Глава 6. Разработка параллельного сервера

6.1. Предисловие к главе

В предыдущих главах были рассмотрены основные механизмы и интерфейсы операционной системы Windows, позволяющие осуществлять обмен данными между распределенными в сети TCP/IP процессами. Приведенные примеры демонстрировали простейшие распределенные приложения (один сервер – один клиент), созданные на основе этих интерфейсов. Для разработки более сложных распределенных приложений в среде Windows, кроме этого требуются еще специальные методы и приемы программирования.

Основной целью этой главы является изучение методов и приобретение навыков разработки распределенных приложений с архитектурой клиент-сервер, но имеющих более сложную, чем один сервер – один клиент структуру. На рисунке 6.1.1 изображена структура распределенного приложения, на создание которого будет в основном ориентировано дальнейшее изложение материала.

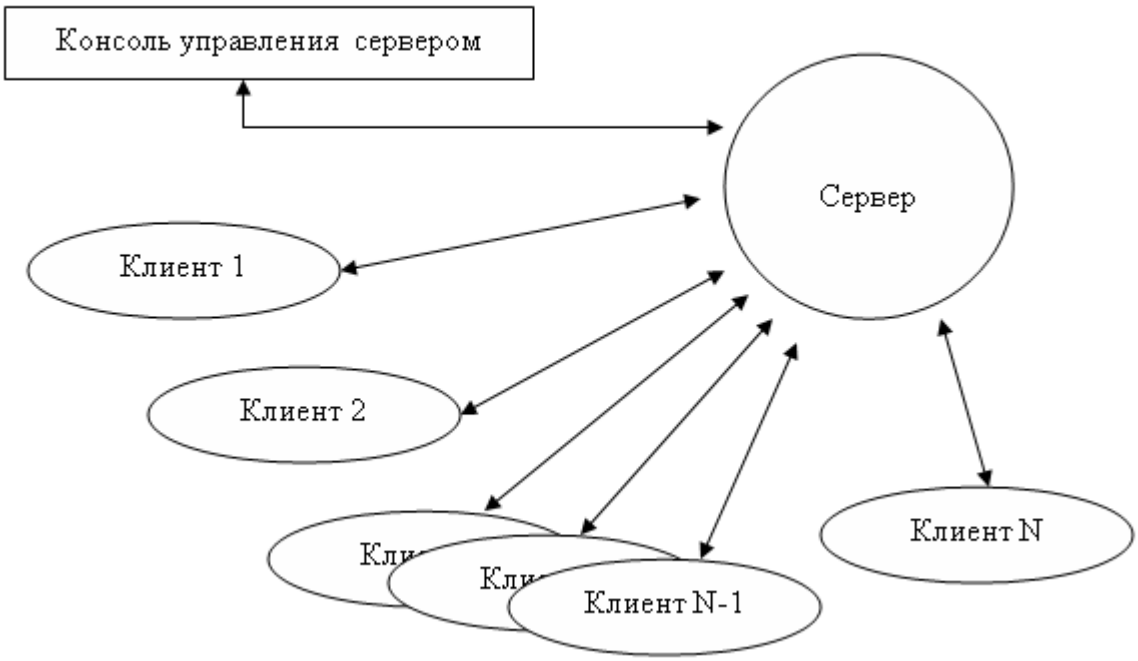

Рисунок 6.1.1. Структура распределенного приложения с сервером, обслуживающим одновременно несколько клиентов

Распределенное приложение, изображенное на рисунке 6.1.1 предполагает наличие одной программы сервера, которая одновременно обслуживает несколько программ клиентов. Управление сервером осуществляется с помощью специальной программы, которую будем называть консолью управления.

Серверы, одновременно обслуживающие несколько клиентов, по методу облуживания подразделяются на ***итеративные*** и ***параллельные*** серверы ***(iterative and concurrent servers)***.

Работа итеративного сервера описывается циклом из четырех шагов: 1) ожидание запроса от клиента; 2) обработка запроса; 3) отправка результата запроса; 4) возврат в ждущее состояние 1. Очевидно, что сервер этого класса может применяться в том случае, если предполагаются короткие запросы от клиентов, не требующие больших затрат на обработку и длинных ответов сервера. Как правило, подобные серверы работают над UDP, когда нет необходимости создавать отдельный канал связи для каждого клиента. Консоль управления в этом случае может быть выполнена в виде специального клиента, запросы которого и есть команды управления сервером.

Параллельные серверы имеют другой цикл работы: 1) ожидание запроса от клиента; 2) запуск нового сервера для обработки текущего запроса; 3) возврат в ждущее состояние 1. Преимущество параллельных серверов заключается в том, что он лишь порождает новые серверы, которые и занимаются обработкой запросов клиентов. Очевидно, что для создания параллельных серверов необходимым условием является мультизадачность операционной среды сервера. По всей видимости, параллельные серверы целесообразно использовать, если предполагается наличие относительно длительного сеанса связи между клиентом и сервером. Как правило, параллельные серверы работают над TCP. Консоль управления может быть создана, как отдельный процесс или поток (в зависимости от возможностей операционной системы) в рамках сервера или так же, как предлагалось для итеративного сервера, выполнить в виде специализированного клиента.

Дальнейшее изложение, в основном, будет посвящено разработке параллельных серверов. При этом мы будем говорить, что в рамках одного сервера работает несколько процессов, работающих параллельно (в одно и тоже время) и выполняющих специфические функции в рамках сервера.

Следует обратить внимание, что определенную путаницу может внести применяемая терминология. Дело в том, что любой процесс сервера может быть реализован, как ***поток*** или как ***процесс*** операционной системы (подразумеваются операционные системы семейства Windows 200x/XP). Поэтому, если в тексте используется понятие “***процесс операционной системы”*** – это будет специально оговариваться. В другом случае, под понятием “***процесс сервера”***, подразумевается часть программы сервера работающая параллельно с другими частями (процессами сервера) независимо о способа реализации.

6. 2. Особенности разработки параллельного сервера

Разработчик параллельного сервера сталкивается с рядом проблем, которые обусловлены необходимостью одновременно обслуживать несколько клиентов. Для этого требуется в рамках программы сервера организовать совместную работу нескольких процессов (еще раз напомним о

терминологическом соглашении), каждый из которых предназначен для обслуживания подключившегося клиента или для выполнения каких-то внутренних задач сервера.

Критическим по времени для параллельного сервера является момент подключения клиента. Сервер не должен тратить много времени на подключение клиента, т.к. в этот момент могут осуществлять подключение другие клиенты, которые из-за занятости сервера могут получить отказ. Поэтому целесообразно в сервере выделить отдельный процесс (и желательно, чтобы он имел наивысший приоритет), который бы был занят только подключением клиентов к серверу.

С другой стороны, может потребоваться управлять процессом подключения. Например, если оператор консоли управления ввел команду, запрещающую подключение новых клиентов к серверу. В этом случае необходимо как-то вмешаться в этот процесс и запретить подключения новых клиентов до получения команды вновь разрешающей подключение к серверу клиентов.

Так как процесс подключения клиента должен выполняться быстро, то загружать его другой работой не целесообразно. Но в работе параллельного сервера необходимо выполнять еще ряд действий, не связанных с подключением клиентов. Например, управление сервером с консоли подразумевает цикл ожидания, ввода и обработки команд оператора. По всей видимости, подобные действия следует выполнять в отдельных процессах.

После подключения клиента к параллельному серверу запускается отдельный процесс, обслуживающий запрос клиента. Для управления обслуживающими процессами и для сбора статистики требуется динамическая структура данных (позволяющая добавлять и удалять элементы структуры), предназначенная для хранения информации о работающих в настоящий момент обслуживающих процессах. Как правило, для хранения подобной информации используют связный список.

Отдельные процессы, работающие в рамках параллельного сервера, могут использовать общие ресурсы. Некоторые ресурсы, могут быть разрушены при совместном использовании несколькими параллельными процессами (например, связный список). Для разрешения этой проблемы следует использовать механизмы синхронизации, позволяющие последовательно использовать такие критические ресурсы.

6.3. Структура параллельного сервера

Структура параллельного сервера, зависит, в конечном счете, от характера решаемой сервером задачи, но все же существуют общие структурные свойства сервера, на которых следует остановиться. Для того, чтобы упростить изложение будем рассматривать далее конкретную реализацию (модель) параллельного сервера с именем ConcurrentServer (рисунок 6.3.1) структура которого, по мнению автора, отражает все требующие внимания моменты.

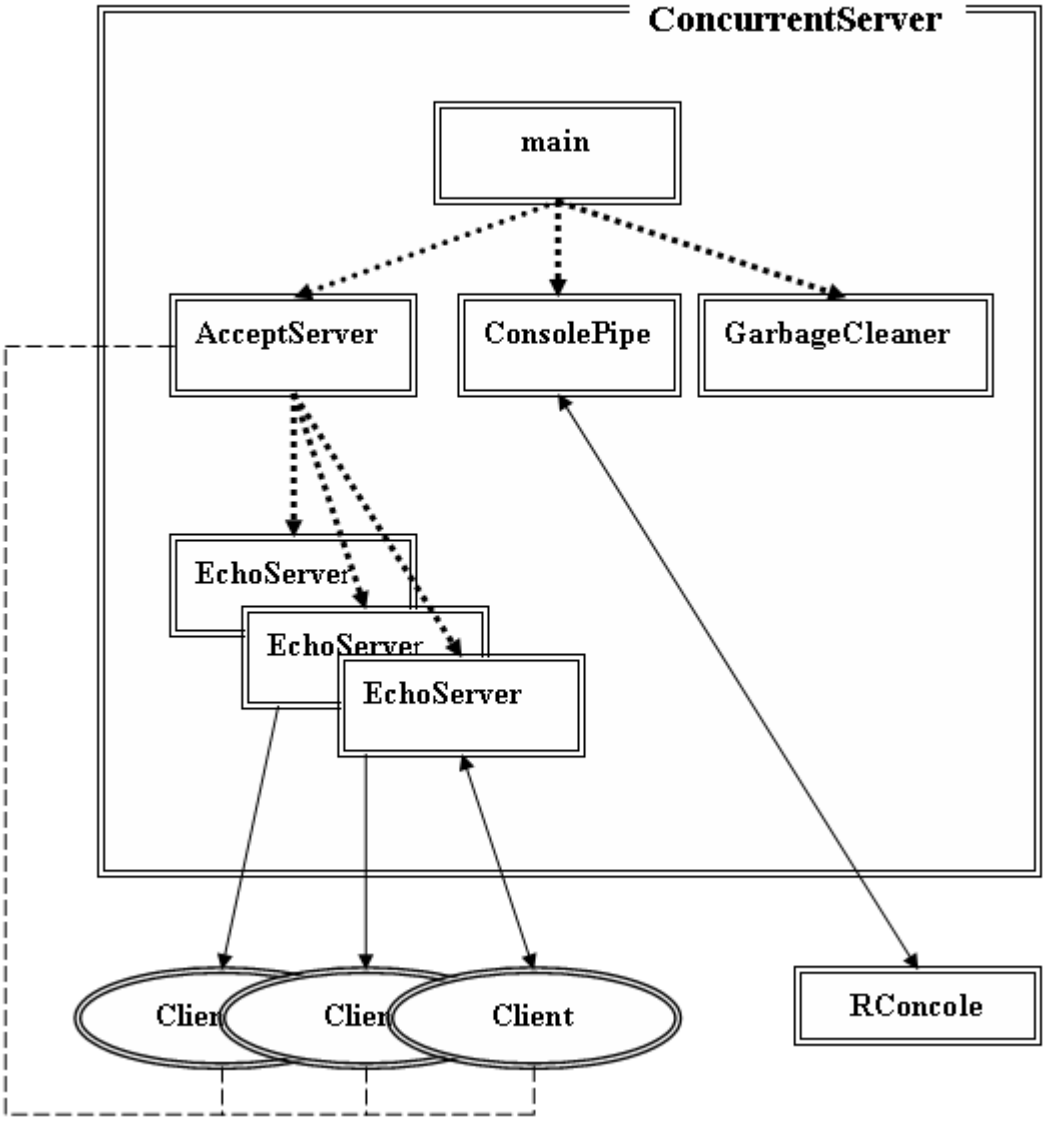

Рисунок 6.3.1. Структура параллельного сервера

На рисунке 6.3.1 изображена структура параллельного сервера, назначением которого является, одновременное обслуживание нескольких клиентских программ. Обслуживание заключается в получении от клиента по установленному TCP-соединению последовательности символов и в возврате (пересылке) этой последовательности обратно. Кроме того, предполагается, что сервер может выполнять команды, введенные с консоли управления, с которой поддерживается связь через именованный канал (Named Pipe).

Все процессы, работающие в рамках сервера, изображены на рисунке прямоугольниками. Пунктирными направленными линиями обозначается создание (запуск) одного процесса другим. Процесс с именем main, является главным процессом сервера, который получает управление от операционной системы. Этот процесс создает и запускает новые процессы AcceptServer, ConsolePipe и GarbageCleaner. Процесс AcceptServer, в свою очередь создает

несколько процессов с именем EchoServer. Сплошными двунаправленными линиями обозначается перемещение данных. Данные перемещаются между EchoServer-процессами сервера и клиентскими программами (с именем Client), обозначенными на рисунке овалами, а также между программой с именем RConsole, реализующей клиентскую часть консоли управления, и процессом ConsolePipe, который реализует серверную часть консоли управления. Штриховой линией, соединяющей изображение клиентских программ и изображение процесса AcceptServer, обозначается процедура создания соединения между клиентом и сервером.

Опишем назначение компонентов изображенного на рисунке 6.3.1 распределенного приложения.

Процесс main. Основным назначением процесса main, является запуск, инициализация и завершение работы сервера. Как уже отмечалось, именно этот процесс первым получает управление от операционной системы. Процесс main запускает основные процессы: AcceptServer, ConsolePipe и RConsole.

Процесс AcceptServer. AcceptServer создается процессом main и предназначен для выполнения процедуры подключения клиентов к серверу, для исполнения команд консоли управления, а также для запуска процессов EchoServer, обслуживающих запросы клиентских программ по созданным соединениям. Кроме того, AcceptServer создает список подключений, который далее будем называть ***ListContanct***. При подключении очередного клиента, процесс AcceptServer добавляет в ListContact элемент, предназначенный для хранения информации о состоянии данного подключения.

Процесс ConsolePipe. ConsolePipe создается процессом main и является сервером именованного канала, по которому осуществляется связь между программой RConsole (консоль управления сервером) и параллельным сервером.

Процесс GarbageCleaner. Основным назначением процесса GarbageCleaner является удаление элемента списка подключений ListContact, после отключения программы клиента. Следует сразу отметить, что ListContact является ресурсом, требующим последовательного использование. Одновременная запись и (или) удаление элементов списка может привести к разрушению списка ListContact.

Процесс EchoServer. Процессы EchoServer создаются процессом AcceptServer по одному для каждого успешного подключения программы клиента. Основным назначением процесса EchoServer является прием данных по созданному процессом AcceptServer подключению и отправка этих же данных без изменения обратно программе клиента. Условием окончания работы сервера является получение от клиента пустого сегмента данных (имеющего нулевую длину).

Программа Client. Программа Client предназначена для пересылки данных серверу и получения ответа от сервера. Программа может работать, как на одном компьютере с сервером (будет использоваться интерфейс

внутренней петли), так и на другом компьютере, соединенным с компьютером сервера сетью TCP/IP. Для окончания работы с сервером программа формирует и отправляет сегмент данных нулевой длины.

Программа RConsole. Программа RConsole предназначена для ввода команд управления сервером и для вывода диагностических сообщений полученных от сервера . RConsole является клиентом именованного канала.

Список подключений ListContact. Список ListContact (не изображен на рисунке) создается основе стандартного класса list и предназначен для хранения информации о каждом подключении. Список создается пустым при инициализации процесса AcceptServer. В рамках этого же процесса осуществляется добавление элементов списка, по одному для каждого подключения. При отключении программы клиента от сервера, соответствующий элемент списка помечается, как неиспользуемый. Удаление неиспользуемого элемента осуществляется процессом GarbageCleaner, который работает в фоновом режиме.

Описанная выше модель распределенного приложения, по мнению автора, является достаточно полной для того, чтобы изложить основные принципы создания параллельного сервера. Дальнейшее изложение материала будет опираться на эту модель.

6.4. Потоки и процессы в Windows

В описанной выше модели параллельного сервера с именем ConcurrentServer, предполагается запуск процессов, работающих параллельно. Сначала процесс main запускает три параллельно работающих процесса (AcceptServer, ConsolePipe и GarbageCleaner), а потом еще процесс AcceptServer осуществляет запуск нескольких процессов (по одному для каждого подключения) EchoServer.

Для организации параллельной работы программ в операционной системе Windows предусмотрены два специальных механизма: ***механизм потоков*** и ***механизм процессов .***

Понятие потока тесно связано с последовательностью действий процессора во время выполнения программы. Исполняя программу, процессор последовательно выполняет инструкции (машинные коды) программы, иногда осуществляя переходы. Такая последовательность выполнения инструкций называется ***потоком управления (thread – нить).*** Будем говорить, что программа является ***многопоточной***, если в ней существуют одновременно несколько потоков управления. Сами потоки в этом случае называются ***параллельными***. Если в программе может существовать только один поток, то такая программа называется ***однопоточной***.

В рамках потока управления многопоточной программы могут вызываться функции. При вызове одной и той же функции в разных потоках управления, важно чтобы эта функция обладала свойством ***безопасности для потоков***. Безопасная для потоков функции в обладает двумя свойствами: 1) свойством ***реентерабельности***; 2) функция обеспечивает

блокировку доступа к критическим ресурсам, которые она использует.

В общем случае функция называется реентерабельной, если она не изменяет собственный код или собственные статические данные. Другими словами программный код реентерабельной функции должен допускать корректное его использование несколькими потоками одновременно.

Блокировка требуется в том случае, если функцией используется ресурс, доступ к которому может быть только упорядоченным (критический ресурс). Примером критического ресурса может служить изменяемые функцией статические и глобальные переменные.

Каждое приложение, работающее в среде Windows, имеет, по крайней мере, один поток, который называется ***первичным*** или ***главным*** потоком. В консольных приложениях этот поток исполняет функцию ***main***. В приложениях с графическим интерфейсом это поток, который исполняет функцию ***WinMain***.

Поток управления в Windows является объектом ядра операционной системы, которому выделяется процессорное время для выполнения приложения. Каждому потоку принадлежат следующие ресурсы:

- код исполняемой функции;
- набор регистров процессора;
- область оперативной памяти;
- стек для работы приложения;
- стек для работы операционной системы;
- маркер доступа, содержащий информацию для системы безопасности.

Все эти ресурсы образуют так называемый ***контекст потока*** в Windows.
Основные функции для работы с потоками перечислены в таблице 6.4.1.

Таблица 6.4.1

Наименование функции	**Назначение**
`CreateThread`	Создать поток
`ResumeThread`	Возобновить поток
`SuspendThread`	Приостановить поток
`Sleep`	Задержать исполнение
`TermitateThread`	Завершить поток

Для создания потоков в Windows используется функция CreateThread, описание которой приводится на рисунке 6.4.1. Третий параметр функции (pFA) указывает на функцию, которая первая получит управление в созданном потоке. Функция потока принимает только один параметр, значение которого передается с помощью четвертого параметра (pPrm). Функция потока должна завершаться вызовом функции ExitThread, с параметром, устанавливающим код возврата. Пример правильного оформления функции потока и именем AcceptServer приведен на рисунке 6.4.2.

```
// -- создать  поток
// Назначение: функция  предназначена для создания  потока

 HANDLE CreateThread(
      PSECURITY_ATTRIBUTES   pSA,   // [in] атрибуты защиты
      DWORD                  sSt,   // [in] размер стека потока
      LPTHREAD_START_ROUTINE pFA,   // [in] функция потока
      LPVOID                 pPrm,  // [in] указатель на  параметр
      DWORD                  flags, // [in] индикатор запуска
      LPDWORD                pId    // [out] идентификатор потока

                  );
// Код возврата: в случае успешного завершения функция
//       возвращает дескриптор потока; иначе возвращается
//       значение NULL
// Примечания: - значение параметра pSA может принимать
//      значение NULL, в этом случае параметры защиты потока
//      будут установлены по умолчанию;
//       - значение параметра sSt может принимать значение
//      NULL, в этом случае размер стека потока будет
//      установлен по умолчанию (1МБ);
//       - если значение параметра flags установлено NULL, то
//      функция потока начнет выполняться сразу после создания
//      потока;  если же установлено значение CREATE_SUSPENDED,
//      то поток остается в состоянии готовности к исполнению
//      функции до вызова функции ResumeThread;
//      - возвращаемое значение идентификатора потока может
//      быть использовано при вызове  некоторых функций
//      управления потоком; допускается установка  NULL для
//      параметра  pId -  в этом случае идентификатор не
//      возвращается
```

Рисунок 6.4.1. Функция CreateThread

```
DWORD WINAPI AcceptServer (LPVOID pPrm)     // прототип
{
 DWORD rc = 0;     // код возврата

 //.........................................
 ExitThread(rc);  // завершение работы потока
}
```

Рисунок 6.4.2. Структура функции потока

На рисунке 6.4.3 приводится пример использования функции CreateThread. В этом примере главный поток main создает три потока с потоковыми функциями AcceptServer, ConsolePipe, GarbageCleaner. Следует обратить внимание, что после запуска потоков, перед завершением функции (и потока) main, три раза вызывается функция WaitForSingleObject у которой в качестве первого параметра используется дескриптор потока, а

второй параметр имеет значение, определенное константой INFINITE. Такой вызов функции WaitForSingleObject приостанавливает выполнение основного потока main до завершения работы потока, соответствующего указанному в параметре дескриптору. Отсутствие этих функций могло бы привести к завершению потока main, до завершения порожденных им потоков. В этом случае порожденные потоки тоже автоматически завершаются операционной системой. После завершения работы потока следует освободить связанные с потоком ресурсы с помощью функции CloseHandle.

```
 #include <windows.h>      // для функций управления потоками
//.............................................................
HANDLE hAcceptServer,     // дескриптор потока AcceptServer
       hConsolePipe,      // дескриптор потока ConsolePipe
       hGarbageCleaner    // дескриптор потока GarbageCleaner
DWORD WINAPI AcceptServer(LPVOID pPrm);   // прототипы функций
DWORD WINAPI ConsolePipe(LPVOID pPrm);
DWORD WINAPI GarbageCleaner(LPVOID pPrm);

int _tmain(int argc, _TCHAR* argv[])         //главный поток
{
 volatile TalkersCommand  cmd = START;       // команды сервера
 hAcceptServer   = CreateThread(NULL,NULL, AcceptServer,
                                (LPVOID)&cmd, NULL, NULL),
 hConsolePipe    = CreateThread(NULL,NULL, ConsolePipe,
                                (LPVOID)&cmd, NULL, NULL),
 hGarbageCleaner = CreateThread(NULL,NULL,GarbageCleaner,
                                 (LPVOID)NULL, NULL, NULL);

WaitForSingleObject(hAcceptServer,INFINITE);
CloseHandle(hAcceptServer);
WaitForSingleObject(hConsolePipe,INFINITE);
CloseHandle(hConsolePipe);
WaitForSingleObject(hGarbageCleaner,INFINITE);
CloseHandle(hConsolePipe);
return 0;
};
```

Рисунок 6.4.3. Пример использования функции CreateThread

Другой важный момент в приведенном примере, на который следует обратить внимание, это применение квалификатора volatile. Оператор volatile указывает компилятору на необходимость размещения переменной cmd в памяти и не осуществлять относительно этого размещения никакой оптимизации. Дело в том, что область памяти отведенная переменной cmd, используется двумя параллельно работающими потоками AcceptServer и ConsolePipe. Поэтому оптимизация может привести к тому, что потоки будут использовать различные области памяти.

Один поток может завершить другой поток с помощью функции TerminateThread (рисунок 6.4.4). Использовать эту функцию следует только в аварийных ситуациях, т.к. завершение потока подобным образом не освобождает распределенные операционной системой ресурсы.

```
// -- завершить  поток
// Назначение: функция  предназначена для завершения  потока
//             без освобождения ресурсов

 BOOL TerminateThread(
      HANDLE   hT,   // [in] дескриптор потока
      DWORD    rc    // [in] код завершения потока
                    )
// Код возврата: в случае успешного завершения функция
//       возвращает ненулевое значение; иначе возвращается
//       значение NULL
```

Рисунок 6.4.4. Функция TerminateThread

Исполнение каждого потока может быть приостановлено с помощью функции SuspendThread (рисунок 6.4.5). С каждым созданным потоком связан специальный счетчик, показывающий сколько раз была выполнена приостановка потока функцией SuspendThread. Максимальное значение счетчика приостановок равно MAXIMUM_SUSPEND_COUNT. Поток исполняется только в том случае если значение счетчика приостановок рано нулю.

```
// -- приостановить  поток
// Назначение: функция  предназначена для временной
//             приостановке исполнения  потока

 DWORD SuspendThread(
      HANDLE   hT,   // [in] дескриптор потока
                    )
// Код возврата: в случае успешного завершения функция
//       возвращает текущее значение счетчика приостановок;
//       иначе возвращается значение -1
```

Рисунок 6.4.5. Функция SuspendThread

Для уменьшения текущего значения счетчика приостановок потока используется функция ResumeThread (рисунок 6.5.6).

```
// -- возобновить поток
// Назначение: функция  предназначена для уменьшения счетчика
//             приостановки потока на единицу

 DWORD ResumeThread(
      HANDLE   hT,   // [in] дескриптор потока
                   )
// Код возврата: в случае успешного завершения функция
//       возвращает текущее значение счетчика приостановок;
//       иначе возвращается значение -1
// Примечание: если значение счетчика приостановок после
//       выполнения функции станет равным нулю, то поток
//       возобновляет свою работу с того места, где он был
//       приостановлен функцией SuspendThread
```

Рисунок 6.4.6. Функция ResumeThread

Полезной, особенно на этапе отладки, является функция Sleep (рисунок 6.4.7), с помощью которой поток может задержать свое исполнение на заданный интервал времени.

```
// -- задержать поток
// Назначение: функция  предназначена для задержки исполнения
//             потока на заданный интервал времени

 VOID  Sleep(
      DWORD   ms    // [in]  интервал времени в миллисекундах
            );
```

Рисунок 6.4.7. Функция Sleep

Под ***процессом*** операционной системы Window понимается объект ядра, которому принадлежат системные ресурсы, используемые исполняемым приложением операционной системы. Выполнение процесса начинается с первичного потока main. Во время выполнения, процесс может создавать новые потоки и порождать новые процессы. С точки зрения техники программирования, работа с процессами очень напоминает работу с потоками с одним существенным отличием. Отличие заключается в том, что потоки, работающие в рамках одного процесса, разделяют общее пространство памяти, а каждый процесс имеет свое собственное пространство. Поэтому для взаимодействия между различными процессами операционной системы (обмен данными, синхронизация и т.п.) надо использовать средства из разряда IPC, о которых уже говорилось раньше. Создание и запуск отдельного процесса по затратам ресурсов значительно превосходит затраты на создание потока в рамках существующего процесса. Кроме того, более затратными оказывается процедура переключения с одного процесса на другой. Все эти особенности приводят к значительной

потере производительности параллельного сервера, использующего механизм процессов операционной системы, по сравнению с сервером, сделанного с применением механизма потоков.

При разработке параллельных серверов использовать механизмы управления процессами целесообразно с тех случаях, когда в рамках сервера используются такие компоненты сервера (чаще всего это обслуживающие процессы), которые могут разрушить работу всего параллельного сервера. В этом случае, действительно, целесообразно реализовать эти компоненты сервера как отдельные процессы операционной системы Windows.

Дальнейшее изложение ориентировано на разработку сервера с применением механизма потоков операционной системы Windows. С использованием механизма процессов Windows можно ознакомиться в [4, 14].

6.5. Синхронизация потоков параллельного сервера

С некоторыми механизмами синхронизации используемыми в Windows мы уже познакомились, когда говорили о функции WaitSingleObject, которая использовалась для ожидания окончания работы потока. Еще один рассмотренный способ синхронизации – приостановка потока с помощью функции SuspendThread.

Критическим ресурсом, который можно использовать только последовательно, в нашей модели параллельного сервера является список подключений ListContact. На рисунке 6.5.1 предлагается пример реализации такого списка с помощью стандартного класса list [13].

Рассмотрим структуру элемента списка подключений (структура Contact). Два поля type и sthread предназначены для описания состояния соединения с клиентом на разных этапах: type – на этапе подключения; sthread – на этапе обслуживания. Поля s, prms, lprms используются для хранения параметров соединения. Для хранения дескриптора обсуживающего потока (в нашей модели потока EchoServer) используется поле hthread. Дескриптор htimer, может быть использован, для организации ожидающего таймера, позволяющего ограничить время работы обслуживающего процесса. Поля msg, srvname могут использоваться обслуживающими потоками для записи диагностирующего сообщения и символических имен обрабатывающих потоков.

Синхронизация будет осуществляться между двумя потоками: AcceptServer и GarbageCleaner. Как уже описывалось выше, синхронизация необходима в связи с тем, что одновременное добавление элемента в список ListContact, выполняемое потоком AcceptServer, и удаление элемента списка, выполняемое потоком GarbageCleaner может привести к непредсказуемым последствиям.

```
#include <list>
#include "Winsock2.h"
//.....................................................
using namespace std;

struct Contact          // элемент списка подключений
{
 enum TE{               // состояние  сервера подключения
    EMPTY,              // пустой элемент списка подключений
    ACCEPT,             // подключен (accept), но не обслуживается
    CONTACT             // передан обслуживающему серверу
         }     type;    // тип элемента списка подключений
 enum ST{               // состояние обслуживающего сервера
    WORK,               // идет обмен данными с клиентом
    ABORT,              // обслуживающий сервер завершился не нормально
    TIMEOUT,            // обслуживающий сервер завершился по времени
    FINISH              // обслуживающий сервер завершился  нормально
       }       sthread; // состояние  обслуживающего сервера (потока)

 SOCKET       s;         // сокет для обмена данными с клиентом
 SOCKADDR_IN prms;       // параметры  сокета
 int          lprms;     // длина prms
 HANDLE       hthread;   // handle потока (или процесса)
 HANDLE       htimer;    // handle таймера

 char msg[50];           // сообщение
 char srvname[15];       //  наименование обслуживающего сервера

 Contact( TE t = EMPTY, const char* namesrv = "") // конструктор
     {memset(&prms,0,sizeof(SOCKADDR_IN));
      lprms = sizeof(SOCKADDR_IN);
      type = t;
      strcpy(srvname,namesrv);
      msg[0] = 0;};

 void SetST(ST sth, const char* m = "" )
      {sthread = sth;
       strcpy(msg,m);}
   };
typedef list<Contact> ListContact;          // список подключений
```

Рисунок 6.5.1. Пример реализации списка подключений ListContact

Операционная система Windows обладает широким спектром механизмов синхронизации потоков и процессов: критические секции, мьютексы, события, семафоры, ожидающий таймер и т.д. Теоретические основы механизмов синхронизации потоков и процессов подробно изложены в [4, 15]. Здесь будет рассматриваться только механизм критических секций. С остальными способами синхронизации процессов и потоков операционной системы Windows можно ознакомиться в [4, 14].

Таблица 6.5.1

Наименование функции	Назначение
`DeleteCriticalSection`	Разрушить критическую секцию
`EnterCriticalSection`	Войти в критическую секцию
`InitializeCriticalSection`	Инициализировать критическую секцию
`LeaveCriticalSection`	Покинуть критическую секцию
`TryEnterCriticalSection`	Пытаться войти в критическую секцию

Критические секции, является одним из самых простых механизмов синхронизации и в нашей модели могут быть использованы для исключения совместного использования списка ListContact потоками AcceptServer и GarbageCleaner. Критическая секция является объектом операционной системы типа CRITICAL_SECTION. Для работы с этим объектом используются функции, которые перечислены в таблице 6.5.1.

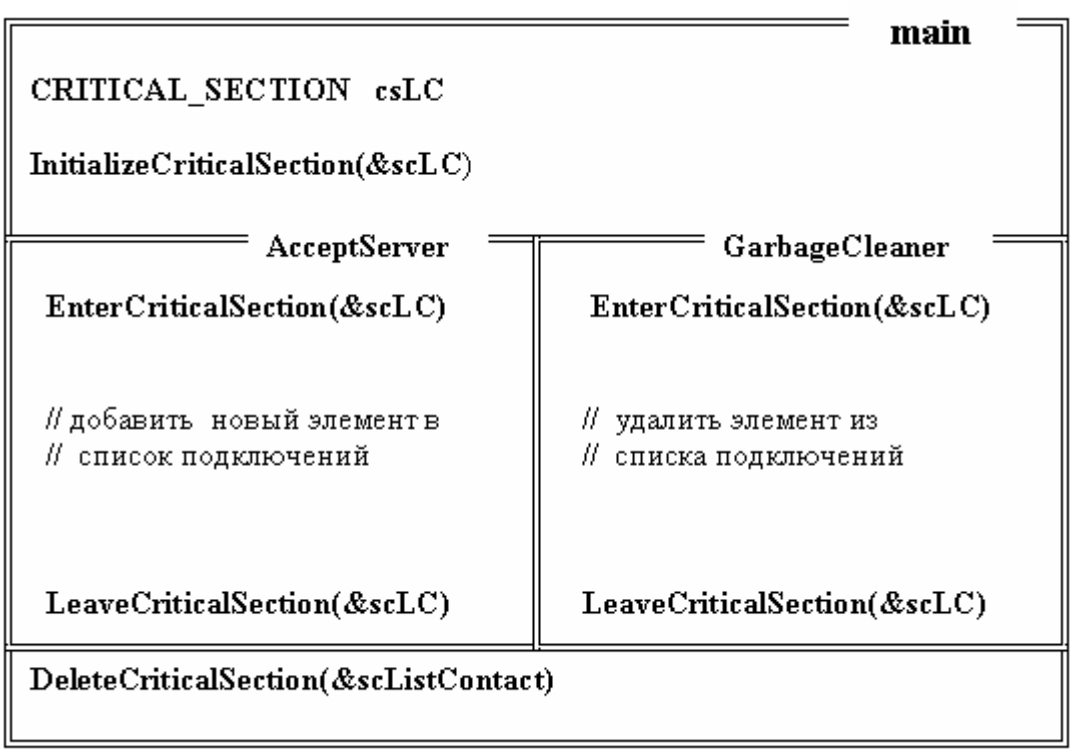

Рисунок 6.5.2. Схема применения механизма критических секций

Схема использования механизма критических секций изображена на рисунке 6.5.2.

```
// -- инициализировать критическую  секцию

// Назначение: функция  предназначена для  инициализации
//             критической секции
 VOID InitializeCriticalSection (
       LPCRITICAL_SECTION pCS, // указатель на критическую секцию
                                )

// -- войти в критическую  секцию
// Назначение: функция  предназначена для входа в критическую
//             секцию
 VOID EnterCriticalSection (
       LPCRITICAL_SECTION pCS, // указатель на критическую секцию
                               )
// Примечание: в том случае если в критической секции не
//       находится ни один из потоков, функция завершает
//       свою работу, пропуская поток внутрь критической
//       секции (занимая секцию); если же секция занята
//       другим потоком, то функция приостанавливает
//       выполнение потока до того момента освобождения секции

//-- покинуть критическую  секцию
// Назначение: функция  предназначена для выхода из
//             критической  секции

VOID LeaveCriticalSection (
       LPCRITICAL_SECTION pCS, // указатель на критическую секцию
                              )

//-- пытаться войти в критическую  секцию
// Назначение: функция  предназначена для условного входа в
//             критическую  секцию и

 BOOL TryEnterCriticalSection (
       LPCRITICAL_SECTION pCS, // указатель на критическую секцию
                              )
// Код возврата: функция возвращает  ненулевое значение в
//         в том случае если секция не занята или поток
//         уже находится в критической секции; иначе
//         возвращается значение NULL

//-- разрушить критическую секцию
// Назначение: функция  предназначена для разрушения
//             критической секции
   VOID DeleteCriticalSection (
       LPCRITICAL_SECTION pCS, // указатель на критическую секцию
                               )
```

Рисунок 6.5.3. Функции реализующие механизм критических секций

На рисунке изображены 6.5.2 два параллельно работающих потока AcceptServer и GarbageCleaner. При подключении очередного клиента в рамках потока AcceptServer в список подключений добавляется элемент, содержащий информацию об этом подключении. Поток GarbageCleaner просматривает последовательно элементы списка подключений и удаляет неиспользуемые элементы. Для того, чтобы добавление и удаление элементов списка не осуществлялось одновременно, эти операции помещают внутри критической секции соответствующего потока. Каждая критическая секция начинается функцией EnterCriticalSection, а заканчивается функцией LeaveCriticalSection. Следует обратить внимание, что эти функции используют в качестве параметра общий объект синхронизации.

6.6. Асинхронный вызов процедур

Может возникнуть ситуация, когда одному из потоков параллельного сервера потребуется выполнить в рамках другого или нескольких других потоков процедуру. Причем старт процедуры должен быть согласованным с исполняющим потоком. Для решения такой задачи может быть применен механизм асинхронного вызова процедур.

Асинхронной процедурой называется функция, которая выполняется асинхронно в контексте какого-нибудь потока. Для исполнения асинхронной процедуры необходимо определить асинхронную процедуру, указать поток, в контексте которого она будет выполняться, и дать разрешение на выполнение асинхронной процедуры.

Если перейти к описанной выше модели сервера, то можно предложить следующий пример использования асинхронных процедур. При подключении очередного клиента, функция AcceptServer запускает обслуживающий поток EchoServer. С этого момента AcceptServer теряет связь с потоком EchoServer и, в принципе, даже не знает о моменте окончания его работы. Будем предполагать, что в начале и при окончании работы потока EchoServer, поток AcceptServer должен выдавать на свою консоль сообщение о завершении работы обслуженного клиента. В этом случае поток EchoServer может поставить функцию (асинхронную процедуру) в специальную очередь к потоку AcceptServer. Момент выполнения асинхронной процедуры определяется внутри функции потока AcceptServer.

Основные функции необходимые для асинхронного вызова процедур перечислены в таблице 6.6.1.

Таблица 6.6.1

Наименование функции	Назначение
`QueueUserAPC`	Поставить асинхронную процедуру в очередь
`SleepEx`	Приостановит поток для выполнения асинхронных процедур

На рисунке 6.6.1 изображена схема использования механизма асинхронного вызова процедур для параллельного сервера ConcurrentServer.

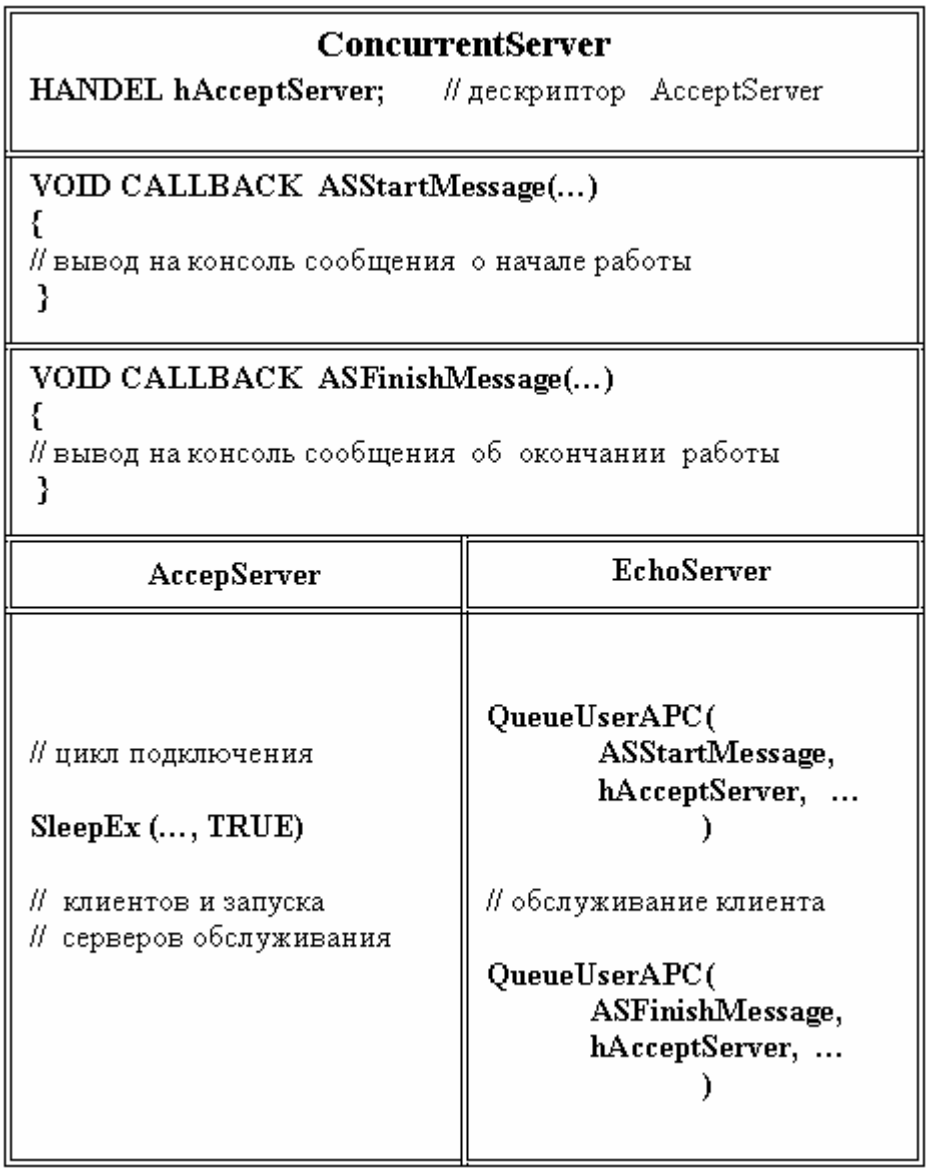

Рисунок 6.6.1. Схема использования механизма асинхронного вызова процедур

На схеме изображен параллельный сервер ConcurrentServer, в рамках которого определены две асинхронные процедуры ASStartMessage и ASFinishMessage, а также два параллельно работающих потока AcceptServer и EchoSever.

В начале своей работы функция EchoServer выполняет функцию QueueUserAPC (рисунок 6.6.2), которая помещает асинхронную процедуру ASStartMessage в очередь к потоку AcceptServer. Эта очередь обеспечивается операционной системой Windows и работает по алгоритму FIFO (First Input First Output). После отключения программы клиента, перед самым завершением своей работы, функция EchoServer вновь исполняет функцию

QueueUserAPC, но уже для постановки в очередь асинхронной процедуры ASFinishMessage.

Выполнение всех асинхронных процедур, находящихся к этому моменту в очереди потока, осуществляется последовательно в потоке AcceptServer, после того, как в рамках этого потока будет выполнена функция SleepEx (рисунок 6.6.3).

```
// -- поставить асинхронную процедуру в очередь
// Назначение: функция  предназначена для постановки  в
//             FIFO-очередь к потоку асинхронную  процедуру

 DWORD QueueUserAPC(
          PAPCFUNC  fn,   // [in] имя функции асинхронной процедуры
          HANDLE    hT,   // [in] дескриптор исполняющего потока
          DWORD     pm    // [in] передаваемый параметр
                   )
// Код возврата: в случае успешного завершения функция
//       возвращает ненулевое значение; иначе возвращается
//       значение NULL
// Примечание:  в случае неудачного выполнения устанавливается
//       системный код возврата, который может быть получен
//       с помощью функции GetLastError
```

Рисунок 6.6.2. Функция QueueUserAPC

```
// -- приостановить поток для выполнения асинхронных процедур
// Назначение: функция позволяет приостановить поток для
//             ожидания и  последовательного выполнения в
//             контексте данного потока асинхронных процедур,
//             находящихся к этому моменту в очереди

 DWORD SleepEx(
             DWORD  ms,   // [in] интервал времени в миллисекундах
             BOOL   rg    // [in] режим
              )
// Код возврата: в случае истечения заданного интервала
//       времени  функция возвращает значение NULL, иначе
//       возвращает ненулевое значение
// Примечание:- если  для параметра rg установлено значение
//        TRUE, то при наличии асинхронных процедур в
//        очереди потока они начинают немедленно выполняться;
//        если же асинхронных процедур в очереди нет, то
//        ожидается их появление заданный в параметре ms
//        интервал времени;
//        если для параметра rg установлено значение FALSE,
//        то поток просто приостанавливается на заданный
//        параметром ms интервал времени
```

Рисунок 6.6.3. Функция SleepEx

Асинхронные процедуры не могут возвращать никакого значения и принимают только один параметр. Прототип асинхронной процедуры изображен на схеме использования механизма асинхронных процедур (рисунок 6.6.1).

6.7. Использование ожидающего таймера

Очевидно, что производительность параллельного сервера в значительной степени зависит от количества одновременно подключившихся клиентов: с ростом подключившихся клиентов, производительность убывает. Обслуживание каждого клиента связано с выделением ему определенных ресурсов: процессорного времени, оперативной памяти, сетевого трафика и т.п. В связи с ограниченностью ресурсов возникает необходимость управлять процессом обслуживания.

Одной из задач управления сервером является выявление слишком продолжительных подключений. Подключения, которые удерживаются клиентом сверх разумного времени, могут возникнуть по самым разным причинам: особенности алгоритма, зацикливание или зависание программы клиента и т.п. Очевидным решением проблемы является введение ограничения на продолжительность соединения. Для реализации такого решения может быть использован механизм ожидающего таймера.

Опишем еще один очевидный случай, когда может быть востребован ожидающий таймер. Речь идет о поддержке некоторого расписания работ в рамках сервера. Например, предполагается, что некоторые услуги сервера поддерживаются только в определенные интервалы времени суток, или существует поток (или процесс) периодически запускаемый по определенному расписанию, для выполнения внутренних функций самого сервера (например, для вывода собранной статистики в предназначенный для этого файл).

Ожидающим таймером в Windows, называется объект синхронизации, который переходит в ***сигнальное состояние*** при наступлении заданного момента времени. Если ожидающий таймер ждет момента перехода в сигнальное состояние, то говорят, что он находится в ***активном состоянии***. Другое состояние ожидающего таймера ***пассивное*** – из этого состояния он не может перейти в сигнальное состояние.

По способу перехода из сигнального стояния в несигнальное, ожидающие таймеры разделяются на ***таймеры с ручным сбросом*** и таймеры с ***автоматическим сбросом***, иначе называемые ***таймерами синхронизации***.

По способу перехода из несигнального состояния в сигнальное, ожидающие таймеры бывают ***периодические*** и ***непериодические***. Периодические таймеры работают по циклу: активное состояние – сигнальное состояние – активное состояние. Непериодические таймеры могут только один раз перейти из активного состояния в сигнальное.

Основные функции, необходимые для работы с ожидающим таймером перечислены в таблице 6.7.1.

Таблица 6.7.1

Наименование функции	Назначение
CancelWaitableTimer	Отменить ожидающий таймер
CreateWaitableTimer	Создать ожидающий таймер
OpenWaitableTimer	Открыть существующий ожидающий таймер
SetWaitableTimer	Установить ожидающий таймер
WaitForSingleObject	Ждать сигнального состояния ожидающего таймера

```
// -- создать ожидающий таймер
// Назначение: функция предназначена для создания дескриптора
//             ожидающего таймера и установки его статических
//             параметров

 HANDLE CreateWaitableTimer(
       LPSECURITY_ATTRIBUTES sattr, // [in] атрибуты безопасности
       BOOL                  reset, // [in] тип сброса таймера
       LPCTSTR               tname  // [in] имя таймера
                           );
// Код возврата: в случае успешного завершения функция
//       возвращает дескриптор вновь созданного или уже
//       существующего с таким именем ожидающий таймер; в
//       последнем случае функция GetLastError  вернет
//       значение ERROR_ALREADY_EXISTS
// Примечание:- если  для параметра sattr установлено значение
//        NULL, то атрибуты безопасности устанавливаются по
//        умолчанию  и дескриптор не наследуется дочерними
//        процессами;
//        - если  значение параметра reset равно TRUE, то  это
//        таймер с ручным сбросом, если FALSE – то это таймер
//        синхронизации;
//        - если  значение параметра tname равно NULL, то
//        создается безымянный таймер, иначе этот параметр
//        указывает на строку с именем ожидающего таймера.
```

Рисунок 6.7.1. Функция CreateWaitableTimer

Создание ожидающего таймера и установка его статических параметров осуществляется функцией CreateWaitableTimer (рисунок 6.7.1).

Для перевода ожидающего таймера в активное состояние и его установки динамических параметров предназначена функция SetWaitableTimer (рисунок 6.7.2).

```
// -- установить ожидающий таймер
// Назначение: функция предназначена для перевода таймера в
//             в активное состояние и установки его параметров

    BOOL SetWaitableTimer(
    HANDLE                hWTimer, //[in] дескриптор таймера
    const LARGE_INTEGER   DueTime, // [in] время срабатывания
    LONG                  lPeriod, // [in] период времени
    PTIMERAPCROUTINE      apcFunc, // [in] процедура завершения
    LPVOID                prmFunc, // [in] параметр процедуры
    BOOL                  ofPower  // [in] управление питанием
                          );

// Код возврата: в случае успешного завершения функция
//       возвращает ненулевое значение, иначе возвращается
//       значение FALSE
// Примечание:-  параметр DueTimer  содержит адрес структуры
//        типа LARGE_INTEGER (целое число размером  64 бита),
//        если это число положительное, то его значение
//        указывает абсолютное время перехода таймера в
//        сигнальное состояние; если число отрицательное,
//        считается, что задан интервал времени от текущего
//        системного времени; время задается в  единицах
//        равных 100 наносекунд (10^(-7)сек).
//        - значение lPeriod  определяет, является ли таймер
//        периодическим: если его значение равно нулю, то
//        таймер  не периодический; если значение больше нуля
//        то таймер периодический и lPeriod задает период в
//        миллисекундах;
//        - параметр apcFunc должен указывать на функцию
//        завершения, которая устанавливается в очередь
//        асинхронных  процедур после перехода  таймера в
//        сигнальное состояние; если  для параметра apcFunc
//        установлено значение NULL, то функция завершения
//        не используется;
//        - параметр prmFunc может содержать единственный
//        аргумент, который может быть передан функции
//        завершения;
//        - параметр ofPower устанавливает режим управления
//        питанием: если установлено значение TRUE, то после
//        после перехода таймера в сигнальное состояние
//        компьютер переключается в режим экономии
//        электроэнергии; если установлено FLASE, то
//        переключения не происходит
```

Рисунок 6.7.2. Функция SetWaitableTimer

Если создан поименованный таймер, то он может быть использован в контексте другого процесса с помощью функции OpenWaitableTimer (рисунок 6.7.3). Для перевода таймера в неактивное состояние применяется

функция CancelWaitableTimer (рисунок 6.7.4). Для ожидания сигнального состояния таймера используется универсальная функция WaitForSingleObject (рисунок 6.7.5), о которой уже упоминалось выше, в связи с ожиданием завершения потока.

```
// -- открыть существующий ожидающий таймер
// Назначение: функция предназначена для создания дескриптора
//             существующего ожидающего таймера

    HANDLE OpenWaitableTimer(
          DWORD          raccs,  //[in] режимы доступа
          BOOL           rinht,  //[in] режим наследования
          LPCTSTR        tname   //[in] имя таймера
                          );

// Код возврата: в случае успешного завершения функция
//       возвращает дескриптор ожидающего таймера, иначе
//       возвращается значение NULL
// Примечание:-  параметр raccs  может принимать любую
//       комбинацию следующих флагов:
//       TIMER_ALL_ACCESS – произвольный доступ к таймеру,
//       TIMER_MODIFY_STATE – можно только изменять состояние,
//       SYNCHRONIZE – можно использовать только в функциях
//       ожидания;
//       - если параметр rinht установлен в значение FALSE, то
//       дескриптор таймера не наследуется дочерними
//       процессами, если установлено значение TRUE, то
//       осуществляется наследование.
```

Рисунок 6.7.3. Функция OpenWaitableTimer

```
// -- отменить ожидающий таймер
// Назначение: функция предназначена для перевода таймера
//             в пассивное состояние

 BOOL CancelWaitableTimer(
               HANDLE  hTimer  //[in] дескриптор ожидающего таймера
                          );

// Код возврата: в случае успешного завершения функция
//       возвращает ненулевое значение, иначе возвращается
//       значение NULL
```

Рисунок 6.7.4. Функция CancelWaitableTimer

Следует отметить, что для ожидания сигнального состояния ожидающего таймера, можно использовать и другие функции. Например, если используется несколько ожидающих таймеров (или других объектов

синхронизации), то применяется функция WaitForMultipleeObjects [4, 14]. В параметрах этой функции можно указать массив дескрипторов объектов синхронизации и различные режимы ожидания и проверки сигнального состояния. Если кроме того, требуется исполнять асинхронные процедуры, то можно использовать функции WaitForSingleObjectEx и WaitForMultipleeObjectsEx.

```
// -- отменить ожидающий таймер
// Назначение: функция предназначена для перевода таймера
//             в пассивное состояние

 DWORD WaitForSingleObject(
             HANDLE  hTimer //[in] дескриптор ожидающего таймера
             DWORD   mstout //[in] временной интервал в миллисекундах
                          );

// Код возврата: в случае успешного завершения функция
//     возвращает одно из следующих значений:
//     WAIT_OBJECT_0 – таймер в сигнальном состоянии;
//     WAIT_TIME_OUT – истек интервал времени, таймер не
//     в сигнальном состоянии;
// Примечание: если значение параметра mstout равно нулю, то
//     не осуществляется приостановка потока, а только
//     осуществляется проверка сигнального состояния таймера;
//     если значение mstout больше нуля, то осуществляется
//     приостановка потока пока не будет исчерпан заданный
//     интервал времени или таймер не перейдет в сигнальное
//     состояние; если значение mstout  равно INFINITE, то
//     сигнальное состояние ожидается бесконечно долго.
```

Рисунок 6.7.5. Функция WaitForSingleObject

С помощью функции SetWaitableTimer может быть установлена функция завершения, которая устанавливается в очередь асинхронных процедур после перехода ожидающего таймера в сигнальное состояние. Функция завершения должна иметь такой же прототип, как и у асинхронных процедур. Прототип и механизм вызова асинхронных процедур уже рассматривался выше

Если перейти к рассмотрению модели ConcurrentServer, то, по всей видимости, ожидающий таймер будет создаваться потоком AcceptServer для каждого запущенного потока EchoSever. Сразу же после перехода одного из ожидающих таймеров, в очередь асинхронных процедур потока AcceptServer будет поставлена соответствующая процедура завершения.

На рисунке 6.7.6 изображена схема использования ожидающего таймера в модели параллельного сервера ConcurrentServer. Следует обратить внимание на новое применение асинхронной процедуры

ASFinishMessage, назначение которой рассматривалось выше. Теперь, в рамках этой асинхронной функции отменяется ожидающий таймер.

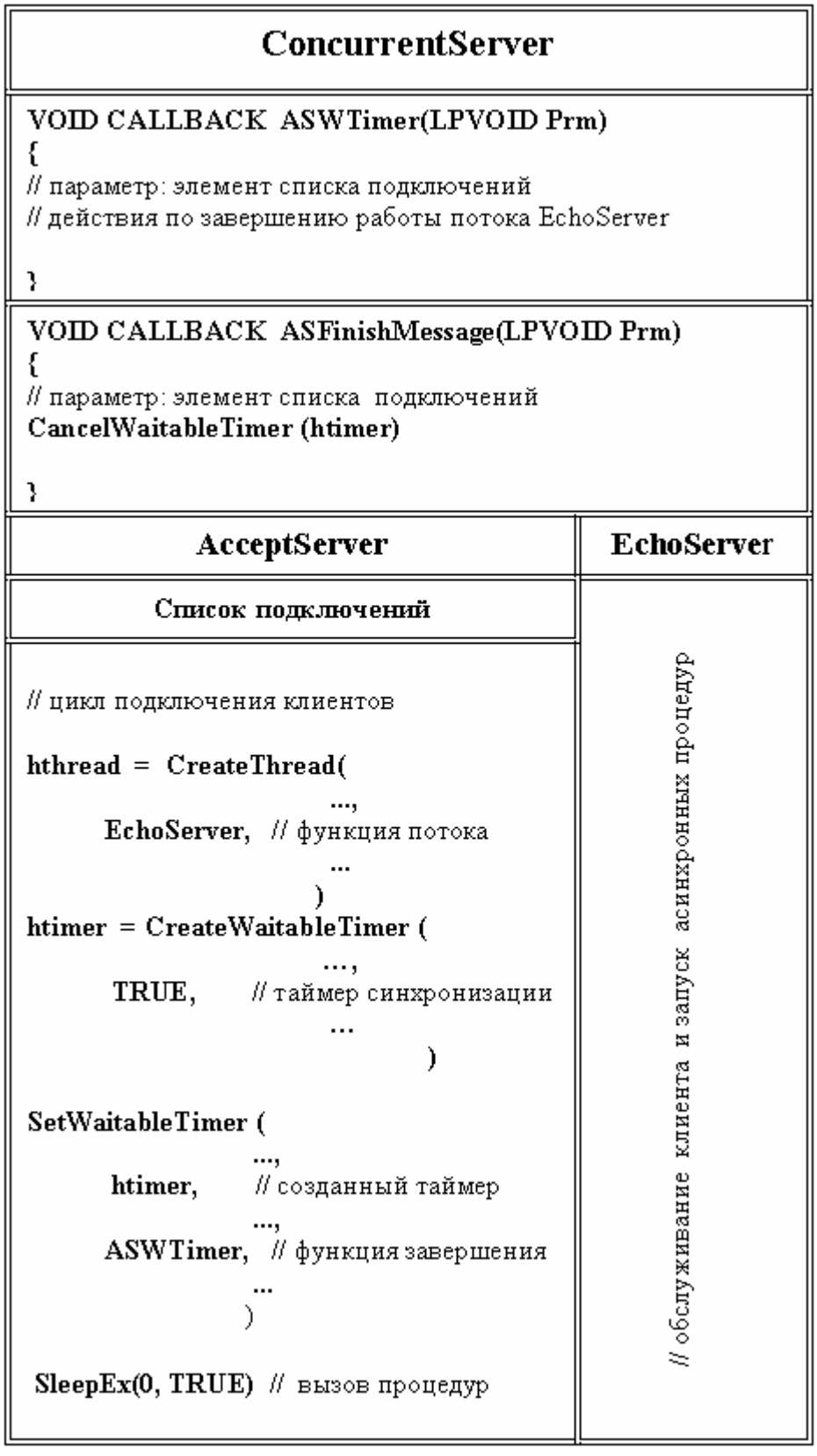

Рисунок 6.7.6. Схема использования ожидающего таймера в модели ConcurrentServer

На рисунке 6.7.6 обозначен список подключений. Структура элементов этого списка рассматривалась выше. Основное предназначение списка – это хранение информации о каждом подключении. В модели сервера ConcurrentServer предполагается хранить дескрипторы обрабатывающего потока и ожидающего таймера в элементах списка подключений. Так как асинхронные процедуры в модели всегда связаны с конкретным подключением, то удобно передавать в качестве параметра этим процедурам соответствующий элемент списка подключений.

В заключении следует отметить, что как и все дескрипторы (HANDEL) операционной системы Windows, неиспользуемые дескрипторы ожидающего таймера должны закрываться с помощью функции CloseHandle.

6.8. Применение атомарных операций

Иногда параллельным потокам необходимо выполнять некоторые несложные действия над общими переменными, исключая совместный доступ к этим переменным. Если в этом случае использовать критические секции или другие механизмы синхронизации, то может оказаться, что затраты на синхронизацию потоков значительно превысят затраты на выполнение самих операций. В таких случаях применяют блокирующие функции. Блокирующие функции выполняют несколько элементарных операций, которые объединяются в одну неделимую операцию, называемую ***атомарной операцией***.

В таблице 6.8.1 приведены четыре блокирующие функции, используемые в операционных системах семейства Windows.

Таблица 6.8.1

Наименование Функции	Назначение
`InterlockedCompareExchange`	Сравнить и заменить значение
`InterlockedDecrement`	Уменьшить значение на единицу
`InterlockedExchange`	Заменить значение
`InterlockedExchangeAdd`	Изменить значение
`InterlockedIncrement`	Увеличить значение на единицу

Все перечисленные функции требуют, чтобы адреса переменных были выровнены на границу слова, т.е. были кратны четырем. Для такого выравнивания достаточно, чтобы переменная была объявлена в программе со спецификаторами типов long, unsigned long или LONG, ULONG, DWORD.

Функция InterlockedExchange (рисунок 6.8.1) позволят установить (заменить) значение переменной. Если требуется заменить конкретное известное значение, то можно использовать функцию InterlockedCompareExchange (рисунок 6.8.2). Функции InterlockedIncrement и InterlockedDecrement (рисунки 6.8.3 и 6.8.4) используются для увеличения

или уменьшения значения переменной на единицу. Функция InterlockedExchangeAdd (рисунок 6.8.5) позволяет увеличить или уменьшить значение переменной на заданную величину.

```
// -- заменить значение
// Назначение: функция предназначена выполнения атомарной
//             операции замены переменной

 LONG InterlockedExchange(
             LPLONG  pt,    // [in] адрес заменяемой переменной
              LONG    vl     // [in] новое значение переменной
                         );

// Код возврата: функция возвращает старое значение переменной
```

Рисунок 6.8.1. Функция InterlockedExchange

```
// -- сравнить и заменить значение
// Назначение: функция предназначена выполнения атомарной
//             операции сравнения и замены переменной в случае
//             удачного сравнения

 PVOID InterlockedCompareExchange(
                 PVOID   pt,    // [in] адрес заменяемой переменной
                 PVOID   vl     // [in] новое значение переменной
                 PVOID   cp     // [in] значение для сравнения
                                   );

// Код возврата: функция возвращает старое значение переменной
```

Рисунок 6.8.2. Функция InterlockedCompareExchange

```
// -- увеличить значение на единицу
// Назначение: функция предназначена выполнения атомарной
//             операции инкремента

 LONG InterlockedIncrement(
                 LPLONG  pt,    // [in] адрес изменяемой переменной
                          );

// Код возврата: функция возвращает старое значение переменной
```

Рисунок 6.8.3. Функция InterlockedIncrement

```
// -- уменьшить значение на единицу
// Назначение: функция предназначена выполнения атомарной
//             операции декремента

 LONG InterlockedDecrement(
                 LPLONG  pt,    // [in] адрес изменяемой переменной
                          );

// Код возврата: функция возвращает старое значение переменной
```

Рисунок 6.8.4. Функция InterlockedDecrement

```
// -- изменить значение
// Назначение: функция предназначена выполнения атомарной
//             операции изменения значения переменной на
//             заданную величину

 LONG InterlockedExchangeAdd(
                 LPLONG  pt,    // [in] адрес изменяемой переменной
                 LONG    vl      // [in] прибавляемое значение
                             );

// Код возврата: функция возвращает старое значение переменной
```

Рисунок 6.8.4. Функция InterlockedExchangeAdd

В модели параллельного сервера ConcurrentServer атомарные операции могут быть использованы, например, для управления количеством одновременно обслуживаемых клиентов.

Предположим, что число одновременно обслуживаемых клиентов хранится в переменной с именем ClientServiceNumber, а максимальное количество одновременно работающих клиентов в глобальной переменной MaxClientServiceNumber. Тогда удобно в асинхронных процедурах ASStartMessage и ASFinishMessage выполнять увеличение и уменьшение значения переменной ClientServiceNumber. Перед очередным подключением клиента функция AcceptServer сравнивает значение переменных ClientServiceNumber и MaxClientServiceNumber и, если число подключенных клиентов достигло максимального значения, оказывает клиенту в подключении. При такой схеме работы нет параллельных потоков одновременно изменяющих переменную ClientServiceNumber, т.к. функции AcceptServer, ASStartMessage, ASFinishMessage работают в одном потоке последовательно. Более того, можно даже предусмотреть команду консоли ConsolePipe управления сервером, позволяющую уменьшать или увеличивать значение MaxClientServiceNumber, что позволяет регулировать нагрузку на весь сервер.

Предположим теперь, что в фоновом режиме работает еще один поток с именем AdvisorServer, который определенным образом оценивает нагрузку на параллельный сервер и в необходимые моменты уменьшает или увеличивает значение MaxClientServiceNumber. В этом случае параллельная работа с консолью уже не возможна, т.к. возможно несогласованное изменение переменной MaxClientServiceNumber.

Проблема возникнет и в том случае, если еще усложнить работу сервера и добавить еще один поток AcceptServer (предположим прослушивающего другой порт) или разрешить подключение более одной консоли (увеличив количество экземпляров именованного канала). В этом случае значения переменных ClientServiceNumer и MaxClientServiceNumer станут непредсказуемыми.

6.9. Не блокирующий режим работы сокета

В описанной выше модели параллельного сервера ConcurrentServer на функцию потока AcceptServer возложена обязанность подключения клиентских приложений, запуска обслуживающего потока (и функции) EchoServer, а также исполнение некоторых команд управления сервером. Остановимся подробнее на процессе обработки команд управления.

Для подключения клиентских программ AcceptServer использует функцию Winsock2 accept, которая приостанавливает работу потока AcceptServer до момента подключения клиента. После того, как клиентская программа выполнит функцию connect, функция accept сервера сформирует новый сокет для обмена данными между сервером и клиентом, а также возобновит исполнение потока AcceptServer.

Для ввода команд сервера (в том числе и исполняемых потоком AcceptServer) в модели ConcurrentServer предназначается функция потока ConsolePipe, которая поддерживает именованный канал с клиентом консоли RConsole.

Предположим, что с консоли управления введена команда stop, предназначенная для приостановки сервером подключения новых клиентских программ до выдачи команды start, которая должна возобновить возможность подключения клиентов. Если в момент ввода команды stop, поток AcceptServer был приостановлен функцией accept, то действие этой команды наступит только после подключения следующего клиента, т.к. нет возможности проверить наличие, введенной и переданной функцией ConsolePipe команды. Введенная следующая команда может “затереть” предыдущую и команда stop не выполнена вообще.

Возникшую проблему можно решить несколькими способами. Например, можно выполнить команду connect из функции ConsolePipe и этим самым заставить AcceptServer сделать цикл и проверить наличие команды управления. В этом случае будет необходим механизм, позволяющий различать подключение от ConsolePipe и других клиентов. Можно было бы реализовать подключение консоли не через именованный канал, а также как клиентов – через сокет. В последнем случае, также как и в

предыдущем случае требуется различать подключение консоли и остальных клиентов.

Рассмотрим еще один способ решения проблемы управления процессом подключения клиентов в потоке AcceptServer. Этот способ основан на специальном режиме работы сокета.

Алгоритм работы функции accept, который рассматривался выше, был обусловлен ***режимом блокировки (blocking mode)***, установленным (по умолчанию) для сокета. Переключение сокета в ***режим без блокировки (nonblocking mode)***, позволяет избежать приостановки программы. В режиме без блокировки выполнение accept, не приостанавливает выполнение потока, как это было прежде, а возвращает значение нового сокета, если обнаружен запрос на создание канала (функция connect, выполненная клиентом), или значение INVALID_SOCKET, если запроса на создание канала нет в очереди запросов или возникла ошибка. Для того, чтобы отличить ошибку от отсутствия запроса, используется уже рассмотренная выше функция WSAGetLastError, которая в последнем случае возвращает значение WSAEWOULDBLOCK.

Для переключения режимов сокета применяется функция ioctlsocket, входящая в состав Winsock2 (рисунок 6.9.1).

```
// -- переключить режим работы сокета
// Назначение: функция предназначена для управления режимом
//             ввода/вывода сокета

 int ioctlsocket(
            SOCKET  sock, // [in] дескриптор сокета
            long    scmd, // [in] команда, применяемая к сокету
             u_long*  pprm  // [in,out] указатель на параметр команды
                );
// Код возврата: в случае успешного выполнения функция
//        возвращает нулевое значение, иначе возвращается
//        значение SOCKET_ERROR
// Примечание: параметр scmd может принимать следующие
//        значения: FIONBIO, FIONREAD, SIOCATMARK; команда
//        FIONBIO применяется для переключения режимов:
//        blocking/nonblocking; для установки режима
//        nonblocking значение параметра  *pprm должно быть
//        отличным от нуля.
```

Рисунок 6.9.1. Функция ioctlsocket

На рисунке 6.9.2 представлен фрагмент функции потока AcceptServer в котором используется функция ioctlsocket для переключения сокета в режим без блокировки. После успешного выполнения функции ioctlsocket вызывается функция CommandsCycle, фрагменты которой приведены на рисунке 6.9.3. Предполагается, что команды консоли поток AcceptServer выбирает из области памяти, адрес которой передается в качестве параметра

функции AcceptServer. Тип TalkersCommand, к которому преобразуется параметр функции потока, является перечислением (тип enum) команд применяемых для управления сервером.

```
DWORD WINAPI AcceptServer(LPVOID  pPrm)  //  сервер ожидания запроса
{
//.............................................................
try
 {
  //... WSAStartup(...),sS = socket(...),bind(sS, ...),listen(sS,...),..

  u_long nonblk;
  if (ioctlsocket( sS, FIONBIO, &(nonblk = 1)) == SOCKET_ERROR)
          throw SetErrorMsgText("ioctlsocket:",WSAGetLastError());
  CommandsCycle(*((TalkersCommand*)pPrm));

  //...closesocket(sS),WSACleanup()...................................
 }
 //.............................................................
 ExitThread(*(DWORD*)pPrm);                 // завершение потока
 }
```

Рисунок 6.9.2. Пример применения функции ioctlsocket

```
void CommandsCycle(TalkersCommand& cmd)      // цикл обработки команд
{
int  squirt = 0;
while(cmd  != EXIT)     // цикл обработки команд консоли и подключений
{
 switch (cmd)
 {
 //.....................................................
 case START: cmd = GETCOMMAND; // возобновить подключение клиентов
             squirt =  AS_SQUIRT; //#define AS_SQUIRT 10
             break;
 case STOP:  cmd = GETCOMMAND; // остановить подключение клиентов
             squirt = 0;
             break;
 //.....................................................
 };
 if (AcceptCycle(squirt))   //цикл  запрос/подключение (accept)
   {
    cmd = GETCOMMAND;
    //.... запуск потока EchoServer.....................
    //.... установка ожидающего таймера для процесса EchoServer ...
   }
 else SleepEx(0, TRUE);    // выполнить асинхронные процедуры
};
};
```

Рисунок 6.9.3. Пример обработки команд stop, start и exit

Функция CommandCycle (рисунок 6.9.3) предназначена для обработки команд управления сервером и подключения клиентов. Подключение клиентов осуществляется функцией AcceptCycle (рисунок 6.9.4). Изображенный фрагмент функции CommandCycle содержит цикл обработки команд stop (остановить подключение клиентов), start (возобновить подключение клиентов) и exit (завершение работы потока AcceptServer), а также функцию AcceptCylcle, предназначенную для подключения клиентов. Если команда принята функцией на обработку, то устанавливается новое значение команды getcommand, обозначающее, что функция AcceptCycle готова к приему новой команды управления.

В случае успешного подключения клиента функция AcceptCylcle возвращает значение true. Значение squirt, передаваемое в качестве параметра функции AcceptCylcle является максимальным количеством итераций выполнения функции accept (в режиме без блокировки) для подключения клиента за один вызов функции AcceptCylcle. Если клиент подключился, то для него запускается поток EchoServer, устанавливается ожидающий таймер, проверяется и увеличивается счетчик подключений и т.п. Если клиент не подключился, то осуществляется запуск асинхронных процедур с помощью функции SleepEx.

```
bool AcceptCycle(int squirt)
{
bool rc = false;
Contact c(Contact::ACCEPT,"EchoServer");

while(squirt-- > 0 && rc == false)
{
if ((c.s = accept(sS,
                  (sockaddr*)&c.prms, &c.lprms)) == INVALID_SOCKET)
 {
  if (WSAGetLastError() != WSAEWOULDBLOCK)
          throw  SetErrorMsgText("accept:",WSAGetLastError());
 }
else
 {
  rc = true;                // подключился
  EnterCriticalSection(&scListContact);
  contacts.push_front(c);
  LeaveCriticalSection(&scListContact);
  }
}
return rc;
};
```

Рисунок 6.9.4. Пример применения функции accept в режиме без блокировки сокета

После каждого вызова функции accept для сокета в режиме без блокировки (рисунок 6.9.4), следует проверять код возврата на равенство

значению WSAEWOULDBLOK. Это значение возвращается в том случае, если очередь подключений пуста. В функции AcceptCycle организован цикл проверки очереди подключений повторяющийся squirt раз.

6.10. Использование приоритетов

Производительность параллельного сервера в значительной степени зависит от правильного распределения ресурсов между конкурирующими потоками. Важнейшим ресурсом при этом является процессорное время.

По принципу обслуживания потоков операционная система Windows относится к классу систем с вытесняющей многозадачностью [15]. Каждому потоку операционной системы периодически выделяется квант процессорного времени. Величина кванта, выделяемая потоку, зависит от типа операционной системы Windows, типа процессора и приблизительно равна двадцати миллисекундам. По истечении кванта времени исполнение текущего потока прерывается, контекст потока сохраняется и процессор передается потоку с высшим приоритетом.

Приоритеты потоков в Windows определяются относительно приоритета процесса, в рамках которого они исполняются. Приоритеты потоков изменяются от 0 (низший приоритет) до 31 (высший приоритет). Основные функции необходимые для работы с приоритетами приведены в таблице 6.10.1.

Таблица 6.10.1

Наименование функции	Назначение
`GetPriorityClass`	Получить приоритет процесса
`GetThreadPriority`	Получить приоритет потока
`GetProcessPriorityBoost`	Определить состояние режима процесса
`GetThreadPriorityBoost`	Определить состояние режима потока
`SetPriorityClass`	Изменить приоритет процесса
`SetThreadPriority`	Изменить приоритет потока
`SetProcessPriorityBoost`	Установить или отменить динамический режим потоков процесса
`SetThreadPriorityBoost`	Установить или отменить динамический режим потока

Приоритеты процессов устанавливаются при их создании функцией CreateProcess [14]. Операционная система Windows различает четыре типа процессов в соответствии с их приоритетами: фоновые процессы, процессы с нормальным приоритетом, процессы с высоким приоритетом и процессы реального времени.

Фоновые процессы выполняют свою работу, когда нет активных пользовательских процессов. Обычно эти процессы следят за состоянием системы.

Процессы с нормальным приоритетом – это обычные пользовательские процессы. Это приоритет присваивается пользовательским процессам по умолчанию. В рамках этого класса допускается небольшое понижение или повышение приоритетов процесса.

```
// -- получить приоритет процесса
// Назначение: функция предназначена для определения
//             приоритетного класса процесса

 DWORD GetPriorityClass(
                   HANDLE hP    // [in] дескриптор процесса
                        );
// Код возврата: в случае  успешного  выполнения   функция
//    возвращает  одно из следующих значений, обозначающих
//    приоритетный класс процесса:
//    IDLE_PRIORITY_CLASS – фоновый процесс;
//    BELOW_NORMAL_PRIORITY_CLASS – ниже нормального;
//    NORMAL_PRIORITY_CLASS – нормальный процесс;
//    ABOVE_NORMAL_PRIORITY_CLASS – выше нормального;
//    HIGH_NORMAL_PRIORITY_CLASS – высокоприоритетный процесс;
//    REAL_NORMAL_PRIORITY_CLASS – процесс реального времени;
//    иначе возвращается значение NULL.
```

Рисунок 6.10.1. Функция GetPriorityClass

```
// -- изменить приоритет процесса
// Назначение: функция предназначена для изменения приоритета
//             процесса

 BOOL SetPriorityClass(
                  HANDLE hP,    // [in] дескриптор процесса
                  DWORD  py    // [in] новый приоритет процесса
                        );
// Код возврата: в случае  успешного  выполнения   функция
//    возвращает ненулевое  значение,  иначе  возвращается
//    значение FALSE.
// Примечание: параметр py может принимать одно из следующих
//    значений:
//    IDLE_PRIORITY_CLASS – фоновый процесс;
//    BELOW_NORMAL_PRIORITY_CLASS – ниже нормального;
//    NORMAL_PRIORITY_CLASS – нормальный процесс;
//    ABOVE_NORMAL_PRIORITY_CLASS – выше нормального;
//    HIGH_NORMAL_PRIORITY_CLASS – высокоприоритетный процесс;
//    REAL_NORMAL_PRIORITY_CLASS – процесс реального времени.
```

Рисунок 6.10.2. Функция SetPriorityClass

Процессы с высоким приоритетом это тоже пользовательские процессы, от которых требуется более быстрая реакция на некоторые события. Обычно такие приоритеты имеют программные системы, работающие на платформе операционной системы Windows.

Процессы реального времени обычно работают непосредственно с аппаратурой компьютера и продолжительность их работы ограниченно отведенным временем реакции на внешние события.

Узнать приоритет процесса можно с помощью функции GetPriorityClass (рисунок 6.10.1), а изменить с помощью функции SetPriorityClass (рисунок 6.10.2). Эти функции используют в качестве параметра дескриптор или псевдодескриптор процесса. Псевдодескриптор текущего процесса может быть получен с помощью функции GetCurrentProcess (рисунок 6.10.3).

```
// -- получить псевдодескриптор процесса
// Назначение: функция предназначена для получения
//             псевдодескриптора текущего процесса

 HANDLE GetCurrentProcess(VOID);

// Код возврата: в случае успешного выполнения функция
//    возвращает псевдодескриптор текущего процесса, иначе
//    возвращается значение FALSE.
```

Рисунок 6.10.3. Функция GetCurrentProcess

При диспетчеризации процессов и потоков квант времени выделяется потоку. Приоритет потока, который учитывается операционной системой при выделении процессорного времени, называется ***базовым*** или ***основным приоритетом потока***. Всего существует 32 базовых приоритетов – от 0 до 31. Для каждого базового приоритета существует очередь потоков. При диспетчеризации квант процессорного времени выделяется потоку, который стоит первым в очереди с наивысшим приоритетом. Базовый приоритет потока определятся исходя из приоритета процесса и уровня приоритета потока. Уровень приоритета потока может быть: низший, ниже нормального, нормальный, выше нормального, высший приоритет, приоритет фонового потока и приоритет потока реального времени. Значения базовых приоритетов потоков можно определить, воспользовавшись таблицей 6.10.2.

При создании потока его базовый приоритет устанавливается как сумма приоритета процесса, в контексте которого этот поток выполняется, и значения THREAD_PRIORITY_NORMAL (0), соответствующего нормальному уровню приоритета потока. Значение уровня приоритета потока может быть изменено с помощью функции SetThreadPriority

(рисунок 6.10.4), а узнать текущий уровень приоритета потока – с помощью функции GetThreadPriority (рисунок 6.10.5).

```
// -- изменить приоритет потока
// Назначение: функция предназначена для изменения приоритета
//             потока

 BOOL SetThreadPriority(
                     HANDLE hT,    // [in] дескриптор потока
                     DWORD  py     // [in] новый приоритет потока
                       );

// Код возврата: в случае  успешного  выполнения   функция
//    возвращает ненулевое  значение,  иначе  возвращается
//    значение FALSE.
// Примечание: параметр py может принимать одно из следующих
//    значений:
//    THREAD_PRIORITY_LOWEST – низший приоритет;
//    THREAD_PRIORITY_BELOW_NORMAL – ниже среднего;
//    THREAD_PRIORITY_NORMAL – нормальный;
//    THREAD_PRIORITY_ABOVE_NORMAL – выше нормального;
//    THREAD_PRIORITY_HIGHEST – высший приоритет;
//    THREAD_PRIORITY_IDLE – фоновый поток;
//    THREAD_PRIORITY_TIME_CRITICAL – поток реального времени.
```

Рисунок 6.10.4. Функция SetThreadPriority

```
// -- получить приоритет потока
// Назначение: функция предназначена для получения приоритета
//             потока

 DWORD GetThreadPriority(
               HANDLE hT,    // [in] дескриптор потока
                       );

// Код возврата: в случае  успешного  выполнения   функция
//    возвращает одно из следующих значений:
//    THREAD_PRIORITY_LOWEST – низший приоритет;
//    THREAD_PRIORITY_BELOW_NORMAL – ниже среднего;
//    THREAD_PRIORITY_NORMAL – нормальный;
//    THREAD_PRIORITY_ABOVE_NORMAL – выше нормального;
//    THREAD_PRIORITY_HIGHEST – высший приоритет;
//    THREAD_PRIORITY_IDLE – фоновый поток;
//    THREAD_PRIORITY_TIME_CRITICAL – поток реального времени;
//    иначе возвращается   значение FALSE.
```

Рисунок 6.10.5. Функция GetThreadPriority

Таблица 6.10.2

Приоритет потока	Приоритет процесса					
	реального времени	высокий	выше норм.	нормальный.	ниже норм.	фоновый.
реального времени	31	15	15	15	15	15
высший	26	15	12	10	8	6
выше норм.	25	14	11	9	7	5
нормальный	24	13	10	8	6	4
ниже норм.	23	12	9	7	5	3
низший	22	11	8	6	4	2
фоновый	16	1	1	1	1	1

Если базовый приоритет потока находится в пределах от 0 до 15, то он может изменяться динамически операционной системой. При получении потоком сообщения Windows или при переходе его в состояние готовности система повышает его приоритет на 2. В процессе выполнения базовый приоритет такого потока понижается на единицу после каждого отработанного кванта, но не ниже базового значения.

```
// -- установить или отменить динамический режим всех потоков
//    процессов
// Назначение: функция предназначена для установки или отмены
//             динамического изменения базового приоритета
//             всех потоков процесса

 BOOL SetProcessPriorityBoost(
                        HANDLE hP, // [in] дескриптор процесса
                        BOOL   de  // [in] состояние режима
                               );

// Код возврата: в случае успешного выполнения функция
//    возвращает ненулевое значение, иначе возвращается FALSE.
// Примечание: если значение параметра de установлено в TRUE,
//    то режим динамического изменения базового приоритета
//    для потоков процесса запрещается; если значение de
//    установлено в FALSE – режим разрешается.
```

Рисунок 6.10.6. Функция SetProcessPriorityBoost

Для управления режимом динамического изменения базового приоритета потока могут быть использованы функции SetProcessPriorityBoost и SetThreadPriorityBoost (рисунки 6.10.5 и 6.10.6). С помощью функции SetProcessPriorityBost осуществляется отмена или возобновление режима динамического изменения базового приоритета всех

потоков, работающих в контексте общего процесса. Для изменения режима только для одного потока применяется функция SetThreadPriorityBoost.

```
// -- установить или отменить динамический режим потока
// Назначение: функция предназначена для установки или отмены
//             динамического изменения базового приоритета
//             потока

 BOOL SetThreadPriorityBoost(
                        HANDLE hP,  // [in] дескриптор потока
                        BOOL   de   // [in] состояние режима
                             );

// Код возврата: в случае  успешного  выполнения   функция
//    возвращает ненулевое значение, иначе возвращается FALSE.
// Примечание: если значение параметра de установлено в TRUE,
//    то режим динамического изменения базового приоритета
//    для потока запрещается; если значение de    установлено
//    в FALSE – режим разрешается.
```

Рисунок 6.10.7. Функция SetThreadPriorityBoost

Определить состояние режима динамического изменения приоритетов потока можно с помощью функций GetProcessPriorityBoost (рисунок 6.10.8) и GetThreadPriorityBoost (рисунок 6.10.9).

```
// -- определить состояние  режима процесса
// Назначение: функция предназначена для определения состояния
//             режима динамического изменения базового
//             приоритета   всех потоков процесса

 BOOL GetProcessPriorityBoost(
                        HANDLE hP, // [in] дескриптор процесса
                        PBOOL  de  // [out] состояние режима
                              );

// Код возврата: в случае  успешного  выполнения   функция
//    возвращает ненулевое значение, иначе возвращается FALSE.
// Примечание: если значение параметра de установлено в TRUE,
//    то режим динамического изменения базового приоритета
//    для потоков процесса запрещен; если значение de
//    установлено в FALSE – режим разрешен.
```

Рисунок 6.10.8. Функция GetProcessPriorityBoost

```
// -- определить состояние  режима потока
// Назначение: функция предназначена для определения состояния
//             режима динамического изменения базового
//             приоритета  потока

 BOOL GetThreadPriorityBoost(
                        HANDLE hT, // [in] дескриптор потока
                        PBOOL  de  // [out] состояние режима
                              );

// Код возврата: в случае  успешного  выполнения   функция
//    возвращает ненулевое значение, иначе возвращается FALSE.
// Примечание: если значение параметра de установлено в TRUE,
//    то режим динамического изменения базового приоритета
//    для потока запрещен; если значение de  установлено
//    в FALSE – режим разрешен.
```

Рисунок 6.10.9. Функция GetThreadPriorityBoost

В модели параллельного сервера ConcurrentServer основная конкуренция за процессорное время происходит между потоком AcceptServer, обрабатывающими потоками (потоки EchoServer), и потоками GarbageCleaner и ConsolePipe. Принцип распределения приоритетов между этими потоками, зависит от стратегии, преследуемую разработчиком. Очевидным является только то, что с пониженным приоритетом (в фоновом режиме) должен работать поток GarbageCleaner, который моделирует внутренний процесс сервера по сборке мусора. С повышением приоритета потока AcceptServer, по-видимому, более активным станет подключение клиентов, с повышением приоритетов обрабатывающих потоков клиенты будут быстрее обслуживаться и отключаться от сервера.

Можно говорить, о системе управлении приоритетами, которая бы периодически оценивала статистику работы сервера и самостоятельно перестраивала распределение приоритетов. Например, если в очередь подключений растет, то может быть следует увеличить приоритет потока AcceptServer, или, наоборот, если подключения редки, но работает много обслуживающих потоков, то следует повысить приоритет последних и понизить приоритет всех остальных. И, наконец, можно просто назначать приоритеты с консоли управления

6.11. Обработка запросов клиента

До сих пор предполагалось, что параллельный сервер исполняет однотипные запросы клиентов. При подключении клиента, запускался определенный (известный) поток, который обслуживал данное соединение. В реальности часто бывает, что заранее не известно, какого рода услуга будет запрошена клиентом у сервера.

Обычно сервер должен распознавать некоторое количество команд (запросов), которые клиент может направить в его адрес. Каждая команда идентифицируется своим кодом и может содержать определенный набор операндов. Например, часто первая команда, которую обрабатывает сервер, является команда **connect (**или, что-то подобное**),** в которой указывается запрашиваемый клиентом сервис и другие дополнительные параметры.

Поступающая на вход сервера команда должна пройти лексический, синтаксический и (если необходимо) семантический анализ, и только после этого может быть исполнена сервером. С методами построения лексических, синтаксических и семантических анализаторов можно ознакомиться в [16, 17]. Здесь эти вопросы рассматриваться не будут.

Рассмотрим принципы построения параллельного сервера, обрабатывающего несколько различных запросов клиента на примере, уже рассмотренной выше, модели ConcurrentServer.

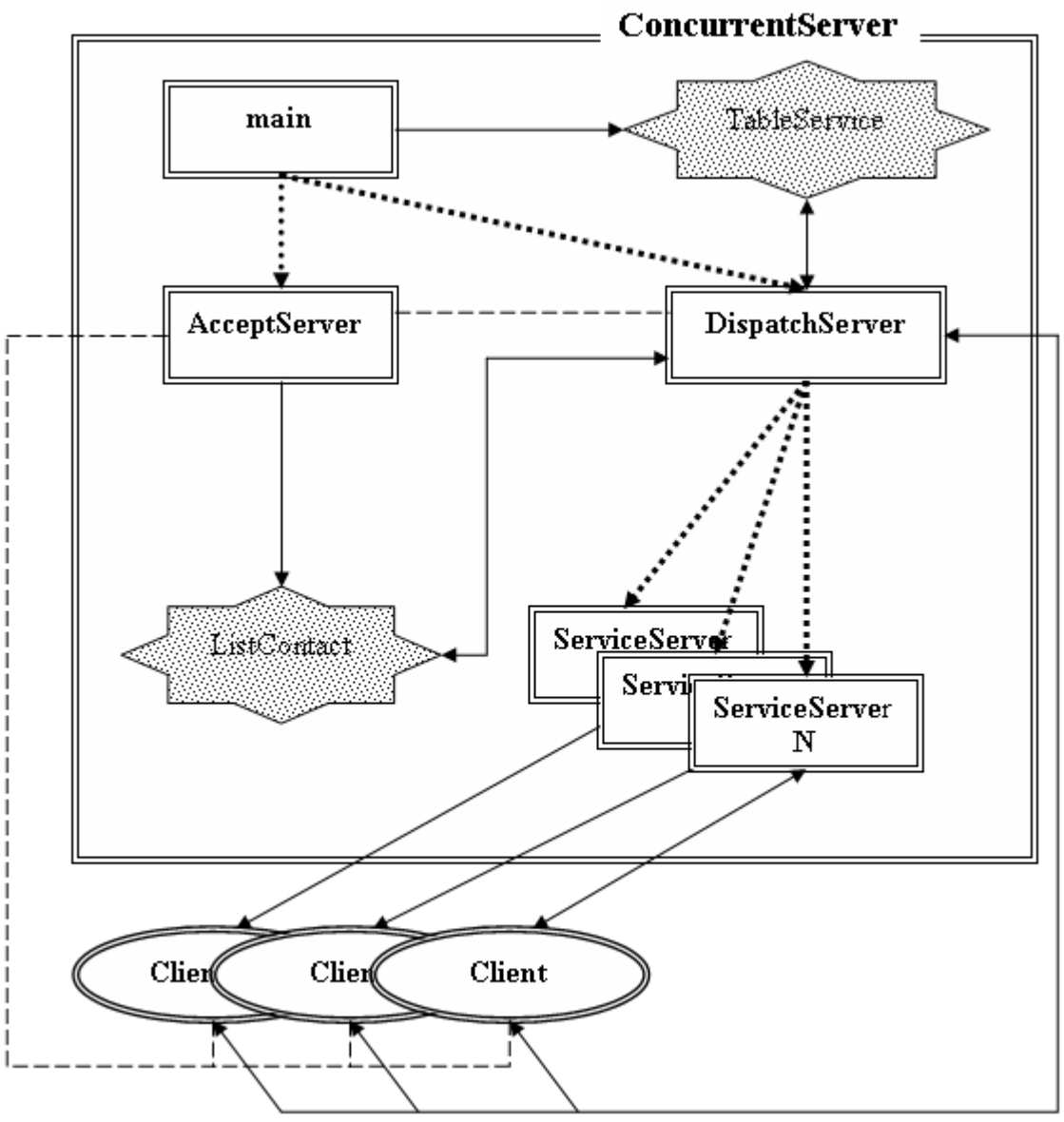

Рисунок 6.11.1. Структура параллельного сервера с диспетчером запросов

На рисунке 6.11.1 изображена структура параллельного сервера ConcurrentServer, обслуживающего разнотипные запросы. Для простоты здесь не изображены некоторые потоки, которые уже рассматривались ранее. Как и прежде сплошными направленными линиями изображается движение информации, пунктирными запуск потока, а штриховой − процедуры подключения и синхронизации.

Основным отличием новой структуры, является промежуточный поток DispatchServer между AcceptServer и обслуживающими потоками ServiceServer (раньше это был единственный поток называемый EchoServer). Теперь предполагается, что сначала программа клиента осуществляет процедуру подключения (для этого используется поток AcceptServer), потом поток DispatchServer принимает от клиента запрос (команду) на обслуживание и после этого уже запускается соответствующий поток ServiceServer, который исполняет команду и в случае необходимости обменивается данными с клиентом.

Введение промежуточного звена, обуславливается тем, что после этапа подключения, клиент (в общем случае) может достаточно долго не запрашивать у сервера услугу (не выполнять функцию send, пересылающую команду сервера). Ожидать поступление команды в потоке AcceptServer не целесообразно, т.к. его основное назначение − подключение клиентов.

На поток DispatchServer возлагается прием первой команды от клиента после подключения, содержащей идентификатор, запрашиваемого сервиса. Соответствие идентификатора и адреса потоковой функции содержится в специальной таблице TableService, который формируется при инициализации сервера (например, на основе конфигурационного файла).

Поток DispatchServer получает информацию (сокет и его параметры) о новом подключении через список ListContact (элементы списка создаются и заполняются в потоке AcceptServer). Поиск в списке ListContact нового подключения DispatchServer осуществляет после того, как поток AcceptServer сигнализирует потоку DispatchServer (на рисунке сигнал обозначен штриховой линией). Сигнал о наличии нового подключения можно выполнить с помощью, уже рассмотренного выше, механизма асинхронных процедур или с помощью механизма событий, который будет рассматриваться ниже. В любом случае поток DispatchServer должен отследить интервал времени (с помощью ожидающего таймера) от момента подключения до получения первой команды, проанализировать команду на правильность, а также запустить поток соответствующий запрошенному сервису (и) или оправить диагностирующее сообщение.

По всей видимости, увеличение функциональности сервера, потребует некоторого переосмысления процесса управления его работой. Очевидным является необходимость динамически запрещать или разрешать поддержку определенных запросов, классифицировать запросы по приоритетности, устанавливать различные интервалы.

6.12. Применение механизма событий

Выше уже упоминался механизм событий, позволяющий оповестить поток о некотором выполненном действии, произошедшем за пределами потока. Саму ***задачу оповещения*** часто называют ***задачей условной синхронизации***.

В операционных системах семейства Windows события описываются объектами ядра ***Events***. Различают два типа событий: с ручным сбросом и с автоматическим сбросом. Различие между этими типами заключается в том, что событие с ручным сбросом можно перевести в несигнальное состояние только с помощью функции ResetEvent, а событие с автоматическим сбросом переходит в несигнальное состояние как с помощью функции ResetEvent, так и при помощи функции ожидания.

Функции необходимые для работы с событиями приведены в таблице 6.12.1.

Наименование функции	Назначение
`CreateEvent`	Создать событие
`OpenEvent`	Открыть событие
`PulseEvent`	Освободить ожидающие потоки
`ResetEvent`	Перевести событие в несигнальное состояние
`SetEvent`	Перевести событие в сигнальное состояние

Для создания события используется функция CreateEvent (рисунок 6.12.1). С помощью функции OpenEvent (рисунок 6.12.2) можно получить дескриптор уже созданного события и установить некоторые его характеристики. С помощью функции SetEvent (рисунок 6.12.3) любое событие может быть переведено в сигнальное состояние. Для перевода любого события в несигнальное состояние применяется функция ResetEvent (рисунок 6.12.4). Следует отметить, что если событие с автоматическим сбросом, ждут несколько потоков, то переходе события в сигнальное состояние освобождается только один поток.

Функция PulseEvent (рисунок 6.12.5) используется для событий с ручным сбросом. Если одно и тоже событие ожидается несколькими потоками, то выполнение функции PulseEvent приводит к тому, что все эти потоки выводятся из состояния ожидания, а само событие сразу же переходит в несигнальное состояние.

В таблице не приводиться, уже рассмотренная выше, универсальная функция WaitForSingleObject, которая используется для перевода потока в состояние ожидания до получения сигнала.

Следует обратить внимание, что при создании события можно указать состояние (сигнальное или не сигнальное) в котором находится данное событие. Кроме того, задав имя события, можно обеспечить к нему доступ из другого процесса.

```
// -- создать событие
// Назначение: функция предназначена для создания  события и
//             установки его параметров

 HANDLE  CreateEvent(
      LPSECURITY_ATTRIBUTES sattr, // [in] атрибуты безопасности
      BOOL                  etype, // [in] тип события
      BOOL                  state, // [in] начальное состояние
      LPCTSTR               ename  // [in] имя события
                  );

// Код возврата: в случае  успешного  выполнения   функция
//    возвращает дескриптор события  иначе  NULL.
// Примечания:- если значение параметра sattr  установлено в
//    NULL, то значения атрибутов безопасности будут
//    установлены по умолчанию;
//    - если значение параметра etype установлено в TRUE, то
//    создается событие с ручным сбросом, иначе с -
//    автоматическим;
//    - если значение параметра state установлено в TRUE, то
//    начальное состояние события является сигнальным, иначе
//    состояние не сигнальное;
//    - параметр ename задает имя события, что позволяет
//    работать разным процессам работать с общим событием;
//    если не предполагается использовать имя события, то для
//    этого параметра может быть установлено значение NULL.
```

Рисунок 6.12.1. Функция CreateEvent

```
// -- открыть событие
// Назначение: функция предназначена для создания дескриптора
//             уже существующего поименованного события

 HANDLE  OpenEvent(
        DWORD   accss, // [in] флаги доступа
        BOOL    rinrt, // [in] режим наследования, TRUE - разрешено
        LPCTSTR ename  // [in] имя события
                  );

// Код возврата: в случае  успешного  выполнения   функция
//    возвращает дескриптор события  иначе  NULL.
// Примечания:- параметра accss  может принимать любую
//   логическую комбинацию следующих флагов:
//   EVENT_ALL_ACCESS – полный доступ к событию;
//   EVENT_MODIFY_STATE – допускается изменение состояния;
//   SYNHRONIZE – можно использовать в функциях ожидания.
```

Рисунок 6.12.2. Функция OpenEvent

```
// -- перевести событие в сигнальное состояние
// Назначение: функция предназначена перевода существующего
//             события  в сигнальное состояние

 BOOL  SetEvent(
             HANDLE   hE  // [in] дескриптор события
               );

// Код возврата: в случае  успешного  выполнения   функция
//    возвращает ненулевое значение, иначе  NULL.
```

Рисунок 6.12.3. Функция SetEvent

```
// -- перевести событие в  несигнальное состояние
// Назначение: функция предназначена перевода существующего
//             события  в несигнальное состояние

 BOOL  ResetEvent(
             HANDLE   hE  // [in] дескриптор события
               );

// Код возврата: в случае  успешного  выполнения   функция
//    возвращает ненулевое значение, иначе  NULL.
```

Рисунок 6.12.4. Функция ResetEvent

```
// -- освободить ожидающие потоки
// Назначение: функция предназначена вывода всех ожидающих
//             сигнала потоков  из состояния ожидания и
//             перевода события в несигнальное состояние

 BOOL  PulseEvent(
             HANDLE   hE  // [in] дескриптор события
                 );

// Код возврата: в случае  успешного  выполнения   функция
//    возвращает ненулевое значение, иначе  NULL.
// Примечание: функция используется только для событий с
//    ручным сбросом
```

Рисунок 6.12.5. Функция PulseEvent

6.13. Использование динамически подключаемых библиотек

При разработке параллельного сервера часто бывает полезным разделить процедуры управления сервером и обслуживания клиентов. Имеется в виду, что поддерживаемый сервером сервис может меняться при неизменной логике управления сервером. Одним из способов такого

разделения является размещения функций обслуживающих потоков в динамически подключаемой библиотеке (dll-библиотеке).

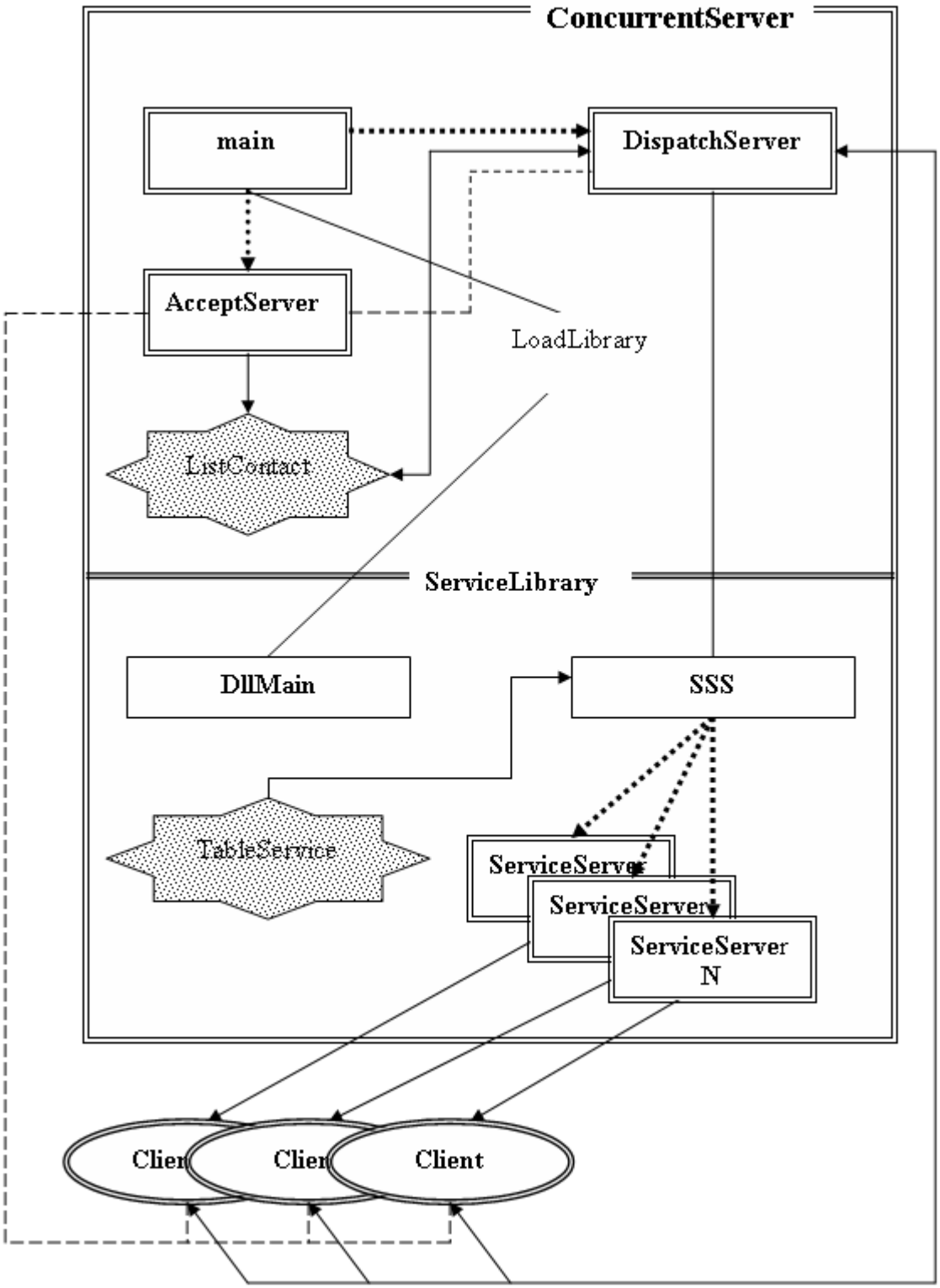

Рисунок 6.13.1. Структура параллельного сервера с динамической библиотекой

Рассмотренный выше обслуживающий разнотипные запросы сервер использовал таблицу TableService для того, чтобы сопоставить запросу

клиента функцию обслуживающего потока. Удобно поместить эту таблицу вместе с функциями обслуживающих потоков в dll-библиотеке. В этом случае сервер может динамически загружать dll-библиотеку.

Таким образом, теперь параллельный сервер состоит из двух частей: управляющий сервер и dll-библиотека функций потоков обслуживания. При такой структуре, набор сервисных услуг предоставляемых сервером зависит от версии динамической библиотеки. Местонахождение библиотеки может быть одним из параметров инициализации сервера.

На рисунке 6.13.1 изображена структура параллельного сервера, использующего динамическую библиотеку ServiceLibrary, для хранения таблицы TableService и функций обслуживающих потоков. Библиотека загружается в память при инициализации сервера. В состав библиотеки входит стандартная функция DllMain (рисунок 6.13.2), экспортируемая функция SSS для запуска потока по заданному клиентом коду команды, а также функции обслуживающих потоков.

```
// -- главная функция динамически подключаемой библиотеки
// Назначение: функция обозначает точку входа в программный
//             модуль динамически подключаемой библиотеки,
//             функции получает управление от операционной
//             системы в момент загрузки.

 BOOL  WINAPI DllMain(
             HINSTANCE hinst,    //[in] дескриптор dll
             DWORD     rcall,    //[in] причина вызова
             LPVOID    wresv     // резерв Windows
                    );

// Код возврата: в случае  успешного  завершения  функция
//    должна вернуть значение  TRUE, иначе  FALSE.
// Примечание: - в параметре hinst операционная система
//      передает дескриптор, который фактически равен
//      виртуальному адреса загруженной dll;
//    - параметр rcall может принимать одно из следующих
//      значений:
//      DLL_PROCESS_ATTACH – dll загружена в адресное
//      пространство процесса;
//      DLL_THREAD_ATTACH – dll вызывается в контексте
//      потока, созданного в рамках процесса;
//      DLL_THREAD_DETACH – завершился поток в контексте
//      которого загружена dll;
//      DLL_PROCESS_DETACH – dll – выгружается из
//      адресного пространства процесса;
//    - если параметр rcall имеет значение DLL_PROCESS_ATTACH,
//      а параметр wresv равен NULL, то библиотека загружена
//      динамически, другое значение wresv, говорит о
//      статической загрузке библиотеки.
```

Рисунок 6.13.2. Функция DllMain

Динамические библиотеки представляют собой программный модуль, который может быть загружен в виртуальную память процесса как статически, во время создания исполняемого модуля процесса, так и динамически во время исполнения процесса операционной системой. Дальнейшее изложение, в основном, будет касаться динамически загружаемых библиотек. Принципы использования статически загружаемых библиотек изложены в [12].

Для создания dll-библиотеки в среде Visual Studio необходимо выбрать проект типа Win32 Dynamic-Link Library. Как и любая программа на языке C++, динамически подключаемая библиотека имеет главную функцию, которая отмечает точку входа программы при ее исполнении операционной системой. Главная функция dll-библиотеки называется DllMain – ее шаблон автоматически создается Visual Studio.

Функции необходимые для работы с динамически подключаемыми библиотеками приведены таблице 6.13.1. Следует отметить в таблице 6.13.1 содержатся не все функции Win32 API, применяемые для работы с библиотеками. С полным перечнем функций можно ознакомиться в [4, 12, 14].

Таблица 6.13.1

Наименование функции	**Назначение**
`FreeLibrary`	Отключить dll-библиотеку от процесса
`GetProcAddress`	Импортировать функцию
`LoadLibrary`	Загрузить dll-библиотеку

Функция LoadLibrary (рисунок 6.13.3) применяется для загрузки динамической библиотеки в память компьютера. Для выгрузки библиотеки используется функция FreeLibrary (рисунок 6.13.4). Следует отметить, что если при загрузке библиотеки обнаруживается, что она уже загружена (даже другим процессом), то повторной загрузки не осуществляется. В то же время следует помнить, что для dll-библиотек используется механизм проецирования dll-библиотек, предоставляющий каждому ***клиенту библиотеки*** (так принято называть приложения использующие функции dll-библиотеки) полную независимость. Для создания общей памяти для нескольких процессов в рамках одной dll-библиотеки приходится предпринимать дополнительные усилия [12]. Выгрузка (точнее освобождение ресурсов) dll-библиотеки осуществляется после того, как последний процесс, использовавший библиотеку выполнит функцию FreeLibrary. О функциях dll-библиотеки которые предназначены для вызова из вне говорят, что они ***экспортируются*** библиотекой. Когда рассматривают эти же функции со стороны клиента dll-библиотеки, то говорят об ***импорте функций*** из dll-библиотеки. Для импорта функций dll-библиотеки применяется функция GetProcAddress (рисунок 6.13.4). Следует отметить, что импортировать можно не только функции, но некоторые переменные.

Импорт переменных рассмотрен в [4], а дальнейшее изложение касается только импорта функций.

```
// -- загрузить dll-библиотеку
// Назначение: если dll-библиотека еще не находится в памяти
//             компьютера,  то осуществляется загрузка
//             библиотеки, настройка адресов и проецирование
//             dll-библиотеки в адресное пространство
//             процесса; если же  dll-библиотека к моменту
//             вызова функции уже загружена, то происходит
//             только проецирование; в любом случае управление
//             получит функция DllMain.

 HMODULE LoadLibrary(
             LPCTSTR  fname,    //[in]   имя файла dll-библиотеки
                    );

// Код возврата: в случае  успешного  завершения  функция
//     возвращает дескриптор загруженного модуля, иначе NULL
// Примечание: при поиске файла используется следующая
//      последовательность:
//      1) каталог, из которого запущено приложение;
//      2) текущий каталог;
//      3) системный каталог Windows;
//      4) каталоги, указанные в переменной окружения PATH.
```

Рисунок 6.13.3. Функция LoadLibrary

```
// -- отключить dll-библиотеку от процесса
// Назначение: функция предназначена для освобождения ресурсов
//             процесса, занимаемых dll-библиотекой; если
//             нет больше процессов, которые используют
//             библиотеку, то осуществляется ее выгрузка
 BOOL FreeLibrary(
            HMODULE hDll,  //[in]   дескриптор dll-библиотеки
                  );

// Код возврата: в случае  успешного  завершения  функция
//     возвращает ненулевое значение, иначе NULL
```

Рисунок 6.13.4. Функция FreeLibrary

Экспортируемая dll-библиотекой функция должна быть специальным образом оформлены, а именно иметь прототип изображенный на рисунке 6.13.6. Модификатор extern “C” используется для указания компилятору на то, что эта функция имеет имя в стиле языка C. Квалификатор __declspec(dllexport) применяется для обозначения экспортируемых dll-библиотекой функций.

```
// -- импортировать функцию
// Назначение: функция предназначена извлечения (импорта)
//             адреса функции dll-библиотеки

 FARPROC  GetProcAddress(
            HMODULE hDll,  //[in]   дескриптор dll-библиотеки
            LPCTSTR name   //[in]   имя импортируемой функции
                  );
// Код возврата: в случае  успешного  завершения  функция
//     возвращает адрес функции, иначе NULL
```

Рисунок 6.13.5. Функция GetProcAddress

```
// -- прототип экспортируемой dll-библиотекой функции

extern "C" __declspec(dllexport) Funcname (.....);
```

Рисунок 6.13.6. Прототип экспортируемой функции

```
#include "stdafx.h"
#include "Windows.h"
#include "DefineTableService.h" // макро для TableService
#include "PrototypeService.h"  // прототипы  обслуживающих потоков

BEGIN_TABLESERVICE              // таблица
  ENTRYSERVICE("Echo", EchoServer),
  ENTRYSERVICE("Time", TimeServer),
  ENTRYSERVICE("0001", ServiceServer01)
END_TABLESERVICE;

extern "C" __declspec(dllexport) HANDLE  SSS (char* id, LPVOID prm)
{
 HANDLE rc = NULL;
 int  i = 0;
 while(i < SIZETS && strcmp(TABLESERVICE_ID(i), id) != 0)i++;
 if (i < SIZETS)
      rc =  CreateThread(NULL,NULL,
                         TABLESERVICE_FN(i), prm ,NULL,NULL);
 return rc;
};

BOOL APIENTRY DllMain( HANDLE hinst, DWORD  rcall, LPVOID wres)
{
 return TRUE;
}
```

Рисунок 6.13.7. Пример построения TableService и функции SSS

На рисунке 16.13.7 приводится текст программы простейшего dll-модуля. Главная функция DllMain имеет только одно назначение – она

возвращает значение TRUE. За время использования сервером dll-библиотеки функция DllMain вызываться несколько раз: при загрузке и освобождении библиотеки, при запуске и остановке потоковой функции. Таблица TableService создается с помощью достаточно простых макроопределений хранящихся в заголовочном файле DefineTableService.h (рисунок 6.13.8).

```
struct __TableEntry {
                     char __Id[9];
                     DWORD  ((WINAPI* __Fn)) (LPVOID);
                    };
#define BEGIN_TABLESERVICE   __TableEntry  __TableService[] = {
#define ENTRYSERVICE(s,t)  {s,t}
#define END_TABLESERVICE    };
#define TABLESERVICE_ID(i)   __TableService[i].__Id
#define TABLESERVICE_FN(i)   __TableService[i].__Fn
#define SIZETS sizeof(__TableService)/sizeof(__TableEntry)
```

Рисунок 6.13.8. Пример макроопределений для построения TableService

Заголовочный файл PrototypeService.h должен содержать все прототипы потоковых функций, которые определяются в таблице TableService.

Следует обратить внимание, что экспортируемой является только одна функция SSS, которая запускает обслуживающий поток по заданному коду команды. Пример импорта функции SSS программой клиента dll-библиотеки приводится в фрагменте программы на рисунке 6.13.9.

```
//.............................................................
HANDLE (*ts)(char*, LPVOID);
HMODULE st  = LoadLibrary("ServiceTable");
//.............................................................

ts = (HANDLE (*)(char*, LPVOID))GetProcAddress(st, "SSS");

//.............................................................

contacts.begin()->hthread = ts("Echo",(LPVOID)&(*contacts.begin()));

//.............................................................
//.............................................................

FreeLibrary(st);
//.............................................................
```

Рисунок 6.13.9. Пример импорта функции SSS

6.14. Принципы разработки системы безопасности сервера

Система безопасности сервера является достаточно объемным понятием, которое может включать организационные мероприятия, специализированное аппаратное обеспечение, специальное программное обеспечение и т.д. и т.п. Здесь будет говориться только о вопросах предотвращения несанкционированного доступа к ресурсам (услугам) сервера. Причем в той части, которая касается программной реализации сервера.

В зависимости от функциональной нагрузки сервера возможны различные подходы к системе безопасности. Нет смысла создавать систему безопасности для сервера, если несанкционированный доступ к его ресурсам в принципе не может принести ущерб. С другой стороны, для специализированного сервера, например, управляющего технологическим процессом, может потребоваться мощная система защиты.

В последних версиях операционной системы Windows, любой исполняемый процесс всегда выполняется от имени одного из пользователей операционной системы, имеющего учетную запись в базе данных менеджера учетных записей (SAM, Security Account Manager). Поэтому далее для простоты мы не будем различать понятия: клиент, пользователь (в смысле ученой записи) и процесс, запущенный от имени этого пользователя.

Клиенты сервера могут быть разбиты на две группы: группа администраторов и группа пользователей сервера.

Группа администраторов, включает в себя привилегированных клиентов сервера, которые могут выполнять команды управления сервером через консоль и (или) с помощью специального обслуживающего потока (в зависимости от реализации сервера). Как правило, администраторам доступны все ресурсы сервера. В общем случае, внутри группы администраторов возможна иерархия, которая зависит от сложности системы управления сервером.

Группа пользователей, включает в себя потребителей ресурсов сервера. В общем случае, внутри группы пользователей, тоже возможна иерархия, разграничивающая доступ пользователей к ресурсам сервера.

Система безопасности не может быть стационарной: в любой момент могут появиться новые пользователи, смениться администраторы, измениться перечень ресурсов и т.д. А это значит, что система безопасности должна быть настраиваемой.

Прежде, чем заниматься разработкой системы безопасности сервера, следует ответить на вопрос: будет ли это собственная система безопасности или же она будет опираться на систему безопасности операционной системы. Например, Microsoft SQL Server использует, как внутренний способ аутентификации пользователя, так аутентификацию Windows. Следует, однако, иметь в виду, что разработка собственной системы защиты от несанкционированного доступа не является простым делом. Затраты на разработку этой системы, могут оказаться сравнимыми с разработкой самого сервера.

Продуктивнее, с точки зрения автора, использовать систему безопасности операционной системы Windows. Действительно, в рамках системы безопасности Window уже поддерживается понятие пользователя и группы пользователей, есть готовые средства позволяющие поддерживать списки групп и пользователей, можно создавать охраняемые объекты, управлять доступом субъектов к объектам, контролировать привилегии т.д. и т.п. Кроме того, есть API позволяющий использовать все это.

В простейшем случае можно в рамках операционной системы Window создать две группы: группа администраторов и группа пользователей сервера. Заполнить эти группы именами пользователей, которым разрешается доступ к соответствующим ресурсам сервера. Пусть наименование этих групп являются параметрами конфигурации сервера. Будем также предполагать, что при подключении клиента к серверу, помимо наименования необходимого сервиса в строке connect содержится имя и пароль подключаемого клиента. Очевидно, что достаточно просто можно проверить, на принадлежность подключаемого клиента к одной из групп и сравнить установленные пароли.

API системы безопасности операционной системы Windows выходит за рамки данного пособия и достаточно подробно описан в [4].

6.15. Итоги главы

1. По принципу работы различают два типа серверов: итеративные серверы и параллельные серверы.
2. Итеративные серверы, как правило, используют для связи с клиентами протокол UDP и используются в тех случаях, когда предполагается, что запросы клиентов являются редкими, а исполнении их не требует много времени. В каждый момент времени итеративным сервером всегда обслуживается только один клиент.
3. В параллельных серверах с каждым клиентом устанавливается TCP-соединение. Одновременно к одному параллельному серверу может быть подключено несколько клиентов.
4. Для организации параллельной работы с несколькими соединениями, применяются механизмы потоков и (или) механизмы процессов. Механизм процессов является более затратным , но и более надежным, т.к. предполагает полное разделение параллельно работающих компонент сервера. С точки зрения программирования проще является, конечно, применение потоков, т.к. в этом случае потоки разделяют общую память процесса, что существенно упрощает разработку.
5. Наличие ресурсов, допускающих только последовательное использование, делает необходимым синхронизировать потоки и (или) процессы. В арсенале Windows имеется много различных механизмов синхронизации: критические секции, мьютексы,

семафоры, события, асинхронные процедуры, ожидающие таймеры и.д.

6. В тех случаях, когда требуется совместная работа нескольких потоков с одной общей переменной, может быть применен специальный механизм упрощенной синхронизации – атомарные операции.
7. Для подключения клиентов к параллельному серверу применяется не блокирующий режим работы сокета, позволяющий сделать этот процесс управляемым.
8. Одним из способов распределения ресурса процессорного времени между параллельными потоками (или процессами) сервера является назначение потокам (или процессам) приоритетов.
9. Параллельный сервер, обрабатывающий разнотипные запросы, должен распределять эти запросы с помощью специальной таблицы, связывающей код запроса и функцию обрабатывающего потока. Для того, чтобы сделать процедуры управления сервером независимыми от функций обслуживания, последние вместе с таблице кодов часто размещают в отдельной динамически подключаемой библиотеке.
10. При разработке системы безопасности сервера, целесообразно использовать систему безопасности операционной системы. Это значительно снизит затраты на ее разработку.