Лабораторная работа № 3

Методы синхронизация потоков

Цель работы: получение практических навыков по использованию Win32 API для синхронизации потоков.

Синхронизация потоков

Выполняющимся потокам часто необходимо каким-то образом взаимодействовать. Например, если несколько потоков пытаются получить доступ к некоторым глобальным данным, то каждому потоку нужно предохранять данные от изменения другим потоком. Иногда одному потоку нужно получить информацию о том, когда другой поток завершит выполнение задачи. Такое взаимодействие возможно между потоками как одного, так и разных процессов.

Синхронизация потоков (*thread synchronization*) - это обобщенный термин, относящийся к процессу взаимодействия и взаимосвязи потоков. Синхронизация потоков может потребовать привлечения в качестве посредника самой операционной системы. Потоки разных процессов не могут взаимодействовать друг с другом без ее участия.

В Win32 существует несколько методов синхронизации потоков. При всей их схожести, часто оказывается, что в конкретной ситуаций один метод более предпочтителен, чем другой.

Критические секции

Один из методов синхронизации потоков состоит в использовании критических секций (critical sections). Это единственный метод синхронизации потоков, который не требует привлечения ядра Windows, то есть критическая секция не является объектом ядра. Как следствие этого этот метод может использоваться только для синхронизации потоков одного процесса.

Критическая секция — это некоторый участок кода, который в каждый момент времени может выполняться только одним из потоков. Если код, используемый для инициализации массива, поместить в критическую секцию, то другие потоки не смогут войти в этот участок кода до тех пор, пока первый поток не завершит его выполнение.

До использования критической секции необходимо инициализировать ее с помощью процедуры Win32 API InitializeCriticalSection(), которая определяется (в Delphi) следующим образом:

```
procedure InitializeCriticalSection(var IpCriticalSection: TRTLCriticalSection);  stdcall;
```

Параметр IpCriticalSection представляет собой запись типа TRTLCriticalSection, которая передается по ссылке. Точное определение записи TRTLCriticalSection не имеет большого значения, поскольку вам вряд ли понадобится когда-либо заглядывать в ее содержимое. От

вас требуется лишь передать неинициализированную запись в параметр IpCtitical Section, и эта запись будет тут же заполнена процедурой.

После заполнения записи в программе можно создать критическую секцию, поместив некоторый участок ее текста между вызовами функций EnterCriticalSection() и LeaveCriticalSection(). Эти процедуры определяются следующим образом:

```
procedure EnterCriticalSection(var IpCriticalSection: TRTLCriticalSection); stdcall;
procedure LeaveCriticalSection(var IpCriticalSection: TRTLCriticalSection);  stdcall;
```

Параметр IpCriticalSection, который передается этим процедурам, является не чем иным, как записью, созданной процедурой InitializeCriticalSection().

Функция *EnterCriticalSection* проверяет, не выполняет ли уже какой-нибудь другой поток критическую секцию своей программы, связанную с данным объектом критической секции. Если нет, поток получает разрешение на выполнение своего критического кода, точнее, ему не запрещают это делать. Если да, то поток, обратившийся с запросом, переводится в состояние ожидания, а о запросе делается запись. Так как нужно создавать записи, объект «критическая секция» представляет собой структуру данных.

Когда функция *LeaveCriticalSection* вызывается потоком, который владеет в текущий момент разрешением на выполнение своей критической секции кода, связанной с данным объектом «критическая секция», система может проверить, нет ли в очереди другого потока, ожидающего освобождения этого объекта. Затем система может вынести ждущий поток из состояния ожидания, и он продолжит свою работу (в выделенные ему кванты времени).

По окончании работы с записью TRTLCriticalSection необходимо освободить ее, вызвав процедуру DeleteCriticalSection(), которая определяется следующим образом:

```
procedure DeleteCriticalSection(var IpCriticalSection: TRTLCriticalSection);  stdcall;
```

Для того, чтобы обойти блокировку потока при обращении к занятой секции, есть функция TryEnterCriticalSection(), которая позволяет проверить критическую секцию на занятость.

Синхронизация с использованием объектов ядра

Многие объекты ядра, включая процесс, поток, файл, мьютекс, семафор, уведомление об изменении файла и событие, могут находиться в одном из двух состояний - «свободно» (*signaled*) и «занято» (*nonsignaled*). Вероятно, проще представлять себе эти объекты подключенными к лампочке. Если свет горит, объект свободен, в обратном случае объект занят.

Например, в момент создания процесса его объект ядра находится в состоянии «занято». Когда процесс завершается, объект переходит в состояние «свободно». Аналогично выполняющиеся потоки (то есть их объекты) пребывают в состоянии «занято», но переходят в состояние «свободно», когда завершают работу. На самом деле некоторые объекты, такие как мьютекс, семафор, событие, уведомление об изменении файла, таймер ожидания, существуют исключительно для того, чтобы вырабатывать сигналы «свободно» и «занято».

Смысл всей этой «сигнализации» в том, чтобы поток мог приостанавливать свою работу до того момента, когда заданный объект перейдет в состояние «свободно». Например, поток одного процесса может временно прекратить работу до завершения другого, просто подождав, когда объект ядра этого другого процесса перейдет в состояние «свободно».

Посредством вызова функций *WaitForSingleObject* и *WaitForMultipleObjects* поток приостанавливает свое выполнение до того момента, когда заданный объект (или объекты) перейдет в состояние «свободно». Рассмотрим функции *WaitForSingleObject*, декларация которой выглядит так:

```
DWORD WaitForSingleObject(
    HANDLE hHandle,     // Дескриптор объекта ожидания
    DWORD dwMilliseconds    // Время ожидания в миллисекундах
);
```

Параметр *hHandle* является дескриптором объекта, уведомление о свободном состоянии которого требуется получить, а *dwMilliseconds* - это время, которое вызывающий поток готов ждать. Если *dwMilliseconds* равно нулю, функция немедленно вернет текущий статус заданного объекта. Таким образом, можно протестировать состояние объекта. Параметру можно также присваивать значение символьной константы *INFINITE (= -1),* в этом случае вызывающий поток будет ждать неограниченное время.

Функция *WaitForSingleObject* переводит вызывающий поток в состояние ожидания до того момента, когда она передаст ему свое возвращаемое значение. Ниже перечислены возможные возвращаемые значения:

- *WAIT_OBJECT_0* - объект находится в состоянии «свободно»;
- *WAIT_TIMEOUT* - интервал ожидания, заданный *dwMilliseconds,* истек, а нужный объект по прежнему находится в состоянии «занято»;
- *WAIT_ABANDONED* относится только к мьютексу и означает, что объект не был освобожден потоком, который владел им до своего завершения;
- *WAIT_FAILED* - при выполнении функции произошла ошибка.

Объекты «мьютекс»

Мьютекс (MUTual Exclusions— взаимоисключения) - это объект ядра, который можно использовать для синхронизации потоков из разных процессов. Он может принадлежать или не принадлежать некоторому потоку. Если мьютекс принадлежит потоку, то он находится в состоянии «занято». Если данный объект не относится ни к одному потоку, то он находится в состоянии «свободно». Другими словами, принадлежать для него означает быть в состоянии «занято».

Если мьютекс не принадлежит ни одному потоку, первый поток, который вызовет функцию WaitForSingleObject, завладевает данным объектом, и тот переходит в состояние «занято». В определенном смысле мьютекс похож на выключатель, которым может пользоваться любой поток по принципу «первым пришел - первым обслужили» (first-come-first,-served).

Дело в том, что при попытке с помощью вызова функции WaitForSingleObject завладеть мьютексом, который уже находится в состоянии «занято», поток переводится в состояние ожидания до того момента, когда данный объект освободится, то есть когда «владелец» мьютекса его освободит (переведет в состояние «свободно»).

По принципу своего действия мьютексы очень похожи на критические секции, за исключением двух моментов. Во-первых, мьютексы можно использовать для синхронизации потоков, переступая через границы процессов. Во-вторых, мьютексу можно присвоить имя и путем ссылки на это имя создать дополнительные дескрипторы существующих объектов мьютексов.

Мьютексы создаются с помощью вызова функции CreateMutex:

```
HANDLE CreateMutexi
    LPSECURITY_ATTRIBUTES   IpMutexAttributes,
                                        // Указатель на атрибуты защиты.
    BOOL bInitialOwner,                 // флаг первоначального владельца.
    LPCTSTR IpName                      // Указатель на имя мьютекса.
);
```

Параметр IpMutexAttributes — это указатель на запись типа TSecurityftttributes. Обычно в качестве данного параметра передастся значение nil, и в этом случае используются атрибуты защиты, действующие по умолчанию.

Параметр blnitialOwner определяет, следует ли считать поток, создающий мьютекс, его владельцем. Если этот параметр равен False, значит, мьютекс не имеет владельца.

Параметр IpName представляет имя мыотскса. Если вы не собираетесь присваивать мьютексу имя, установите этот параметр равным nil. Если же значение этого параметра отлично от nil, функция выполнит в системе поиск мьютекса с таким же именем. При

успешном завершении поиска функция вернет дескриптор найденного мьютекса, в противном случае возвращается дескриптор нового мьютекса. При наличии имени этот объект может совместно использоваться несколькими процессами. Если каким-то процессом создается мьютекс с именем, то поток другого процесса может вызывать функции CreateMutex или OpenMutex с тем же самым именем. В любом случае система просто передаст вызывающему потоку дескриптор исходного мьютекса. Другой способ совместно использовать мьютекс - вызвать функцию DuplicateHandle.

Чтобы работать с несколькими процессами, данный объект должен быть совместно используемым. Причина проста: чтобы завладеть мьютексом или освободить его, потоку потребуется его дескриптор. Поток освобождает этот объект с помощью вызова функции ReleaseMutex:,

```
BOOL ReleaseMutex(
    HANDLE hMutex                    // Дескриптор мьютекса.
);
```

А что случится, если владеющий мьютексом поток завершится, предварительно не освободив его? В действительности система сама освобождает такой мьютекс. Поток, который вызывает функцию WaitForSingleObject для этого объекта, получит возвращенное значение WAIT_ABANDONED, которое указывает на возникшие проблемы с только что завершимся потоком-владельцем. В этом случае ждущий поток должен определить, стоит продолжать выполнение в обычном режиме или нет.

По завершении использования мьютекса необходимо закрыть его с помощью функции Win32 API CloseHandle().

События

События используются в качестве сигналов о завершении какой-либо операции. Однако в отличие от мьютексов, они не принадлежат ни одному потоку. Например, поток А создает событие с помощью функции *CreateEvent* и устанавливает объект в состояние «занято». Поток В получает дескриптор этого объекта, вызвав функцию *OpenEvent,* затем вызывает функцию *WaitForSingleObject*, чтобы приостановить работу до того момента, когда поток А завершит конкретную задачу и освободит указанный объект. Когда это произойдет, система выведет из состояния ожидания поток В, который теперь владеет информацией, что поток А завершил выполнение своей задачи.

Объявление функции *CreateEvent* записывается таким образом:

```
HANDLE CreateEvent(
    LPSECURITY_ATTRIBUTES  lpEventAttributes,
                                        // Указатель на атрибуты защиты.
    BOOL bManualReset,                  // Флаг интерактивного события.
    BOOL bInitialState,                 // Флаг первоначального состояния.
    LPCTSTR lpName                      // Указатель на имя события.
);
```

Эта функция возвращает дескриптор создаваемого объекта «событие». Первый параметр определяет, наследуется ли дескриптор порожденными процессами. Если *lpEventAttributes* имеет значение NULL, дескриптор наследоваться не может.

Если параметр *bManualReset* имеет значение *TRUE*, то при освобождении объект остается в этом состоянии (в отличие от объекта «мъютекс»). Это значит, что все потоки, ожидающие перехода данного объекта в состояние «свободно», будут выведены системой из состояния ожидания. Такой объект называется событием с ручным сбросом (*manual-reset event),* поскольку «разбуженный» (выведенный из состояния ожидания) поток может самостоятельно сбросить состояние объекта «событие» в «занято». Если параметр *bManualReset* имеет значение *FALSE,* то система автоматически сбрасывает состояние рассматриваемого объема в «занято» после «пробуждения» первого потока, ожидающего освобождения данного объекта. Только один поток выводится из состояния ожидания, как и в случае с мьютексами. Такое событие называют событием с автоматическим сбросом (*auto-reset event).*

Параметр *bInitialState* определяет первоначальное состояние (если *TRUE*, то «свободно», если *FALSE*, то «занято») данного события. Параметру *lpName* может быть присвоено имя события. Имя предоставляет способ совместного использования, например посредством функции *OpenEvent.*

В качестве дополнительного варианта, если вы не хотите иметь дела с вопросами защиты, можно установить *lpEventAttributes* в *NULL* (0&). В таком случае декларация примет следующий вид:

```
Declare Function CreateEvent Lib "kernel32" Alias "CreateEventA" ( _
    ByVal lpEventAttributes As Long, _
    By Val bManualReset As Long, _
    By Val blmtialState As Long, _
    By Val lpName As String _
) As Long
```

Так же, как и другие дескрипторы, дескриптор события должен быть закрыт с использованием функции *CloseHandle.*

Объявление функции *OpenEvent* выглядит так:

HANDLE OpenEvent(
DWORD dwDesiredAccess, // Флаг доступа.
BOOL blnheritHandle, // Флаг наследования.
LPCTSTR IpName // Указатель на имя события.
);

где параметр *dwDesiredAccess* может принимать одно из трех значений:

- *EVENT_ALL_ACCESS* предоставляет полный доступ к событию;
- *EVENT_MODIFY_STATE* разрешает использование дескриптора события в функциях *SetEvent* и *ResetEvent*, так что вызывающий процесс может изменить состояние данного события (но ничего больше). Это важно для событий со сбросом вручную;
- *SYNCHRONIZE* разрешает использование дескриптора события в любых функциях ожидания (таких как *WaitForSingleObject*), ждущих освобождения данного объекта.

Каждая из этих функций принимает дескриптор события в качестве аргумента. Функция *SetEvent* устанавливает состояние данного события в «свободно», а Reset Event «сбрасывает» событие, то есть присваивает событию статус «занято», функция *PulseEvent* вызывает *SetEvent* для освобождения ожидающих потоков, а затем вызывает *ResetEvent* для перевода данного события в состояние «занято».

Семафоры

Существует еще один метод синхронизации потоков, в котором используются семафорные объекты API. В семафорах применен принцип действия мьютексов, но с добавлением одной существенной детали. В них заложена возможность подсчета ресурсов, что позволяет заранее определенному числу потоков одновременно войти в синхронизуемый участок кода. Для создания семафора используется функция CreateSemaphore(), которая объявляется следующим образом:

function CreateSemaphoreflpSemaphoreAttributes: PSecurityAttributes; llnitialCoimt, iMaximumCount: Longint; IpName: PChar): THandle; stdcall;

Как и в случае функции CreateMutex(), первым параметром, передаваемым функции CreateSemaphore(), является указатель на запись TSecurityAttributes, причем значение Nil

соответствует согласию на использование стандартных атрибутов защиты.

Параметр llnitialCount представляет собой начальное значение счетчика семафорного объекта. Это число может находиться в диапазоне от 0 до значения IMaximumCount. Семафор доступен, если значение этого параметра больше нуля. Когда поток вызывает функцию WaitForSingleObject() или любую другую, ей подобную, значение счетчика семафора уменьшается на единицу. И наоборот, при вызове потоком функции ReleaseSemaphore() значение счетчика семафора увеличивается на единицу.

С помощью параметра IMaximumCount задается максимальное значение счетчика семафорного объекта. Если семафор используется для подсчета некоторых ресурсов, это число должно представлять общее количество доступных ресурсов.

Параметр IpName содержит имя семафора. Поведение этого параметра аналогично поведению одноименного параметра функции CreateMutex().

Функция ReleaseSemaphore () используемая для увеличения значения счетчика семафора имеет больше параметров, чем ее "коллега" ReleaseMutex(). Объявление функции ReleaseSemaphore() выглядит следующим образом:

function ReleaseSemaphore(hSemaphore: THandle; IReleaseCount: Longint; IpPreviousCount: Pointer): BOOL; stdcall;

С помощью параметра IReleaseCount можно задать число, на которое будет уменьшено значение счетчика семафора. При этом старое значение счетчика будет сохранено в переменной типа Longint, на которую указывает параметр IpPreviousCount, если его значение не равно Nil, Скрытый смысл этого средства состоит в том, что семафор никогда не принадлежит ни одному

отдельному потоку. Предположим, что максимальное значение счетчика семафора было равно 10, и десять потоков вызвали функцию WaitForSingleObject(). В результате счетчик потоков сбрасывается до нуля и тем самым семафор переводится в недоступное состояние. После этого достаточно одному из потоков вызвать функцию ReleaseSemaphore() и в качестве параметра lReleaseCount передать число 10, как семафор не просто будет снова пропускать потоки, т.е. станет доступным, но и увеличит значение своего счетчика до прежнего числа — до 10. Это мощное средство может привести к возникновению в вашем приложении трудно отслеживаемых ошибок, поэтому следует использовать его с большой осторожностью.

Семафоры могут быть полезны при совместном использовании ограниченных ресурсов. Предположим, имеется три приложения, каждое из которых должно выполнить вывод на печать, а у компьютера только два параллельных порта. Установив семафор с начальным значением счетчика ресурсов, равным двум, можно заставить приложения запрашивать сервис печати только тогда, когда есть свободный параллельный порт.

Для освобождения дескриптора семафора, выделенного ему с помощью функции CreateSemaphore(), не забудьте вызвать функцию CloseHandle().

Ждущие таймеры

Ждущий таймер (waitable timer) представляет собой новый тип объектов синхронизации, поддерживаемый в Windows NT версии 4.0 и выше. Это полноценный объект синхронизации, который может использоваться для организации задержки в одном или нескольких приложениях.

Ждущий таймер работает в трех режимах. В режиме «ручного сброса» таймер переходит в установленное состояние при истечении заданной задержки и остается установленным до тех пор, пока функция SetWaitableTimer не задаст новую задержку. В режиме «автоматического сброса» таймер переходит в установленное состояние при истечении заданной задержки и остается установленным до первого успешного вызова функции ожидания. В этом режиме он напоминает объект Event в режиме автоматического сброса, поскольку каждый раз при истечении времени задержки разрешается выполнение лишь одной нити. Наконец, ждущий таймер может выполнять функции интервального таймера, который перезапускается с заданной задержкой после каждого срабатывания объекта.

Главная особенность, отличающая ждущие таймеры от системных, — то, что ждущие таймеры могут совместно использоваться несколькими приложениями. Например, вы можете приостановить несколько приложений в фоновом режиме так, чтобы они «просыпались» каждые несколько часов для выполнения некоторой операции.

Процессы получают дескрипторы ждущих таймеров так же, как они получают дескрипторы мьютексов: дублированием, наследованием или открытием по имени.

В следующей таблице перечислены функции, предназначенные для работы со ждущими таймерами.

Функция	Описание
CancelWaitableTimer	Останавливает работу ждущего таймера. Таймер остается в текущем состоянии
CreateWaitableTimer	Создает объект ждущего таймера. Если таймер с заданным именем уже существует, он открывается
OpenWaitableTirner	Открывает существующий ждущий таймер
SetWaitableTimer	Запускает ждущий таймер с заданной

продолжительностью и интервалом срабатывания

СОДЕРЖАНИЕ ОТЧЕТА

1. Наименование лабораторной работы, ее цель.
2. Исследование на конкретном примере следующих методов синхронизации потоков:
    1) критические секции
    2) мьтексы
    3) события

   Задачу для синхронизации выбрать на свое усмотрение.
3. Примеры разработанных приложений (описание программ, результаты и тексты программ).

Примечание:

1. Задачи для каждого метода синхронизации должны быть различными.
2. Задачи должны наглядно демонстрировать выбранный метод синхронизации и учитывать его особенности.
3. Студент, сдающий работу, должен АРГУМЕНТИРОВАННО обосновать задачу, выбранную для синхронизации и метод синхронизации.