условий.

Данная реализация использует функцию PulseEvent и вручную сбрасываемые события (широковещательная модель), так что все события уведомляются о том, что очередь не пуста или не заполнена.

Замечательной особенностью этой реализации является симметрия функций q_get и q_put. Обратите внимание хотя бы на то, как в этих функциях используются предикаты пустой и заполненной очередей или события. Подобная простота не только восхитительна сама по себе, но и имеет благоприятные практические последствия, облегчающие написание, понимание и сопровождение программы, и все это было достигнуто за счет использования модели переменных условий.

Наконец, те, кто программирует на C++, легко сообразят, что приведенный код может быть использован для создания класса синхронизированной очереди; именно это вам и предлагается сделать в упражнении 10.8.

Программа 10.4. QueueObj.c: функции управления очередью

```
/* Глава 10. QueueObj.c. */
/* Функции очереди */
#include "EvryThng.h"
#include "SynchObj.h"
/* Функции управления конечной ограниченной очередью. */
DWORD q_get(queue_t *q, PVOID msg, DWORD msize, DWORD MaxWait)
{
  if (q_destroyed (q)) return 1;
  WaitForSingleObject(q->q_guard, INFINITE);
  while (q_empty(q)) {
   SignalObjectAndWait(q->q_guard, q->q_ne, INFINITE, FALSE);
   WaitForSingleObject(q->q_guard, INFINITE);
  }
  /* Удалить сообщение из очереди. */
  q_remove(q, msg, msize);
  /* Сигнализировать о том, что очередь не заполнена, поскольку мы
удалили сообщение. */
  PulseEvent(q->q_nf);
  ReleaseMutex(q->q_guard);
  return 0;
 }

DWORD q_put(queue_t *q, PVOID msg, DWORD msize, DWORD MaxWait)
{
  if (q_destroyed(q)) return 1;
  WaitForSingleObject(q->q_guard, INFINITE);
  while(q_full(q)) {
   SignalObjectAndWait(q->q_guard, q->q_nf, INFINITE, FALSE);
   WaitForSingleObject(q->q_guard, INFINITE);
```

```
 }
 /* Поместить сообщение в очередь. */
 q_insert(q, msg, msize);
 /* Сигнализировать о том, что очередь не пуста; мы вставили сообщение.*/
 PulseEvent (q->q_ne);
 /* Широковещательная модель CV. */
 ReleaseMutex(q->q_guard);
 return 0;
}

DWORD q_initialize(queue_t  *q, DWORD msize, DWORD nmsgs) {
 /* Инициализация очереди, включая ее мьютекс и события. */
 /* Выделить память для всех сообщений. */
 q->q_first = q->q_last = 0;
 q->q_size = nmsgs;
 q->q_destroyed = 0;
 q->q_guard = CreateMutex(NULL,  FALSE, NULL);
 q->q_ne = CreateEvent(NULL,  TRUE, FALSE, NULL);
 q->q_nf = CreateEvent(NULL,  TRUE, FALSE, NULL);
 if ((q->msg_array = calloc(nmsgs, msize)) == NULL) return 1;
 return 0; /* Ошибки отсутствуют. */
}

DWORD q_destroy(queue_t *q) {
 if (q_destroyed(q)) return 1;
 /* Освободить все ресурсы, созданные вызовом q_initialize.  */
 WaitForSingleObject(q->q_guard, INFINITE);
 q->q_destroyed = 1;
 free(q->msg_array);
 CloseHandle(q->q_ne);
 CloseHandle(q->q_nf);
 ReleaseMutex(q->q_guard);
 CloseHandle(q->q_guard);
 return 0;
}

DWORD q_destroyed(queue_t *q) {
 return (q->q_destroyed);
}

DWORD q_empty(queue_t *q) {
 return (q->q_first == q->q_last);
}

DWORD q_full(queue_t  *q) {
```

```
   return ((q->q_last – q->q_first) == 1 || (q->q_first == q->q_size-l && q->q_last
== 0));
  }

  DWORD q_remove(queue_t *q, PVOID msg, DWORD msize) {
   char *pm;
   pm = (char *)q->msg_array;
   /* Удалить наиболее давнее ("первое") сообщение. */
   memcpy(msg, pm + (q->q_first * msize), msize);
   q->q_first = ((q->q_first + 1) % q->q_size);
   return 0; /* Ошибки отсутствуют. */
  }

  DWORD q_insert(queue_t *q, PVOID msg, DWORD msize) {
   char *pm;
   pm = (char *)q->msg_array;
   /* Добавить новое ("последнее") сообщение. */
   if (q_full(q)) return 1; /* Ошибка – очередь заполнена. */
   memcpy(pm + (q->q_last * msize), msg, msize);
   q->q_last = ((q->q_last + 1) % q->q_size);
   return 0;
  }
```

Комментарии по поводу функций управления очередью с точки зрения производительности

В приложении В представлены данные, характеризующие производительность программы 10.5, в которой используются функции управления очередью. Приведенные ниже замечания по поводу различных факторов, которые могут оказывать влияние на производительность, основываются на этих данных. Программные коды упоминаемых ниже альтернативных вариантов реализации находятся на Web-сайте книги.

• В данной реализации используется широковещательная модель ("вручную сбрасываемое событие/PulseEvent"), обеспечивающая поддержку общего случая, когда один поток может запрашивать или создавать несколько сообщений. Если такая общность не требуется, можно использовать сигнальную модель ("автоматически сбрасываемое событие/SetEvent"), которая, к тому же, обеспечит значительно более высокую производительность, поскольку для тестирования предиката будет освобождаться только один поток. На Web-сайте находится файл QueueObj_Sig.c, содержащий исходный код, в котором вместо широковещательной модели используется сигнальная модель.

• Использование для защиты объекта очереди объекта CRITICAL_SECTION вместо мьютекса также может привести к повышению производительности. Однако в этом случае вместо функции

SignalObjectAndWait следует использовать функцию EnterCriticalSection с последующим ожиданием события. Этот альтернативный подход иллюстрируется двумя файлами — QueueObjCS.c и QueueObjCS_Sig.c, находящимися на Web-сайте книги.

• На Web-сайте находятся два других файла с исходными кодами — QueueObj_noSOAW.c и QueueObjSig_noSOAW.c, в которых функция SignalObjectAndWait не используется и которые обеспечивают выполнение программы под управлением Windows 9x.

• Результаты, приведенные в приложении B, свидетельствуют о нелинейном поведении производительности при большом количестве потоков, состязающихся за доступ к очереди. Проекты для каждой из альтернативных стратегий содержатся на Web-сайте книги; эти проекты соответствуют различным вариантам конвейерной системы ThreeStage, описанной в следующих разделах.

•Резюмируя, следует подчеркнуть, что свойства очередей могут быть расширены таким образом, чтобы очередь могла совместно использоваться несколькими процессами и обеспечивать отправку или получение сразу нескольких сообщений за одну операцию. В то же время, некоторого выигрыша в производительности можно добиться за счет использования сигнальной модели, объектов CRITICAL_SECTIONS или функции SignalObjectAndWait. Соответствующие результаты представлены в приложении B.

Пример: использование очередей в многоступенчатом конвейере

Модель "хозяин/рабочий", во всех ее вариациях, является одной из наиболее популярных моделей многопоточного программирования, а программа 8.2 представляет простую модель "производитель/потребитель", являющуюся частным случаем более общей конвейерной модели (pipeline model).

В другом важном частном случае имеется один главный поток, который производит единичные рабочие задания (work units) для ограниченного количества рабочих потоков и помещает их в очередь. Такая методика может оказаться полезной при создании масштабируемого сервера с большим количеством клиентов (число которых может достигать тысячи и более), когда возможность выделения независимого рабочего потока для каждого клиента весьма сомнительна. В главе 14 задача создания масштабируемого сервера обсуждается в контексте портов завершения ввода/вывода.

В конвейерной модели каждый поток или группа потоков определенным образом обрабатывает единичные задания, например, сообщения, и передает их другим потокам для дальнейшей обработки. Аналогом многопоточного конвейера может служить производственная сборочная линия. Идеальным механизмом реализации конвейера являются очереди.

В программе 10.5 (ThreeStage.c) предусмотрено создание нескольких этапов производства и потребления, на каждой из которых поддерживается

очередь рабочих заданий, подлежащих обработке. Каждая очередь имеет ограниченную, конечную длину. Всего существует три конвейерных ступени, соединяющих четыре этапа обработки. Программа имеет следующую структуру:

• Производители (producers) периодически создают единичные сообщения, дополненные контрольными суммами, используя для этого ту же функцию, что и в программе 8.2, если не считать того, что в каждом сообщении содержится дополнительное поле адресата, указывающее поток потребителя (consumer), для которой предназначено это сообщение, причем каждый производитель связывается только с одним потребителем. Количество пар "производитель/потребитель" задается в виде параметра командной строки. Далее производитель посылает одиночное сообщение передающему потоку (transmitter), помещая его в очередь передачи сообщений. Если очередь заполнена, производитель ждет, пока ее состояние не изменится.

• Передающий поток объединяет имеющиеся единичные сообщения (но не более пяти за один раз) и создает одно передаваемое сообщение, которое содержит заголовок и ряд единичных сообщений. Затем передающий поток помещает каждое передаваемое сообщение в очередь приема сообщений (receiver), блокируясь, если очередь заполнена. В общем случае передатчик и приемник могут связываться между собой через сетевое соединение. Произвольно выбранное здесь значение коэффициента блокирования (blocking factor), равное 5:1, легко поддается регулировке.

• Принимающий поток обрабатывает единичные сообщения, входящие в состав каждого передаваемого сообщения, и помещает каждое из них в соответствующую очередь потребителя, если она не заполнена.

• Каждый поток потребителя получает одиночные сообщения по мере их поступления и записывает сообщение в файл журнала регистрации.

Блок-схема системы представлена на рис. 10.1. Обратите внимание, что эта система моделирует сетевое соединение, в котором сообщения, относящиеся к различным парам "отправитель/получатель" объединяются и передаются по общему каналу связи.

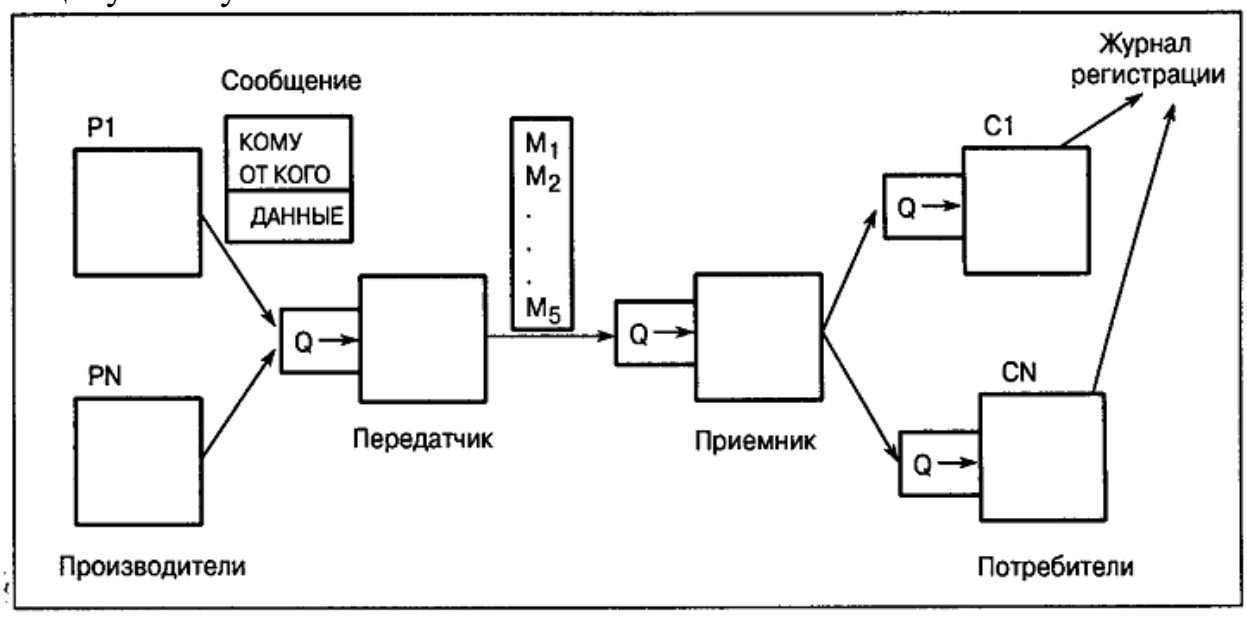

Рис. 10.1. Многоступенчатый конвейер

В программе 10.5 предложен вариант реализации, в котором используются функции очереди из программы 10.4. Функции генерации и отображения сообщений здесь не представлены, но они взяты из программы 8.1. При этом, наряду с контрольными суммами и данными, в блоки сообщений введены поля производителя и адресата.

Программа 10.5. ThreeStage.c: многоступенчатыйконвейер

```
/* Глава 10. ThreeStage.c */
/* Трехступенчатая система производитель/потребитель. */
/* Использование: ThreeStage npc goal. */
/* Запустить "npc" пар потоков производителя и потребителя. */
/* Каждый производитель должен сгенерировать в общей сложности */
/* "goal" сообщений, каждое из которых снабжается меткой, указывающей */
/* потребителя, для которого оно предназначено. */
/* Сообщения отправляются "передающему потоку", который, прежде чем */
/* отправить группу сообщений "принимающему потоку", выполняет некоторую*/
/* дополнительную обработку. Наконец, принимающий поток отправляет сообщения потокам потребителя. */

#include "EvryThng.h"
#include "SynchObj.h"
#include "messages.h"
#include <time.h>

#define DELAY_COUNT 1000
#define MAX_THREADS 1024

/* Размеры и коэффициенты блокирования очередей. Эти величины являются */
/* произвольными и могут регулироваться для обеспечения оптимальной */
/* производительности. Текущие значения не являются сбалансированными. */
#define TBLOCK_SIZE 5 /*Передающий поток формирует группы из 5 сообщений.*/
#define TBLOCK_TIMEOUT 50 /*Интервал ожидания сообщений передающим потоком.*/
#define P2T_QLEN 10 /* Размер очереди "производитель/передающий поток". */
#define T2R_QLEN 4 /*Размер очереди "передающий поток/принимающий поток".*/
```

```
  #define R2C_QLEN 4 /* Размер очереди "принимающий поток/потребитель" -- */
  /* для каждого потребителя существует только одна очередь.*/

  DWORD WINAPI producer(PVOID);
  DWORD WINAPI consumer(PVOID);
  DWORD WINAPI transmitter(PVOID);
  DWORD WINAPI receiver(PVOID);

  typedef struct _THARG {
   volatile DWORD thread_number;
   volatile DWORD work_goal; /* Используется потоками производителей. */
   volatile DWORD work_done; /* Используется потоками производителей и потребителей. */ '
   char future[8];
  } THARG;

  /* Сгруппированные сообщения, посылаемые передающим потоком потребителю.*/
  typedef struct t2r_msg_tag {
   volatile DWORD num_msgs; /* Количество содержащихся сообщений. */
   msg_block_t messages[TBLOCK_SIZE];
  } t2r_msg_t;

  queue_t p2tq, t2rq, *r2cq_array;

  static volatile DWORD ShutDown = 0;
  static DWORD EventTimeout = 50;

  DWORD _tmain(DWORD argc, LPTSTR * argv[]) {
   DWORD tstatus, nthread, ithread, goal, thid;
   HANDLE *producer_th, *consumer_th, transmitter_th, receiver_th;
   THARG *producer_arg, *consumer_arg;
   nthread = atoi(argv[1]);
   goal = atoi(argv[2]);
   producer_th = malloc(nthread * sizeof(HANDLE));
   producer_arg = calloc(nthread, sizeof(THARG));
   consumer_th = malloc(nthread * sizeof(HANDLE));
   consumer_arg = calloc(nthread, sizeof(THARG));
   q_initialize(&p2tq, sizeof(msg_block_t), P2T_QLEN);
   q_initialize(&t2rq, sizeof(t2r_msg_t), T2R_QLEN);
   /* Распределить ресурсы, инициализировать очереди "принимающий поток/потребитель" для каждого потребителя. */
   r2cq_array = calloc(nthread, sizeof(queue_t));
   for (ithread = 0; ithread < nthread; ithread++) {
```

```
    /* Инициализировать очередь r2c для потока данного потребителя. */
    q_initialize(&r2cq_array[ithread], sizeof(msg_block_t), R2C_QLEN);
    /* Заполнить аргументы потока. */
    consumer_arg[ithread].thread_number = ithread;
    consumer_arg[ithread].work_goal = goal;
    consumer_arg[ithread].work_done = 0;
      consumer_th[ithread] = (HANDLE)_beginthreadex(NULL, 0, consumer, (PVOID)&consumer_arg[ithread], 0, &thid);
    producer_arg[ithread].thread_number = ithread;
    producer_arg[ithread].work_goal = goal;
    producer_arg[ithread].work_done = 0;
      producer_th[ithread] = (HANDLE)_beginthreadex(NULL, 0, producer, (PVOID)&producer_arg[ithread], 0, &thid);
   }
   transraitter_th = (HANDLE)_beginthreadex(NULL, 0, transmitter, NULL, 0, &thid);
   receiver_th = (HANDLE)_beginthreadex (NULL, 0, receiver, NULL, 0, &thid);
   _tprintf(_T("ХОЗЯИН: Выполняются все потоки\n"));
   /* Ждать завершения потоков производителя. */
   for (ithread = 0; ithread < nthread; ithread++) {
    WaitForSingleObject(producer_th[ithread], INFINITE);
     _tprintf(_T("ХОЗЯИН: производитель %d выработал %d единичных сообщений\n"), ithread, producer_arg[ithread].work_done);
   }
   /* Производители завершили работу. */
   _tprintf(_T("ХОЗЯИН: Все потоки производителя выполнили свою работу.\n"));
   /* Ждать завершения потоков потребителя. */
   for (ithread = 0; ithread < nthread; ithread++) {
    WaitForSingleObject(consumer_th[ithread], INFINITE);
     _tprintf(_T("ХОЗЯИН: потребитель %d принял %d одиночных сообщений\n"), ithread, consumer_arg[ithread].work_done);
   }
   _tprintf(_T("ХОЗЯИН: Все потоки потребителя выполнили свою работу.\n"));
   ShutDown = 1; /* Установить флаг завершения работы. */
   /* Завершить выполнение и перейти в состояние ожидания передающих и принимающих потоков. */
   /* Эта процедура завершения работает нормально, поскольку и передающий,*/
   /* и принимающий потоки не владеют иными ресурсами, кроме мьютекса, */
   /* которые они могли бы покинуть по завершении выполнения, не уступив прав владения ими. Можете ли вы улучшить эту процедуру? */
   TerminateThread(transmitter_th, 0);
```

```
 TerminateThread(receiver_th, 0);
 WaitForSingleObject(transmitter_th, INFINITE);
 WaitForSingleObject(receiver_th, INFINITE);
 q_destroy(&p2tq);
 q_destroy(&t2rq);
 for (ithread = 0; ithread < nthread; ithread++) q_destroy(&r2cq_array [ithread]);
 free(r2cq_array);
 free(producer_th);
 free(consumer_th);
 free(producer_arg);
 free(consumer_arg);
 _tprintf(_T("Система завершила работу. Останов системы\n"));
 return 0;
}

DWORD WINAPI producer(PVOID arg) {
 THARG * parg;
 DWORD ithread, tstatus;
 msg_block_t msg;
 parg = (THARG *)arg;
 ithread = parg->thread_number;
 while (parg->work_done < parg->work_goal) {
  /* Вырабатывать единичные сообщения, пока их общее количество */
  /* не станет равным "goal". */
  /* Сообщения снабжаются адресами отправителя и адресата, которые в */
  /* нашем примере одинаковы для всех сообщений, но в общем случае */
  /* могут быть различными. */
  delay_cpu(DELAY_COUNT * rand() / RAND_MAX);
  message_fill(&msg, ithread, ithread, parg->work_done);
  /* Поместить сообщение в очередь. */
  tstatus = q_put(&p2tq, &msg, sizeof(msg), INFINITE);
  parg->work_done++;
 }
 return 0;
}

DWORD WINAPI transmitter(PVOID arg) {
 /* Получись несколько сообщений от производителя, объединяя их в
одно*/
 /* составное сообщение, предназначенное для принимающего потока. */
 DWORD tstatus, im;
 t2r_msg_t t2r_msg = {0};
 msg_block_t p2t_msg;
 while (!ShutDown) {
  t2r_msg.num_msgs = 0;
```

```
 /* Упаковать сообщения для передачи принимающему потоку. */
 for (im = 0; im < TBLOCK_SIZE; im++) {
  tstatus = q_get(&p2tq, &p2t_msg, sizeof(p2t_msg), INFINITE);
  if (tstatus != 0) break;
  memcpy(&t2r_msg.messages[im], &p2t_msg, sizeof(p2t_msg));
  t2r_rasg.num_msgs++;
 }
 tstatus = q_put(&t2rq, &t2r_msg, sizeof(t2r_msg), INFINITE);
 if (tstatus != 0) return tstatus;
}
return 0;
}

DWORD WINAPI receiver(PVOID arg) {
 /* Получить составные сообщения от передающего потока; распаковать */
 /* их и передать соответствующему потребителю. */
 DWORD tstatus, im, ic;
 t2r_msg_t t2r_msg;
 msg_block_t r2c_msg;
 while (!ShutDown) {
  tstatus = q_get(&t2rq, &t2r_msg, sizeof(t2r_msg), INFINITE);
  if (tstatus != 0) return tstatus;
  /* Распределить сообщения между соответствующими потребителями. */
  for (im = 0; im < t2r_msg.num_msgs; im++) {
   memcpy(&r2c_msg, &t2r_msg.messages[im], sizeof(r2c_msg));
   ic = r2c_msg.destination; /* Конечный потребитель. */
   tstatus = q_put(&r2cq_array[ic], &r2c_msg, sizeof(r2c_msg), INFINITE);
   if (tstatus != 0) return tstatus;
  }
 }
 return 0;
}

DWORD WINAPI consumer(PVOID arg) {
 THARG * carg;
 DWORD tstatus, ithread;
 msg_block_t msg;
 queue_t *pr2cq;
 carg = (THARG *)arg;
 ithread = carg->thread_number;
 carg = (THARG *)arg;
 pr2cq = &r2cq_array[ithread];
 while (carg->work_done < carg->work_goal) {
  /* Получить и отобразить (необязательно — не показано) сообщения. */
  tstatus = q_get(pr2cq, &msg, sizeof(msg), INFINITE);
```

```
 if (tstatus != 0) return tstatus;
 carg->work_done++;
 }
 return 0;
}
```

Комментарии по поводу многоступенчатого конвейера

Данная реализация характеризуется некоторыми особенностями, суть которых частично отражена в комментариях, включенных в листинг программы. На эти же особенности обращают ваше внимание и упражнения 10.6, 10.7 и 10.10.

• Значительные возражения вызывает способ, используемый основным потоком для завершения выполнения передающего и принимающего потоков. Лучшим решением было бы использование конечных интервалов ожидания во внутренних циклах передатчика и приемника и прекращение работы после того, как будет установлен соответствующий глобальный флаг. Другой возможный подход заключается в отмене выполнения потоков, как описано далее в этой главе.

• Обратите внимание на существование симметрии между передающим и принимающим потоками. Как и при реализации очереди, это обстоятельство упрощает проектирование, отладку и сопровождение программы.

• Реализация не сбалансирована в смысле согласования скорости генерации сообщений, емкости конвейера и коэффициента блокирования "передатчик/приемник".

• В данной реализации (программа 10.4) для защиты очередей используются мьютексы. Результаты экспериментов с объектами CRITICAL_SECTION не позволили обнаружить сколько-нибудь заметного ускорения работы программы на однопроцессорной системе (см. приложение В). CS-версия программы, ThreeStageCS, находится на Web-сайте. Аналогичным образом вела себя программа и после того, как в ней была использована функции SignalObjectAndWait.

Асинхронные вызовы процедур

Основное возражение, которое можно предъявить к программе ThreeSage.c (программа 10.5) в ее нынешнем виде, касается прекращения выполнения передающего и принимающего потоков с помощью функции TerminateThread. В комментариях, включенных в код, вам предлагается подумать над более элегантным способом завершения выполнения потоков, который обеспечивал бы корректное прекращение работы программы и освобождение ресурсов.

Другой нерешенной проблемой является отсутствие общего метода (не считая использования функции TerminateThread), который обеспечивал бы отправку сигнала определенному потоку или инициировал его выполнение.

События могут посылать сигналы одному потоку, ожидающему наступления автоматически сбрасываемого события, или всем потокам, ожидающим наступления вручную сбрасываемого события, но невозможно добиться того, чтобы сигнал был получен определенным потоком. Используемое до сих пор решение сводилось к тому, что пробуждались все ожидающие потоки, которые самостоятельно определяли, могут ли они теперь продолжить свое выполнение. Иногда привлекается альтернативное решение, суть которого состоит в назначении событий определенным потокам, так что сигнализирующий поток может определять, объект какого события следует перевести в сигнальное состояние одной из функций SetEvent или PulseEvent.

Обе эти проблемы решаются путем использования объектов асинхронного вызова процедур (Asynchronous Procedure Call, APC). События развиваются в следующей последовательности, причем рабочий или целевой потоки должны управляться главным потоком.

• Главный поток указывает APC-функцию данной целевого потока путем помещения объекта APC в очередь APC данного потока. В очередь могут быть помещены несколько APC.

• Целевой поток переходит в состояние дежурного ожидания (alertable wait state), обеспечивающее возможность безопасного выполнения потоком APC. Порядок первых двух шагов безразличен, поэтому о возникновении условий состязательности можно не беспокоиться.

• Указанный поток, находящийся в состоянии ожидания, выполняет все APC, находящиеся в очереди.

• APC могут выполнять любые нужные действия, например, освобождать ресурсы или генерировать исключения. Благодаря этому главный поток может инициировать возбуждение исключения в целевом потоке, хотя само исключение не произойдет до тех пор, пока целевой поток не перейдет в состояние дежурного ожидания.

Выполнение APC является асинхронным в том смысле, что APC может быть помещен в очередь целевого потока в любой момент, но само выполнение является синхронизированным в том смысле, что это может произойти лишь тогда, когда целевой поток входит в состояние дежурного ожидания.

Состояния дежурного ожидания будут вновь обсуждаться в главе 14, посвященной асинхронному вводу/выводу.

Описания необходимых функций и примеры их использования в другом варианте программы ThreeStage приводятся в следующих разделах. Проект для построения новой версии программы (ThreeStageCancel) и соответствующий исходный код (ThreeStageCancel. c) находятся на Web-сайте книги.

Очередизация асинхронных вызовов процедур

Один поток (главный) помещает APC в очередь целевого потока с помощью функции QueueUserAPC:

DWORD QueueUserAPC(PAPCFUNC pfnAPC, HANDLE hThread, DWORD dwData)

hThread — дескриптор целевого потока. dwData — аргумент, который будет передан функции APC при ее выполнении и может являться кодом завершения или сообщать функции иную информацию.

В основной функции программы ThreeStageCancel.c (сравните с программой 10.5) вызовы TerminateThread заменяются вызовами QueueUserAPC следующим образом:

```
// TerminateThread(transmitter_th,  0) ; заменить на APC
// TerminateThread(receiver_th,  0); заменить на APC
tstatus = QueueUserAPC(ShutDownTransmitter, transmitter_th, 1);
if (tstatus == 0) ReportError(…);
tstatus = QueueUserAPC(ShutDownReceiver, receiver_th, 2);
if (tstatus == 0) ReportError (…);
```

Функция QueueUserAPC в случае успешного ее завершения возвращает ненулевое значение, иначе — нуль. В то же время, функция GetLastError () не возвращает никакого полезного значения, и поэтому при вызове функции ReportError не требуется задавать текст сообщения об ошибке (значением последнего аргумента является FALSE).

pfnAPC — указатель на фактическую функцию, вызываемую целевым потоком, как показывает следующий фрагмент, взятый из программы ThreeStageCancel. c:

```
/* APC для завершения выполнения потребителя. */
void WINAPI ShutDownReceiver(DWORD n) {
 printf("Внутри ShutDownReceiver. %d\n", n);
 /* Освободить все ресурсы (в данном примере отсутствуют). */
 return;
}
```

Функция ShutDownTransmitter аналогична вышеприведенной, отличаясь от нее только текстом сообщения. Сразу трудно понять, каким образом эта функция, которая, казалось бы, не выполняет никаких существенных операций, может инициировать прекращение выполнения целевого принимающего потока. Соответствующие пояснения приводятся далее в этой главе.

APC и упущенные сигналы

APC, выполняемые в режиме ядра (используются в операциях асинхронного ввода/вывода), могут немедленно выводить ожидающий поток из состояния ожидания, что может стать причиной потери сигналов PulseEvent. В связи с этим в документации можно встретить советы, в которых функции PulseEvent рекомендуется не использовать, хотя, как было продемонстрировано в данной главе, они могут и приносить пользу. Применение функции PulseEvent в наших примерах было вполне безопасным, поскольку APC, выполняемые в режиме ядра, в них не

используются. Кроме того, применение функции SignalObjectAndWait и тестирование возвращаемого ею значения обеспечивает достаточно надежную защиту от подобных потерь сигналов. Наконец, если вы опасаетесь, что это все-таки может случиться, просто включайте указание конечного интервала ожидания в соответствующие вызовы функций ожидания.

Состояния дежурного ожидания

Во всех предыдущих примерах значение параметра bAlertable, являющегося последним параметром функции SignalObjectAndWait, полагалось равным FALSE. Используя вместо него значение TRUE, мы указываем, что ожидание должно быть, как говорят, *дежурным* (alertable), и тогда после выполнения функции поток перейдет в состояние дежурного ожидания. В этом состоянии поток ведет себя следующим образом:

• Если один или более APC помещаются в очередь потока (указанного в качестве целевого при вызове функции QueueUserAPC) еще до того, как либо объект, указываемый дескриптором hObjectToWaitOn (обычно таким объектом является событие), перейдет в сигнальное состояние, либо истечет интервал ожидания, то все эти потоки выполнятся (при этом не гарантируется какой-то определенный порядок их выполнения), а функция SignalObjectAndWait завершит выполнение, возвращая значение WAIT_IO_COMPLETED.

• Если APC в очередь не помещались, то функция SignalObjectAndWait ведет себя обычным образом, то есть ожидает перехода объекта в сигнальное состояние или истечения интервала ожидания.

Состояния дежурного ожидания будут вновь использоваться нами при выполнении операций асинхронного ввода/вывода (глава 14); именно в связи с этими операциями и получило свое название значение WAIT_IO_COMPLETED. В состояние дежурного ожидания потока можно переводить также с помощью функций WaitForSingleObjectEx, WaitForMultipleObjectsEx и SleepEx, которые оказываются полезными и при выполнении операций асинхронного ввода/вывода.

Теперь можно изменить функции q_get и q_put (см. программу 10.4) таким образом, чтобы завершение работы программы после выполнения APC было корректным, хотя APC-функция и не выполняет никаких иных действий, кроме вывода сообщения и возврата из функции. Все, что в данном случае требуется — это организовать вход в состояние дежурного ожидания и проверить значение, возвращаемое функцией SignalObjectAndWait, как показано в приведенной ниже видоизмененной версии функции q_get (см. файл QueueObjCancel.c, находящийся на Web-сайте).

Программа 10.6. Модифицированная функция q_get, обеспечивающая корректное завершение выполнения потоков

```
DWORD q_put(queue_t *q, PVOID msg, DWORD msize, DWORD MaxWait)
{
  BOOL Cancelled = FALSE;
  if (q_destroyed(q)) return 1;
  WaitForSingleObject(q->q_guard, INFINITE);
  while (q_full(q) && !Cancelled) {
    if (SignalObjectAndWait(q->q_guard, q->q_nf, INFINITE, TRUE) ==
WAIT_IO_COMPLETION) {
     Cancelled = TRUE;
     continue;
    }
    WaitForSingleObject(q->q_guard, INFINITE);
  }
  /* Поместить сообщение в очередь. */
  if (!Cancelled) {
   q_remove(q, msg, msize);
    /* Сигнализировать о том, что очередь не заполнена, поскольку мы
удалили сообщение. */
    PulseEvent(q->q_nf);
    ReleaseMutex(q->q_guard);
  }
  return Cancelled ? WAIT_TIMEOUT : 0;
 }
```

В качестве функции APC могут выступать и функция ShutDownReceiver, и функция ShutDownTransmitter, поскольку приемник и передатчик используют как функцию q_get, так и функцию q_put. Если требуется, чтобы функциям завершения было известно, из какого потока они выполняются, применяйте различные значения для аргументов функций APC, которые передаются третьим аргументом функции QueueUserAPC во фрагменте кода, предшествующем программе 10.6.

Чтобы обеспечить согласованность с предыдущими версиями программы, в качестве кода завершения следует использовать значение WAIT_TIMEOUT.

В качестве альтернативного варианта вместо проверки совпадения возвращаемого значения со значением WAIT_IO_COMPLETION можно предусмотреть генерацию исключения функциями завершения и поместить тело функции q_put в try-блок, дополнив программу обработчиком исключений.

Безопасная отмена выполнения потоков

Обсуждение предыдущего примера продемонстрировало, как безопасно отменить выполнение целевого потока, который использует состояния дежурного ожидания. Несмотря на использование APC, такую отмену выполнения иногда называют *синхронной отменой* (synchronous cancellation), поскольку отмена выполнения, которую инициировал вызов функции

QueueUserAPC главным потоком, сможет осуществиться лишь тогда, когда целевой поток достигнет безопасного состояния дежурного ожидания.

Синхронная отмена выполнения требует участия в этом целевого потока, которая время от времени должна предоставлять возможность прекратить ее выполнение. Естественный способ вхождения в состояние дежурного ожидания предоставляют функции ожидания событий, поскольку в процессе прекращения работы системы объект события может не перейти вновь в сигнальное состояние. Ожидание мьютексов также можно выполнять в дежурном режиме, чтобы учесть те случаи, когда поток ожидает ресурса, который, возможно, не будет вновь доступным. Этот метод, например, может использоваться главным потоком для разрушения взаимной блокировки потоков.

Асинхронная отмена выполнения потоков (asynchronous thread cancellation) может применяться в тех случаях, когда сигнал должен посылаться потоку, который выполняет интенсивные вычисления и находится в состоянии ожидания ввода/вывода или событий исключительно редко, если это вообще происходит. Возможность асинхронной отмены выполнения потоков в Windows отсутствует, хотя и существуют методики, использующие зависящий от типа процессора код, которые позволяют прерывать выполнение определенного потока.

Создание переносимых приложений с использованием потоков Pthreads

Потоки Pthreads уже неоднократно упоминались нами в качестве альтернативной модели многопоточного программирования и синхронизации, доступной в UNIX, Linux и других системах, не принадлежащих семейству Windows. Существует библиотека Windows Pthreads с открытым исходным кодом, используя которую можно создавать переносимые многопоточные приложения, способные выполняться на самых различных системах. Более подробное обсуждение этого вопроса вы найдете на Web-сайте книги. Указанная библиотека с открытым исходным кодом применяется в проекте ThreeStagePthreads, в котором также предоставляется соответствующая ссылка на сайт загрузки.

Стеки потоков и допустимые количества потоков

Следует сделать еще два предостережения. Во-первых, подумайте о размере стека, который по умолчанию составляет 1 Мбайт. В большинстве случаев этого будет вполне достаточно, но если существуют какие-либо сомнения на сей счет, оцените максимальный объем стекового пространства, которое требуется для каждого потока с учетом всех библиотечных и рекурсивных функций, которые вызываются потоком. Переполнение стека может привести к порче памяти или вызвать исключение.

Во-вторых, использование большого количества потоков с большими размерами стеков потребует больших объемов виртуальной памяти для их

обработки и может оказать отрицательное влияние на процессы страничного обмена и состояние файла подкачки. Так, использовать свыше 1000 потоков в некоторых из примеров, приведенных в этой и последующей главах, было бы неразумно. При размере стека 1 Мбайт на один поток для этого потребовалось бы виртуальное адресное пространство объемом 1 Гбайт. Соответствующие меры предосторожности включают тщательное планирование размеров стеков, использование портов завершения ввода/вывода и мультиплексирование операций в пределах одного потока.

Рекомендации по проектированию, отладке и тестированию программ

Рискуя дать совет, противоречащий высказываниям во многих других книгах и технических статьях, в которых основной упор делается на тестировании и уже затем рассматривается все остальное, лично я порекомендовал бы вам распределить свои усилия таким образом, чтобы нашлось время для ознакомления с особенностями проектирования, реализации и использования известных моделей программирования. Лучший метод отладки заключается, прежде всего, в недопущении ошибок; разумеется, такие советы легче давать, чем им следовать. Тем не менее, когда начинают проявляться программные дефекты — а это происходит всегда – наиболее эффективным методом обнаружения и исправления основных причин дефектов является тщательное исследование кода в сочетании с отладкой.

Не возлагайте слишком большие надежды на тестирование программ, ибо каким бы тщательным и обширным оно ни было, многие серьезные дефекты от вас все равно ускользнут. Тестирование способно лишь выявлять дефекты; оно не может служить доказательством отсутствия дефектов и указывает лишь на их симптомы, а не на причины, которые их породили. Для наглядности могу сослаться на личный опыт, приведя в качестве примера версию программы, включающую функцию ожидания сложного семафора, построенную на основе модели CV, в которой конечный интервал ожидания не использовался. Дефект, который мог приводить к блокированию потока на неопределенное время, никак не проявлял себя на протяжении года, пока, в конечном счете, что-то пошло не так, как надо. Мне удалось обнаружить ошибку путем простого просмотра кода, благодаря пониманию принципов, лежащих в основе работы модели переменных условий.

Не следует переоценивать и помощь, которую вам может оказать отладка, поскольку отладчики изменяют временное поведение программы, маскируя возникновение условий состязательности, то есть именно то, что вы хотели бы исследовать. Так, вряд ли можно ожидать, что с помощью отладчика вам удастся выявить проблемы, обусловленные неправильным выбором типа события (автоматически сбрасываемым или сбрасываемым вручную) или функции (SetEvent или PulseEvent). Всегда тщательно продумывайте, чего именно вы хотите добиться, применяя те или иные средства.

В заключение следует подчеркнуть, что тестирование программ с использованием как можно более широкого круга платформ, включая SMP, составляет важнейшую часть любого проекта, целью которого является разработка многопоточного программного обеспечения.

Как избежать создания некорректного программного кода

Каждая ошибка, не допущенная вами в исходном коде, — это, прежде всего, еще одна сэкономленная ошибка, которую вам не придется отыскивать на стадии отладки программы или в процессе проверки работоспособности ее завершенной версии. Ниже приведены некоторые рекомендации, большинство из которых взяты, хотя и в перефразированном виде, из [6].

• **Не полагайтесь на инерционность потоков.** Потоки асинхронны, но мы, например, часто предполагаем, что после создания родительским потоком одного или нескольких дочерних потоков он продолжает свое выполнение. В основе таких предположений лежит допущение об "инерционных" свойствах родительского потока, благодаря которым он, якобы, будет выполняться вплоть до начала выполнения дочерних потоков. Предположения подобного рода особенно опасны в случае SMP-систем, тем не менее, они могут привести к возникновению проблем и в случае однопроцессорных систем.

• **Никогда не делайте ставок на состязании потоков.** Об очередности выполнения потоков практически никогда нельзя сказать ничего определенного. В программе должно предполагаться, что любой готовый к выполнению поток может быть запущен в любой момент и что любой выполняющийся поток в любой момент может быть вытеснен. Никакого упорядочения выполнения потоков не существует, если только вы специально об этом не позаботились.

• **Не следует путать планирование выполнения с синхронизацией.** Стратегии планирования задач и назначения приоритетов не в состоянии обеспечить нужную синхронизацию. Для этого должны использоваться объекты синхронизации.

• **Состязательность за очередность выполнения будет существовать даже при использовании мьютексов для защиты разделяемых данных.** Наличие защиты данных само по себе не может служить гарантией определенной очередности получения доступа к разделяемым данным различными потоками. Например, если один поток добавляет средства на банковский счет, а другой снимает их со счета, то одна только защита счета с помощью мьютекса не сможет гарантировать, что внесение денег на счет всегда будет осуществляться раньше, чем их снятие. В упражнении 10.14 показано, как организовать управление очередностью выполнения потоков.

• **Необходимость кооперирования потоков во избежание взаимной блокировки.** Чтобы гарантировать невозможность взаимоблокировки потоков, вы должны располагать хорошо продуманной иерархией блокировок, которая использовалась бы всеми потоками.

• **Не допускайте разделения событий предикатами.** В реализациях, использующих переменные условий, каждое событие должно связываться с отдельным предикатом. Кроме того, событие всегда должно использоваться с одним и тем же мьютексом.

• **Остерегайтесь совместного использования стеков и связанного с этим риска порчи памяти.** Никогда не забывайте о том, что при возврате из функции или завершении выполнения потока их локальные области памяти становятся недействительными. Память в области стека потока может использоваться другими потоками, но вы должны быть уверены в том, что первый поток все еще существует.

• **Следите за использованием модификатором класса памяти volatile в необходимых случаях.** Всякий раз, когда возможно изменение разделяемой переменной в одном потоке и обращение к нему в другом, класс памяти, используемой этой переменной, должен быть определен как volatile, что будет гарантировать использование ячеек памяти при сохранении и извлечении этой переменной потоками, а не регистров, специфичных для каждого потока.

Ниже приводятся некоторые дополнительные рекомендации и мнемонические правила, которые вам могут пригодиться.

• **Правильно пользуйтесь моделью переменных условий,** проверяя, чтобы два разных мьютекса не использовались с одним и тем же событием. Хорошо изучите модель переменных условий, на которой основана ваша программа. Прежде чем вызывать функцию ожидания перехода переменной условия в сигнальное состояние, убедитесь в выполнении условия, выраженного предикатом.

• **Старайтесь вникнуть в смысл используемых инвариантов и предикатов переменных условий,** даже если они имеют неформальное выражение. Убедитесь в том, что условие, выраженное предикатом, всегда выполняется за пределами критического участка кода.

• **Стремитесь к упрощению.** Многопоточное программирование уже само по себе является достаточно сложным, чтобы еще дополнительно усложнять его введением трудно воспринимаемых моделей и логики. Если ваша программа становится чрезмерно сложной, постарайтесь оценить, продиктована ли эта сложность действительной необходимостью или же является следствием того, что программа была неудачно спроектирована.

• **Тестируйте программы как на однопроцессорных, так и на многопроцессорных системах, а также на системах с различными тактовыми частотами и другими характеристиками.** Природа некоторых дефектов такова, что на однопроцессорных системах они проявляются лишь в редких случаях или вообще никогда, в то время как на SMP-системах вы их сразу определите, и наоборот. Аналогичным образом, чем шире круг характеристик систем, на которых будет выполняться программа, содержащая дефекты, тем выше вероятность сбоя.

• **Тестирование является необходимой, но не достаточной мерой, которая могла бы гарантировать корректную работу программы.**

Известны многочисленные примеры программ, заведомо содержащих дефекты, которые лишь в редких случаях удавалось обнаруживать средствами обычного и даже расширенного тестирования.

• **Смиритесь!** Как бы вы ни старались следовать этим советам, ошибок в своих программах вам не избежать. Это утверждение справедливо даже в случае однопоточных программ, не говоря уже о многопоточных, которые предоставляют нам гораздо больше разнообразных и интереснейших возможностей создавать себе проблемы.

За рамками Windows API

В своем рассмотрении мы намеренно ограничились случаем Windows API. Вместе с тем, Microsoft предоставляет дополнительные средства доступа к таким объектам ядра, как потоки. Так, класс ThreadPool, доступный в C++, C# и других языках программирования, позволяет создавать пулы потоков и очереди задач потоков (для этого служит метод QueueUserWorkItem класса ThreadPool).

Кроме того, Microsoft реализует службу Microsoft Message Queuing (MSMQ), которая предоставляет услуги по передаче сообщений между сетевыми системами. Приведенный в данной главе пример должен был продемонстрировать вам, насколько полезными могут быть универсальные системы очередизации сообщений. MSMQ документирована на Web-сайте компании Microsoft.

Резюме

Разработка многопоточных программ значительно упрощается, если используются хорошо себя зарекомендовавшие, известные модели и методы программирования. В этой главе была продемонстрирована полезность модели переменных условий и решен ряд сравнительно сложных, но важных проблем программирования. APC позволяют одному потоку посылать сигналы другому и вызывать в нем выполнение определенных действий, что позволяет отменять выполнение потоков таким образом, чтобы все потоки в системе имели возможность корректно завершиться.

Сложность синхронизации и управления потоками объясняется тем, что существует множество путей решения одной и той же задачи, и при выборе метода следует учитывать его сложность и характер влияния, которое он может оказать на производительность. Чтобы проиллюстрировать все многообразие доступных возможностей, пример трехступенчатого конвейера был реализован несколькими отличными друг от друга способами.

Тщательное продумывание проекта и путей его реализации является предпочтительным способом улучшения качества программы. Возложение чрезмерных надежд на результаты тестирования и отладку без того, чтобы уделить должное внимания деталям, может привести к возникновению серьезных проблем, обнаружить и устранить которые будет очень трудно.

В следующих главах

В главе 11 показано, как организовать взаимодействие между процессами и потоками, выполняющимися внутри этих процессов, используя именованные каналы (named pipes) и почтовые ящики (mailslots) Windows. В качестве основного примера выбрана клиент-серверная система, в которой для обслуживания запросов клиентов используется пул рабочих потоков. В главе 12 эта же система реализуется с привлечением гнезд Windows Sockets, что делает ее пригодной для использования стандартных протоколов. В обновленной клиент-серверной системе применяются безопасные библиотеки DLL с многопоточной поддержкой, а сервер использует внутрипроцессный сервер (in-process server) на основе DLL.

Дополнительная литература

Источником большей части информации и советов по программированию, приведенных в конце настоящей главы, послужила книга [6]. Из нее же было взято в переработанном виде и решение на базе объекта барьера, использованное в программах 10.1 и 10.2.

В статье Дугласа Шмидта (Douglas Schmidt) и Ирфана Пьярали (Irfan Pyarali) "Strategies for Implementing POSIX Condition Variables in Win32" ("Стратегии реализации переменных условий POSIX в Win32") (доступна по адресу http:// www.es.wustl.edu/~schmidt/win32-cv-1.html), обсуждается ограниченность событий Win32 (Windows) и эмуляция переменных условий, а также дан глубокий анализ и оценка нескольких подходов. Вместе с тем, этот материал был написан еще до появления функции SignalObjectAndWait, и поэтому большое внимание в статье уделяется тому, как избежать потери сигналов. Чтение этой статьи позволит вам по достоинству оценить возможности новых функций. В другой статье тех же авторов (http://www.cs.wustl.edu/~schmidt/win32-cv-2.html) рассматривается создание объектно-ориентированных оболочек вокруг объектов синхронизации Windows, позволяющих добиться независимости интерфейса синхронизации от платформы. Основанная на работе Шмидта и Пьярали реализация, в которой используются потоки Pthreads, доступна по адресу http://sources.redhat.com/pthreads-win32/.

Упражнения

10.1. Переработайте программу 10.1, исключив из нее функцию SignalObjectAndWait; для тестирования полученного результата воспользуйтесь Windows 9x.

10.2. Модифицируйте программу evenpc (программа 8.2), использовав модель переменных условий и обеспечив возможность существования нескольких потребителей. События какого типа потребуются в данном случае?

10.3. Измените логику работы программы 10.2 таким образом, чтобы объект события переходил в сигнальное состояние только один раз.

10.4. Замените мьютекс в объекте очереди, который используется в программе 10.2, объектом CS. Какое влияние это изменение оказывает на производительность и пропускную способность программы? Решение находится на Web-сайте книги, а соответствующие экспериментальные данные приведены в приложении В.

10.5. Для индикации состояний очереди, в которых она не пуста или не заполнена, в программе 10.4 применяется широковещательная модель CV. Будет ли в данном случае работать сигнальная модель CV? Не является ли она даже более предпочтительной в некоторых отношениях? Соответствующие экспериментальные данные приведены в приложении В.

10.6. Поэкспериментируйте с размерами очереди и величиной коэффициента блокирования "передатчик/приемник" в программе 10.5 для выяснения того, какое влияние оказывают эти факторы на загрузку ЦП, а также производительность и пропускную способность программы.

10.7. Видоизмените программы 10.3–10.5, обеспечив их соответствие принятым в Windows соглашениям о правилах образования имен, которых мы придерживаемся на протяжении всей книги.

10.8. *Для программистов на* C++. Приведенный в программах 10.3 и 10.4 код можно использовать для создания в C++ класса синхронизированной очереди; создайте этот класс и протестируйте его, модифицировав соответствующим образом программу 10.5. Какие из функций должны быть общедоступными, а какие — закрытыми?

10.9. Исследуйте, как изменятся показатели производительности программы 10.5 после замены мьютексов объектами CRITICAL_SECTIONS.

10.10. Улучшите программу 10.5, исключив необходимость прекращения выполнения потоков передатчика и приемника. Потоки должны самостоятельно завершать свое выполнение.

10.11. На web-сайте находится файл multisem.c, который реализует сложный семафор, имитирующий объекты Windows (они имеют имена и атрибуты безопасности, могут разделяться процессами, и для них предусмотрены две модели ожидания), а также файл тестовой программы TestMultiSem.c. Выполните сборку и тестирование этой программы. Как в ней используется модель переменных условий? Повышается ли производительность в результате использования объекта CRITICAL_SECTION? Что здесь выступает в роли инвариантов и предикатов переменных условий?

10.12. Проиллюстрируйте целесообразность рекомендаций, приведенных в конце настоящей главы, ссылаясь на ошибки, с которыми вам пришлось столкнуться, или ошибки, содержащиеся в версии программы с дефектами, представленной на Web-сайте.

10.13. Ознакомьтесь со статьей Шмидта и Пьярали "Strategies for Implementing POSIX Condition Variables in Win32" ("Стратегии реализации переменных условий POSIX в Win32") (см. раздел "Дополнительная

литература"). Примените их методы анализа равноправия, корректности, сериализации и других программных факторов к моделям переменных условий (которые в указанной статье называются "идиомами" ("idioms")), фигурирующим в настоящей главе. Заметьте, что сами переменные условия в настоящей главе не эмулируются; вместо этого эмулируется их использование, тогда как Шмидт и Пьярали эмулируют переменные условий, используемые в произвольном контексте.

10.14. Находящиеся на web-сайте проекты batons и batonsmultipleevents демонстрируют альтернативные варианты решения задачи сериализации выполнения потоков. О предпосылках и предшествующих работах других авторов говорится в комментариях, включенных в код. Во втором решении с каждым потоком связывается уникальное событие, что позволяет отслеживать сигнальные состояния отдельных потоков. Для реализации выбран язык C++, что дало возможность воспользоваться средствами стандартной библиотеки шаблонов C++ (Standard Template Library, STL). Проанализируйте, что имеют общего и чем различаются между собой эти два решения и используйте второе из них в качестве средства ознакомления с библиотекой STL.