Лабораторная работа № 4

Объекты и функции синхронизации Win32

ОС Windows предоставляет широкий набор средств, предназначенных для синхронизации потоков и процессов. Рассмотрим их в порядке их усложнения и возрастания мощности. Наиболее простым средством являются блокированные вызовы или функции взаимоблокировки. Приостановить поток до перехода какого-либо объекта ядра в сигнальное состояние позволяют функции ожидания. Затем рассматриваются четыре основных объекта синхронизации: критическая секция, мьютекс, семафор и событие. Иногда для синхронизации применяются два других типа объектов – таймер ожидания и порты завершения ввода/вывода.

Блокированные вызовы

Рассмотрим два потока, в каждом из которых увеличивается значение одной и той же переменной.

```
long i = 0; //определение глобальной переменной процесса
DWORD WINAPI ThreadFunc1(PVOID pvParam)
{
       i++;
       return(0);
}
DWORD WINAPI ThreadFunc2(PVOID pvParam)
{
       i++;
       return(0);
}
```

Оба потока увеличивают значение переменной i на 1, таким образом можно предположить, что при одновременном их запуске значение общей переменной станет равно 2. Однако это не всегда так. Рассмотрим примерную последовательность машинных команд, которую сгенерировал бы компилятор для рассматриваемых потоковых функций.

```
MOV EAX, i  ; значение из i помещается в регистр
INC EAX     ; значение регистра увеличивается на 1
MOV i, EAX  ; значение из регистра помещается обратно в i
```

При последовательном выполнении потоков, то есть когда второй поток начнет выполнение после первого, значение переменной i действительно получит значение 2.

```
MOV EAX, i  ; поток 1
INC EAX     ; поток 1
MOV i, EAX  ; поток 1
MOV EAX, i  ; поток 2
INC EAX     ; поток 2
MOV i, EAX  ; поток 2
```

Но в Windows переключение потоков может произойти в любой момент времени, и последовательность выполнения команд может измениться.

```
MOV EAX, i  ; поток 1
IHC EAX     ; поток 1
MOV EAX, i  ; поток 2
INC EAX     ; поток 2
MOV i, EAX  ; поток 2
MOV i, EAX  ; поток 1
```

После выполнение такой последовательности команд значение переменной станет равно 1. Приведенный пример наглядно демонстрирует сложность разработки многопоточных приложений и необходимость включения в программный код средств синхронизации доступа потоков к разделяемым данным.

Один из самых простых и действенных способов, гарантирующих изменение значение переменной на уровне атомарного доступа (без прерывания другими потоками) – это использование функций взаимоблокировки (Interlocked-функций).

Interlocked-функции – это простейшая форма механизмов синхронизации операций над целыми числами и выполнения сравнений в многопроцессорной среде. Данный набор функций опирается на аппаратную поддержку процессора при доступе к памяти. Например, функция InterlockedDecrement использует префикс команды lock для блокировки шины на время операции вычитания, чтобы другой процессор, модифицирующий тот же участок памяти, не мог выполнить свою операцию в момент между чтением исходных данных и записью их нового значения.

`InterlockedDecrement`	Вычитает единицу из блокированной переменной
`InterlockedIncrement`	Добавляет единицу к блокированной переменной
`InterlockedExchange`	Меняет местами значение блокированной переменной и значение другой переменной
`InterlockedExchangeAdd`	Добавляет значение к блокированной переменной
`InterlockedCompareExchange`	Сравнивает значение блокированной переменной с некоторым значением и в случае, если значения равны, меняет местами значение блокированной переменной и значение некоторой другой переменной

Приведенный выше пример, реализованный с помощью функции InterlockedExchangeAdd, всегда будет давать верный результат.

```
long i = 0;

DWORD WINAPI ThreadFund(PVOID pvParam)
{
	InterlockedExchangeAdd(&i, 1);
	return(0);
}

DWORD WINAPI ThreadFunc2(PVOID pvParam)
{
	InterlockedExchangeAdd(&i, 1);
	return(0);
}
```

Важный аспект, связанный с Interlocked-функциями, состоит в том, что они выполняются чрезвычайно быстро. Вызов такой функции обычно требует не более 50 тактов процессора, и при этом не происходит перехода из пользовательского режима в режим ядра (а он отнимает не менее 1000 тактов).

Критическая секция

Критическая секция (critical section) - это участок кода, требующий монопольного доступа к каким-то общим данным (ресурсам); параллельное выполнение этого участка кода несколькими потоками может привести к непредсказуемым или неверным результатам.

Понятие критической секции подразумевает, что единовременно только **один поток** получал доступ к определенному ресурсу (участку кода, манипулирующему с данным ресурсом). Система может в любой момент вытеснить поток, находящийся в критической,

но ни один из потоков, которым нужен занятый ресурс, не получит процессорное время до тех пор, пока владеющим им поток **не выйдет за границы критической секции**.

Название критическая секция используется и для объекта синхронизации – структуры (типа данных) Windows, используемой для организации *критической секции* в изложенном выше смысле.

Для реализации взаимного исключения с помощью критической секции необходимо глобально (обеспечить доступность из всех потоков процесса, конкурирующих за ресурс) объявить переменную типа CRITICAL_SECTION. Структура CRITICAL_SECTION представляет для программиста «черный ящик», то есть не происходит прямого обращения к полям данной структуры. Работа с CRITICAL_SECTION осуществляется исключительно через специальные функции, которым передается адрес соответствующего экземпляра данной структуры.

Перед началом использования (до обращения какого-либо потока к защищенному ресурсу) элементы структуры необходимо инициализировать с помощью вызова:

```
void InitializeCriticalSection( LPCRITICAL_SECTION lpCriticalSection);
```

Захват критической секции производится с помощью функций:

```
void EnterCriticalSection( LPCRITICAL_SECTION lpCriticalSection );

BOOL TryEnterCriticalSection( LPCRITICAL_SECTION lpCriticalSection );
```

При вызове EnterCriticalSection происходит анализ элементов структуры CRITICAL_SECTION. Если ресурс занят, в них содержатся сведения о том, какой поток пользуется ресурсом. EnterCriticalSection выполняет следующие действия.

Если ресурс свободен, EnterCriticalSection модифицирует элементы структуры, указывая, что вызывающий поток занимает ресурс, после чего немедленно возвращает управление, и поток продолжает свою работу (получив доступ к ресурсу).

Если значения элементов структуры свидетельствуют, что ресурс уже захвачен вызывающим потоком, EnterCriticalSection обновляет их, отмечая тем самым, сколько раз подряд этот поток захватил ресурс, и возвращает управление потоку. Таким образом, поток, владеющий критической секцией, может повторно войти в критическую секцию, что дает возможность организации рекурсивных функций.

Если значения элементов структуры указывают на то, что ресурс занят другим потоком, EnterCriticalSection переводит вызывающий поток в режим ожидания. Система запоминает, что данный поток хочет получить доступ к ресурсу, и - как только поток, занимавший критическую секцию, освобождает ее, ожидающий поток переходит в активное состояние и занимает критическую секцию. Интересным фактом является то, что потоки, ожидающие освобождения критической секции, на самом деле не блокируются «навечно». Данная функция реализована таким образом, что по истечении определенного времени, генерирует исключение (ожидание прекращается ошибкой). Длительность времени ожидания функцией EnterCriticalSection определяется значением системного реестра Windows. Длительность времени ожидания измеряется в секундах и по умолчанию равна 2 592 000 секунд (что составляет ровно 30 суток).

Использование функции TryEnterCriticalSection позволяет опросить критическую секцию, для проверки занята она другим потоком или нет. Возвращение данной функцией значения True свидетельствует о том, что вызывающий поток приобрел права владения критическим участком кода, тогда как значение False говорит о том, что данный критический участок кода уже принадлежит другому потоку. Функция TryEnterCriticalSection никогда не приостанавливает выполнение вызывающего потока, позволяя проверить доступность ресурса и в случае его занятости обработать такую ситуацию соответствующим образом.

В конце участка кода, использующего разделяемый ресурс, должен присутствовать вызов:

```
void LeaveCriticalSection( LPCRITICAL_SECTION lpCriticalSection );
```

Эта функция просматривает элементы структуры CRITICAL_SECTION и уменьшает счетчик числа захватов ресурса вызывающим потоком на 1. Если его значение больше 0, LeaveCriticalSection ничего не делает и просто возвращает управление. Если значение счетчика достигло 0, LeaveCriticalSection сначала выясняет, есть ли в системе другие потоки, ждущие данный ресурс в вызове EnterCriticalSection. Если есть хотя бы один такой поток, функция настраивает значения элементов структуры, чтобы они сигнализировали о занятости ресурса, и отдает его одному из ожидающих потоков при присутствии таковых. LeaveCriticalSection никогда не приостанавливает поток, а управление возвращает немедленно.

При завершении работы с критической секцией для нее необходимо вызвать функцию

```
void DeleteCriticalSection( LPCRITICAL_SECTION lpCriticalSection );
```

Данная функция сбрасывает все переменные-члены внутри структуры CRITICAL_SECTION. Естественно, нельзя удалять критическую секцию в тот момент, когда ею все еще пользуется какой-либо поток.

В силу своей простоты и преимуществ в отношении производительности критические секции являются предпочтительным механизмом синхронизации, если из возможностей достаточно для того, чтобы удовлетворить требования, предъявляемые решаемой задачей.

Однако критическая секция имеет один существенный недостаток: она не может быть использована для синхронизации потоков принадлежащих различным процессам.

Ожидающие функции

Для синхронизации выполнения программных потоков могут быть использованы различные объекты ядра Windows: процессы, потоки, задания, файлы, консольный ввод, уведомления об изменении файлов, события, ожидаемые таймеры, семафоры, мьютексы. Перечисленные объекты имеют различное назначение, но всех их объединяет свойство находиться в *свободном* (signaled) или *занятом* (notsignaled) состояниях. Например, объекты ядра «процесс» сразу после создания всегда находятся в занятом состоянии. В момент завершения процесса операционная система автоматически освобождает его объект ядра «процесс», и он навсегда остается в этом состоянии. Разработчики Windows для каждого из объектов определили свои правила перехода между свободным и занятым состоянием (см. таблицу). Для реализации синхронизации необходимо унифицированное средство, позволяющее «усыпить» потоки и в таком состоянии ожидать освобождения объекта синхронизации, не конкурируя за время процессора, т.е. реализуя эффективное ожидание. В Windows таким средством являются *ожидающие функции* (wait-функции).

При большом разнообразии объектов, пригодных для синхронизации, существует всего несколько функций ожидания, который способны работать с различными типами этих объектов. Это вызвано тем, что обращения к объекту ядра осуществляется с помощью дескриптора, по которому ожидающая функция определят тип объекта и правило его перехода из состояния в состояние.

Ожидающие функции приостанавливают выполнение потока до освобождения объекта (ов) ядра или до завершения времени ожидания. К ожидающим функциям относятся: WaitForMultipleObjects, WaitForMultipleObjectsEx, WaitForSingleObject, WaitForSingleObjectEx, MsgWaitForMultipleObjects, MsgWaitForMultipleObjectsEx, WaitForInputIdle, SleepEx, SignalObjectAndWait. Наиболее часто используются функции WaitForSingleObject и WaitForMultipleObjects.

Функция WaitForSingleObject осуществляет ожидание перехода в свободное состояние одного объекта

```
DWORD WaitForSingleObject(
          HANDLE hObject,
          DWORD dwMilliseconds);
```

Первый параметр, hObject, идентифицирует объект ядра, поддерживающий состояния «свободен-занят». Второй параметр, dwMilliseconds, указывает, сколько времени (в миллисекундах) поток готов ждать освобождения объекта. В качестве значения второго параметра может быть использована константа INFINITE (OxFFFFFFFF -1), указывающая, что вызывающий поток готов ждать этого события хоть бесконечно. Очевидно, передача INFINITE не всегда безопасна; если объект так и не перейдет в свободное состояние, вызывающий поток никогда не проснется. В параметре dwMilliseconds можно передать 0, и тогда WaitForSingleObject немедленно вернет управление.

Возвращаемое значение функции WaitForSingleObject указывает на причину завершения функции. Если функция возвращает WAIT_OBJECT_0, объект ожидания свободен, а если WAIT_TIMEOUT - заданное время ожидания (таймаут) истекло. При передаче неверного параметра (например, недопустимого описателя) WaitForSingleObject возвращает WAIT_FAILED.

Функция WaitForMultipleObjects аналогична WaitForSingleObject с тем исключением, что позволяет ждать освобождения сразу нескольких объектов или какого-то одного из списка объектов:

```
DWORD WaitForMultipleObjects(
          DWORD nCount,
          const HANDLE* lpHandles,
          BOOL bWaitAll,
          DWORD dwMilliseconds
);
```

Параметр nCount определяет количество объектов ядра, на которых будет происходить ожидание. Параметр lpHandles – это указатель на массив описателей объектов ядра.

Параметр bWaitAll определяет событие завершения ожидания. Если он равен TRUE, функция не позволит потоку возобновить свою работу, пока не освободятся все объекты.

Параметр dwMilliseconds идентичен одноименному параметру функции WaitForSingleObject, Если Вы указываете конкретное время ожидания, то по его истечении функция в любом случае возвращает управление. В этом параметре обычно передают INFINITE.

Возвращаемое значение функции WaitForMultipleObjects сообщает, почему возобновилось выполнение вызвавшего ее потока. Значения WAIT_FAILED и WAIT_TIMEOUT аналогичны предыдущей функции. Если параметре bWaitAll было передано значение TRUE в и все объекты перешли в свободное состояние, функция возвращает значение WAIT_OBJECT_0. Если же bWaitAll приравнен FALSE, она возвращает управление, как только освобождается любой из объектов. В этом случае возвращается значение от WAIT_OBJECT_0 до WAIT_OBJECT_0 + nCount– 1. Иначе говоря, если возвращаемое значение не равно WAIT_TIMEOUT или WAIT_FAILED, то если вычесть из него значение WAIT_OBJECT_0, то будет получен индекс в массиве описателей, переданном в качестве второго параметра. Индекс подскажет Вам, какой объект перешел в незанятое состояние.

Успешный вызов WaitForSingleObject или WaitForMultipleObjects меняет состояние некоторых объектов ядра. Под успешным вызовом понимается вызов, при котором объект освободился, и функция возвратила значение WAIT_OBJECT_0. Вызов считается неудачным, если возвращается WAIT_TIMEOUT или WAIT_FAILED, в последнем случае состояние объектов не меняется.

События

Объект событие используются для уведомления одного или нескольких ожидающих потоков о наступлении какого-либо события. Объекты события делятся на два типа: сбрасываемые вручную (manual event или manual-reset event) и автоматически сбрасываемые или единичные события (single event или auto-reset event). Тип события указывает программист при создании события с помощью вызова CreateEvent.

Сбрасываемые вручную события могут сигнализировать одновременно всем потокам, ожидающим наступления этого события, и переводятся в занятое состояние программно. Любой поток может установить (перевести в свободное состояние) это событие при помощи вызова функции SetEvent или сбросить (перевести в занятое состояние) при помощи вызова ResetEvent. Если событие установлено, то оно остается в этом состоянии сколь угодно долгое время, вне зависимости сколько потоков ожидают этого события.

Автоматически сбрасываемые события сбрасываются самостоятельно после освобождения одного из ожидающих потоков, тогда как другие ожидающие потоки продолжают ожидать перехода события в свободное состояние. Другими словами, если при помощи SetEvent устанавливается автоматически сбрасываемое событие (единичное), только один ожидающий поток будет оповещен об этом событии и, соответственно, сможет продолжить работу. После этого система автоматически сбросит событие. Если в момент установки не существует ни одного ожидающего потока, событие останется в свободном состоянии до тех пор, пока в системе не появится какой-либо ожидающий это событие поток.

Объекты-события обычно используют в том случае, когда какой-то поток выполняет инициализацию, а затем сигнализирует другому потоку, что тот может продолжить работу. Инициализирующий поток переводит объект «событие» в занятое состояние и приступает к своим операциям. Закончив, он сбрасывает событие в свободное состояние. Тогда другой поток, который ждал перехода события в свободное состояние, пробуждается и вновь становится планируемым.

Создается объект событие с помощью функции CreateEvent.

```
HANDLE CreateEvent(
          LPSECURITY_ATTRIBUTES lpEventAttributes,
          BOOL bManualReset,
          BOOL bInitialState,
          LPCTSTR lpName
);
```

Параметр bManualReset сообщает системе событие какого типа требуется создать: событие со сбросом вручную (TRUE) или с автосбросом (FALSE). Параметр bInitialState определяет начальное состояние события - свободное (TRUE) или занятое (FALSE).

При успешном выполнении функция вернет дескриптор созданного события. Если событие с таким именем уже создано, то будет возвращен дескриптор уже существующего события, а вызов GetLastError() вернет код ERROR_ALREADY_EXISTS. Функция вернет NULL, если объект события создать не удалось.

Потоки из других процессов могут получить доступ к уже созданному объекту «событие» следующими способами:

1) вызовом CreateEvent с тем же параметром lpName;

2) наследованием описателя;

3) применением функции DuplicateHandle;

4) вызовом функции OpenEvent с передачей в параметре lpName имени, совпадающего с указанным в аналогичном параметре функции CreateEvent.

Событие устанавливается в свободное состояние с помощью функции SetEvent, в качестве единственного параметра которой передается дескриптор этого события. В случае успеха функция вернет ненулевое значение.

```
BOOL SetEvent ( HANDLE hEvent );
```

Функция ResetEvent сбрасывает событие (меняет на занятое состояние). Значение параметра и возвращаемое значение аналогичны предыдущей функции.

```
BOOL ResetEvent (HANDLE hEvent);
```

Функция PulseEvent освобождает все потоки, ожидающие наступления сбрасываемого вручную события, и после этого сразу же автоматически сбрасывается. В случае использования автоматически сбрасываемого события PulseEvent освобождает только один ожидающий поток, если таковые имеются.

Семафоры

Объекты синхронизации семафоры (semaphores) содержат в себе счетчик текущего числа ресурсов. Семафор находится в свободном состоянии, когда значение его счетчика больше 0. При равенстве счетчика 0, объект переходит в занятое состояние.

Поток получает доступ к ресурсу, вызывая одну из ожидающих функций и передавая ей описатель семафора, который охраняет этот ресурс. Wait-функция проверяет у семафора счетчик текущего числа ресурсов: если его значение больше 0 (семафор свободен), уменьшает значение этого счетчика на 1, и вызывающий поток остается планируемым. Если ожидающая функция определяет, что счетчик текущего числа ресурсов равен 0 (семафор занят), система переводит вызывающий поток в состояние ожидания. Когда другой поток увеличит значение этого счетчика, система возобновит выполнение ждущего потока. Поток же в свою очередь захватит ресурс, уменьшив значение счетчика семафора на 1. Следует отметить, что система не допускает присвоения отрицательных значений счетчику числа ресурсов семафора.

Объект ядра «семафор» создается вызовом функции CreateSemapbore.

```
HANDLE CreateSemaphore(
          LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
          LONG lInitialCount,
          LONG lMaximumCount,
          LPCTSTR lpName
);
```

Параметр lMaximumCount сообщает системе максимальное допустимое значение счетчика семафора (общее количество ресурсов). Параметр lInitialCount указывает начальное значение счетчика (сколько ресурсов доступно изначально). Значение параметра lInitialCount должно удовлетворять условию $0 \le$ lInitialCount $\le$ lMaximumCount. При успешном выполнении функция вернет дескриптор семафора, в противном случае NULL.

Для увеличения значения счетчика текущего числа ресурсов (освобождения ресурсов) используется функция ReleaseSemaphore.

```
BOOL ReleaseSemaphore(
          HANDLE hSemaphore,
          LONG lReleaseCount,
          LPLONG lpPreviousCount
);
```

Данная функция складывает величину lReleaseCount (количество освобождаемых ресурсов) со значением счетчика текущего числа ресурсов. Обычно в параметре lReleaseCount передают 1, но возможно указать и большее значение. Функция возвращает предыдущее значение счетчика ресурсов в lpPreviousCount. Если в программе не требуется это значение, допустима передача значения NULL в качестве значения данного параметра.

Мьютекс

Объект ядра взаимное исключение (mutual exception), или мьютекс (mutex), гарантируют потокам взаимоисключающий доступ к ресурсу. По своему поведению мьютекс практически аналогичен критической секциии, но в отличие от последней обеспечивает более универсальную функциональность. Основное отличие заключается в том, что критические секции являются объектами пользовательского режима, а мьютексы – объектами ядра. Таким образом, мьютекс позволяет синхронизировать доступ к ресурсу нескольких потоков из разных процессов. Кроме того мьютекс позволяет задать максимальное время ожидания доступа к ресурсу.

Если поток является владельцем мьютекса, он обладает правом эксклюзивного использования ресурса, который защищается этим мьютексом. Ни один другой поток не может завладеть мьютексом, который уже принадлежит одному из потоков. Но поток, которому уже принадлежит мьютекс, может стать владельцем мьютекса повторно. В таком случае поток обязан освободить мьютекс столько раз, сколько он его захватил.

```
HANDLE CreateMutex(
          LPSECURITY_ATTRIBUTES lpMutexAttributes,
          BOOL bInitialOwner,
          LPCTSTR lpName

);
```

Параметр определяет начальное состояние мьютекса. Если данный параметр равен FALSE, созданный мьютекс не принадлежит ни одному из потоков и находится в свободном состоянии. Если же значение равно TRUE, поток сразу же захватывает мьютекс.

Возвращаемое значение функции содержит дескриптор созданного мьютекса. Значение NULL указывает на ошибку при вызове функции.

Поток получает доступ к разделяемому ресурсу, вызывая одну из ожидающих функций и передавая ей описатель мьютекса, который охраняет этот ресурс. Wait-функция проверяет состояние мьютекса. Если мьютекс свободен, то поток захватывает его, переводя в занятое состояние. Если Wait-функция определяет, что мьютекс занят, вызывающий поток переходит в состояние ожидания, за исключением случая, если владелец мьютекса совпадает с вызывающим потоком.

Когда поток, занимающий ресурс, заканчивает с ним работать, необходимо освободить мьютекс вызовом функции ReleaseMutex.

```
BOOL ReleaseMutex(HANDLE hMutex);
```

В качестве единственного параметра функция получает дескриптор мьютекса. В случае неуспешного завершения операции, функция возвращает значение FALSE. Данная ситуация может возникнуть, когда мьютекс не принадлежит вызывающему потоку.

Задание

Реализовать на языке VB .NET 2005 консольное приложение «Война потоков», сохранив системные вызовы и использование объектов синхронизации. Потоки необходимо реализовать стандартными средствами .NET. Для генерации объявления функций рекомендуется использовать программу API Viewer 2004.

Приложение представляет собой игру, которая называется «Война потоков». Игра имеет следующие правила. Пользователь управляет перемещающейся в нижней части экрана пушкой и может стрелять по врагам, которые летают по всему экрану. Цель игры – уничтожить как можно больше противников. Управление перемещением пушки осуществляется при помощи клавиш «влево» и «вправо». Для выстрела используется клавиша «пробел». Пользователь может совершить одновременно только три выстрела. Каждую секунду может появиться новый противник. Со временем вероятность

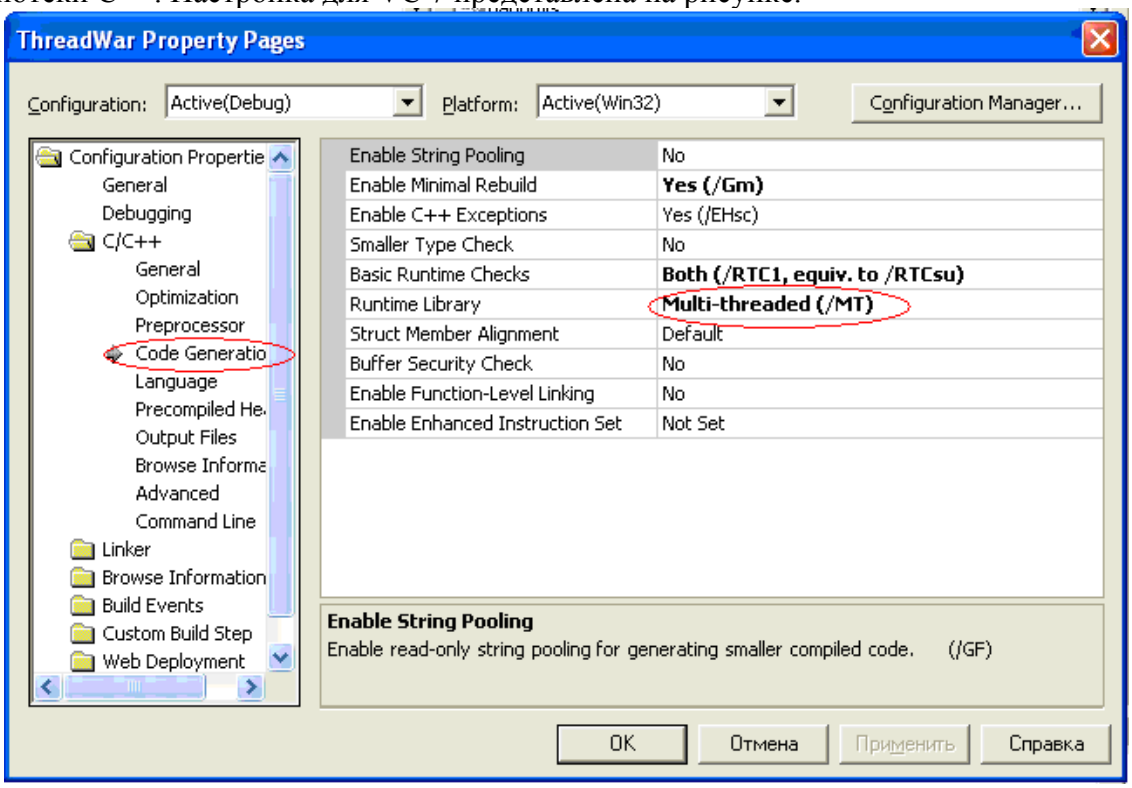
возникновения нового противника увеличивается. Кроме того, со временем противники начинают двигаться быстрее. За каждого уничтоженного противника начисляется одно очко. Если противник перемещается за край экрана, считается, что пользователь промахнулся. Если было упущено 30 врагов, считается, что игра проиграна. Враги не появляются до тех пор, пока не нажата одна из кнопок управления курсором («влево» или «вправо»). Если кнопки не нажаты в течение 15 секунд, враги появятся на экране автоматически.

Программный код игры на языке С

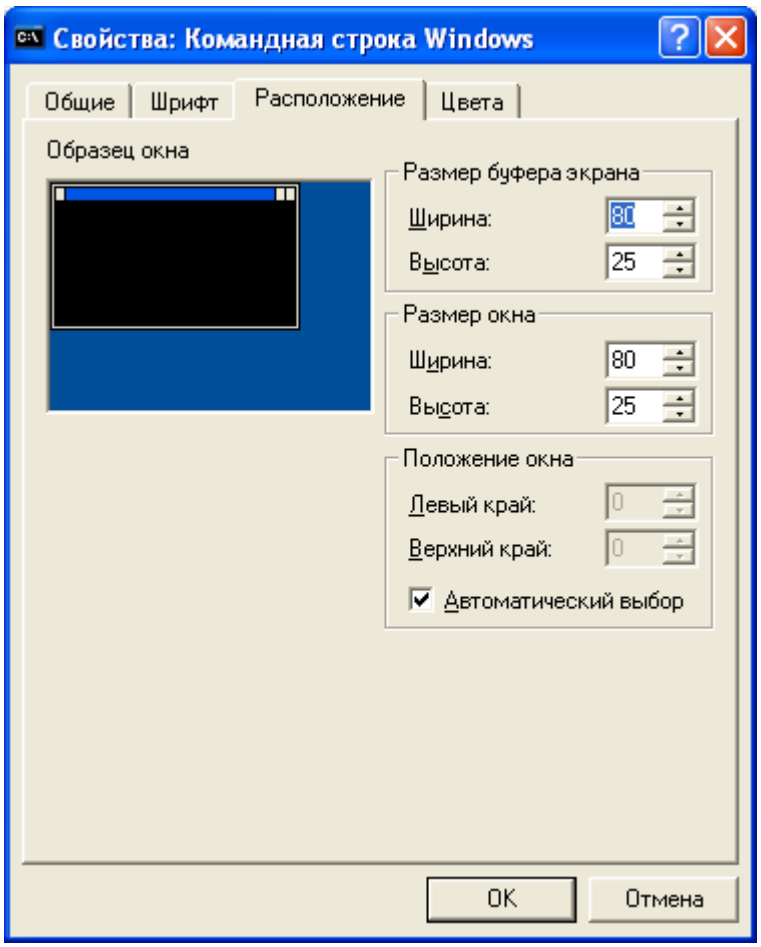
Ниже представлен программный код, реализующий игру на языке С.

1. Внимательно изучите листинг программы и комментарий к нему.
2. Запустите программу на выполнение в среде Microsoft Visual C++.

Учтите, что при компиляции программы необходимо изменить конфигурацию проекта таким образом, чтобы при компиляции использовалась многопоточная версия библиотеки C++. Настройка для VC 7 представлена на рисунке.



Также следует учесть, что при запуске консоль полностью не помещается на экране, и поэтому необходимо изменить ее размеры по умолчанию. Для чего необходимо вызвать контекстное меню заголовка окна, выбрать пункт «Умолчания» и установить требуемый размер буфера экрана.




```
#include "stdafx.h"
#include <windows.h>
#include <process.h>
#include <stdlib.h>
#include <time.h>
#include <stdio.h>

// Объекты синхронизации
HANDLE screenlock; // изменением экрана занимается только один поток
HANDLE bulletsem;  // можно выстрелить только три раза подряд
HANDLE startevt;       // игра начинается с нажатием клавиши "влево" или
"вправо"
HANDLE conin, conout;  // дескрипторы консоли
HANDLE mainthread; // Основной поток main
CRITICAL_SECTION gameover;
CONSOLE_SCREEN_BUFFER_INFO info; // информация о консоли

// количество попаданий и промахов
long hit=0;
long miss=0;
char badchar[]="-\\|/";

// Создание случайного числа от n0 до n1
int random(int n0, int n1)
{
        if (n0==0&&n1==1) return rand()%2;
        return rand()%(n1-n0)+n0;
}

// вывести на экран символ в позицию x и y
```

```
	void writeat (int x,int y, wchar_t c)
	{
	// Блокировать вывод на экран при помощи мьютекса
		WaitForSingleObject(screenlock,INFINITE);
		COORD pos={x,y};
		DWORD res;
		WriteConsoleOutputCharacter(conout, &c , 1 , pos , &res );
		ReleaseMutex(screenlock);
	}

	// Получить нажатие на клавишу (счетчик повторейний в ct)
	int getakey(int &ct)
	{
		INPUT_RECORD input;
		DWORD res;
		while (1)
		{
			ReadConsoleInput (conin, &input, 1 , &res);
			// игнорировать другие события
			if (input.EventType!=KEY_EVENT) continue;
			// игнорировать события отпускания клавиш
			// нас интересуют только нажатия
			if (!input.Event.KeyEvent.bKeyDown) continue;
			ct=input.Event.KeyEvent.wRepeatCount;
			return input.Event.KeyEvent.wVirtualKeyCode;
		}
	}

	// Определить символ в заданной позиции экрана
	int getat(int x, int y)
	{
		wchar_t c;
		DWORD res;
		COORD org={x,y};
		  // Блокировать доступ к консоли до тех пор, пока процедура не
будет выполнена
		WaitForSingleObject(screenlock,INFINITE);
		ReadConsoleOutputCharacter(conout,&c,1,org,&res);
		ReleaseMutex(screenlock); // unlock
		return c;
	}

	// Отобразить очки в заголовке окна и проверить условие завершения игры
	void score(void)
	{
		wchar_t  s[128];
		  swprintf(s, L"Война потоков - Попаданий:%d, Промахов:%d", hit,
miss);
		SetConsoleTitle(s);
		if (miss>=30)
		{
			EnterCriticalSection(&gameover);
			SuspendThread(mainthread); // stop main thread
			  MessageBox(NULL,L"Игра окончена!" ,L"Thread War", MB_OK|
MB_SETFOREGROUND);
			exit(0); // не выходит из критической секции
		}
	}

	// это поток противника
	void badguy(void *_y)
	{
		int y=(int)_y; // случайная координата y
```

```
        int dir;
        int x;
        // нечетные у появляются слева, четные у появляются справа
        x=y%2?0:info.dwSize.X;
        // установить направление в зависимости от начальной позиции
        dir=x?-1:1;
        // пока противник находится в пределах экрана
        while ((dir==1&&x!=info.dwSize.X)||(dir==-1&&x!=0))

        {
                int dly;
                BOOL hitme=FALSE;
                writeat(x,y,badchar[x%4]);

                for (int i=0;i<15;i++)
                {
                        Sleep(40);
                        if (getat(x,y)=='*')
                        {
                                hitme=TRUE;
                                break;
                        }
                }
                writeat(x,y, ' ');

                if (hitme)
                {
                        // в противника попали!
                        MessageBeep(-1);
                        InterlockedIncrement(&hit) ;
                        score();
                        _endthread();
                }
                x+=dir;
        }
// противник убежал!
        InterlockedIncrement(&miss) ;
        score();
}

// этот поток занимается созданием потоков противников
void badguys(void *)
{
   // ждем сигнала к началу игры в течение 15 секунд
        WaitForSingleObject(startevt, 15000);
        // создаем случайного врага
        // каждые 5 секунд появляется шанс создать
        // противника с координатами от 1 до 10
        while (1)
        {
                if (random(0,100)<(hit+miss)/25+20)
                // со временем вероятность увеличивается
                _beginthread(badguy,0, (void *)(random(1,10)));
                Sleep(1000); // каждую секунду
        }
}

// Это поток пули, каждая пуля - это отдельный поток
void bullet(void *_xy_)
{
        COORD xy=*(COORD *)_xy_;
        if (getat(xy.X,xy.Y)=='*') return; // здесь уже есть пуля
        // надо подождать
        // Проверить семафор
```

```
		// если семафор равен 0, выстрела не происходит
		if (WaitForSingleObject(bulletsem,0)==WAIT_TIMEOUT) return;

		while (--xy.Y)
		{
			writeat(xy.X,xy.Y,'*'); // отобразить пулю
			Sleep(100);
			writeat(xy.X, xy.Y, ' ');    // стереть пулю
		}

		// выстрел сделан.- добавить 1 к семафору
		ReleaseSemaphore ( bulletsem, 1, NULL);
	}

	// Основная программа
	int _tmain(int argc, _TCHAR* argv[])
	{
		HANDLE me;
		// Настройка глобальных переменных
		conin=GetStdHandle(STD_INPUT_HANDLE );
		conout=GetStdHandle(STD_OUTPUT_HANDLE);
		SetConsoleMode(conin, ENABLE_WINDOW_INPUT);
		me=GetCurrentThread(); // не является реальным дескриптором
		   // изменить псевдодескриптор на реальный дескриптор текущего
потока
			DuplicateHandle(GetCurrentProcess(),me,GetCurrentProcess(),
&mainthread, 0, FALSE, DUPLICATE_SAME_ACCESS);
		startevt=CreateEvent(NULL, TRUE, FALSE, NULL);
		screenlock=CreateMutex(NULL, FALSE, NULL);
		InitializeCriticalSection ( &gameover);
		bulletsem=CreateSemaphore(NULL, 3, 3, NULL);
		GetConsoleScreenBufferInfo(conout, &info);

		// Инициализировать отображение информации об очках
		score();
		// Настроить генератор случайных чисел
		srand((unsigned)time(NULL));

		// установка начальной позиции пушки
		int y=info.dwSize.Y-1;
		int x=info.dwSize.X/2;
		// запустить поток badguys; ничего не делать до тех пор,
		// пока не произойдет событие или истекут 15 секунд
		_beginthread (badguys, 0, NULL); // основной цикл игры
		while (1)
		{
			int c,ct;
			writeat(x,y, '|'); // нарисовать пушку
			c=getakey(ct);   // получить символ
			switch (c)
			{
			case VK_SPACE:
				static COORD xy;
				xy.X=x;
				xy.Y=y;
				_beginthread(bullet,0,(void *)&xy);
					Sleep(100); // дать пуле время улететь на
некоторое расстояние
				break;
			case VK_LEFT:  // команда "влево!"
				SetEvent(startevt);    // поток badguys работает
				writeat(x,y,' ');      // убрать с экрана пушку
				while (ct--)        // переместиться
					if (x) x--;
```

```
                        break;
                case VK_RIGHT: // команда "вправо!"; логика та же
                        SetEvent(startevt);
                        writeat(x,y,' ');
                        while (ct--)
                                if (x!=info.dwSize.X-1) x++;
                        break;
                }
        }
}
```

Рассмотрим наиболее существенные моменты предложенного решения. Многопоточный подход позволяет сделать каждую из подпрограмм достаточно простой. Например, каждая пуля следит только за собственным положением и перемещением. Один и тот же код управляет всеми противниками вне зависимости от того, сколько их и где они расположены.

Программа использует несколько синхронизационных объектов. Мьютекс screenlock предотвращает одновременный доступ нескольких потоков к экрану консоли. Семафор bulletsem ограничивает количество пуль, одновременно находящихся. Событие startevt блокирует генерацию врагов до тех пор, пока пользователь не нажмет на клавишу или не истечет 15 секунд. Критическая секция gameover предотвращает многократный вывод на экран сообщения об окончании игры.

В начале работы функция main производит инициализацию множества объектов, включая объекты синхронизации. Дескриптор потока сохраняется таким образом, что когда игра завершается, подпрограмма score блокирует обработку ввода. Следует обратить внимание на то, что дескриптор, возвращаемый функцией GetCurrentThreadHandle, на самом деле не является реальным дескриптором. Это специальное значение, которое обозначает текущий поток. Например, если сохранить это значение, а потом передать его вызову SuspendThread, вы автоматически приостановите выполнение потока, который выполнил этот вызов. Чтобы получить реальный дескриптор, функция main обращается к вызову DuplicateHandle.

Прежде чем функция main войдет в цикл обработки ввода, она запускает функцию badguys в новом потоке. Функция badguys ожидает событие startevt в течение 15 секунд. Если событие происходит или 15 секунд истекают, начинается генерация врагов. Ожидание реализуется при помощи функции WaitForSingleObject,

Выполнив начальную инициализацию, функция main входит в цикл ожидания клавиатурного ввода и вывода на экран консоли изображения пушки. Нажатия на клавиши «влево» и «вправо» изменяют положение пушки, а при нажатии на клавишу «пробел» создается новый программный поток, в котором запускается функция bullet.

Поток badguys каждую секунду с некоторой вероятностью создает нового противника. Для этого он проверяет случайно полученное значение, после чего ждет в течение секунды и вновь проверяет случайное значение. Если он решает, что необходимо создать нового противника, он случайным образом определяет у координату нового врага и запускает поток badguy.

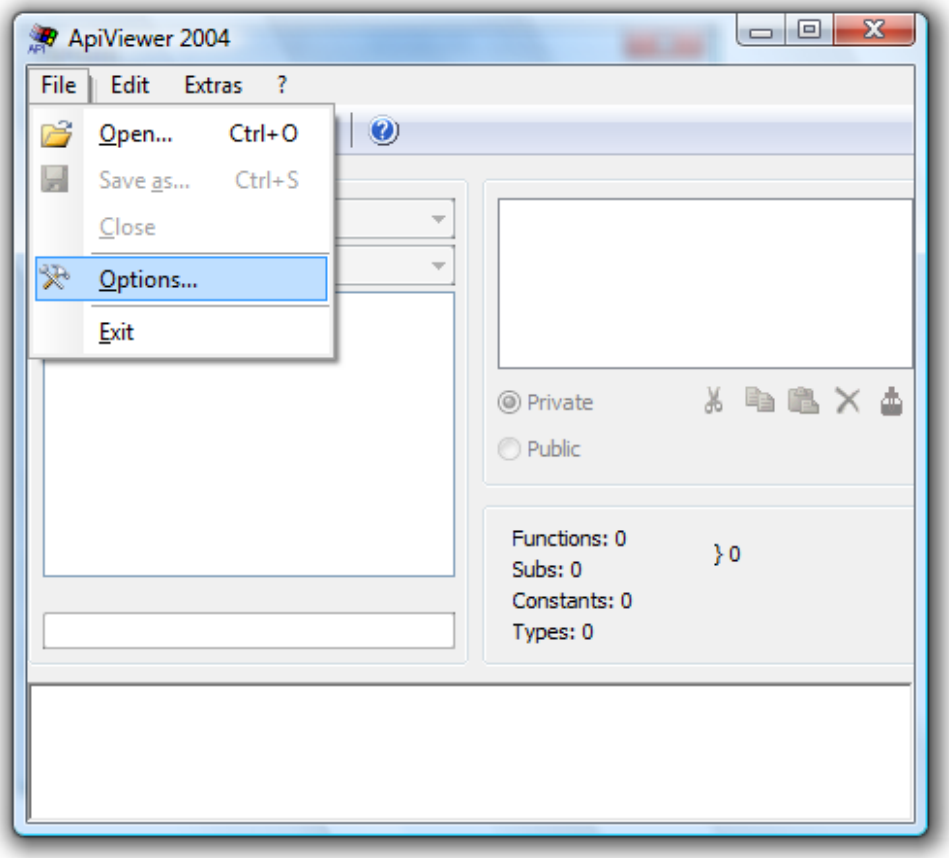
Если сравнивать с другими потоками программы, поток badguy выполняет несколько более сложные действия. Прежде всего, он определяет, будет ли противник двигаться справа налево или слева направо, а затем начинает перемещение противника. По мере того как он перемещает противника, он постоянно проверяет, произошло ли столкновение с пулей. Если столкновение произошло, поток отмечает попадание и обращается к функции score для того, чтобы обновить информацию об очках. Если противник скрывается за границей экрана, поток также отмечает это и вновь обращается к score. Функция score не только обновляет заголовок окна, но также проверяет условие окончания игры.

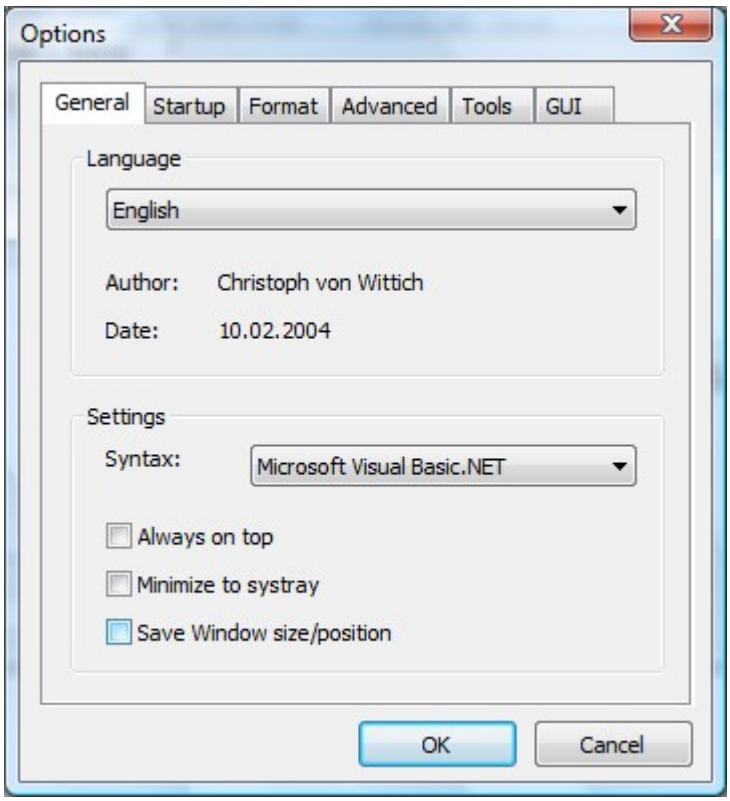
Для обновления переменных hit и miss, в которых хранится количество попаданий и промахов, используется вызов InterlockedIncrement. При этом, если два врага одновременно попытаются изменить значения этих переменных, все сработает нормально.

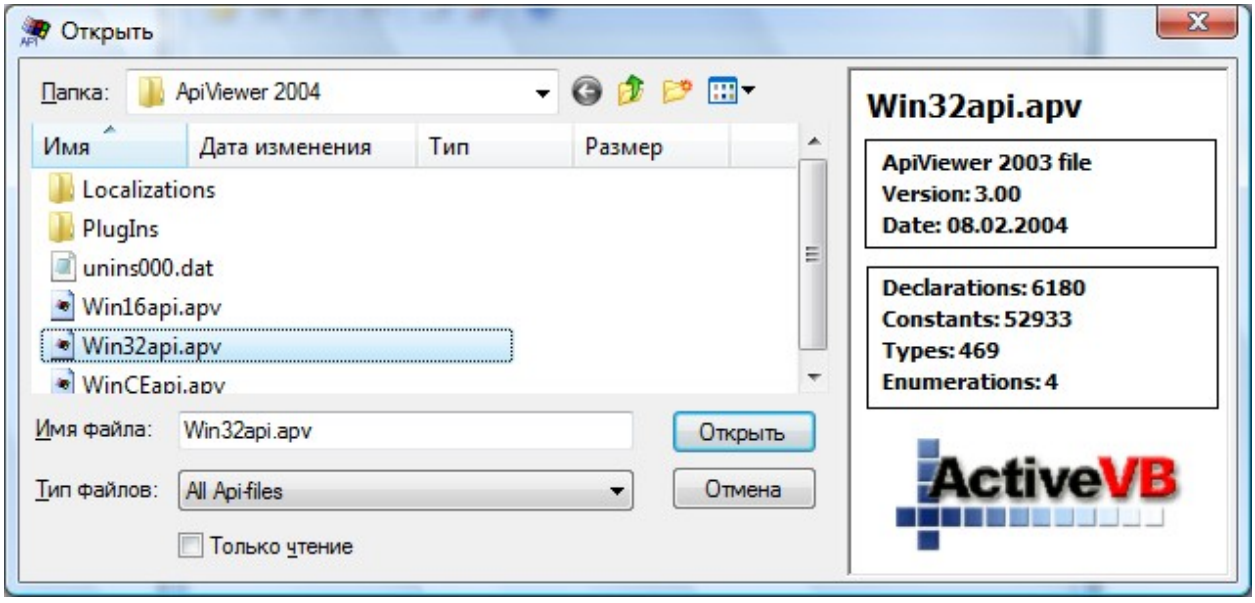
Если срабатывает условие завершения игры, функция score выполняет критическую секцию gameover. Таким образом, сообщение об окончании игры появится на экране только один раз. При этом также происходит завершение работы потока main и обращение к exit для того, чтобы завершить игру. Для этого пользователь должен подтвердить завершение игры.

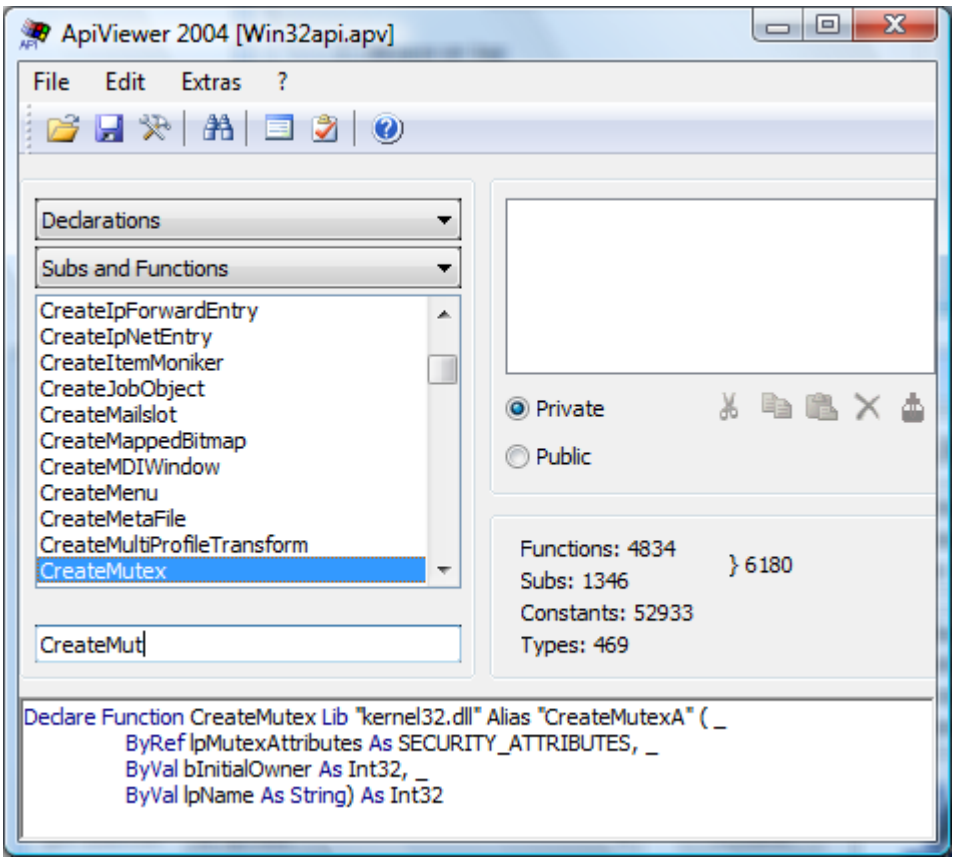
Поток bullet проверяет, можно ли осуществить выстрел. Этот поток немедленно завершает работу в случае, если в позиции, куда должна переместиться пуля, уже находится другая пуля. Это может произойти в случае, если пользователь стреляет слишком быстро. Поток bullet также производит проверку семафора bulletsern. Чтобы выстрел произошел, значение семафора должно быть отличным от нуля. Благодаря семафору на экране не могут одновременно находиться три пули. Если поток bullet не может завладеть семафором, он завершает работу. Таким образом, создать четвертую пулю нельзя – игра просто игнорирует запрос.

Работа с программой ApiViewer 2004




Options
General
Startup
Format
Advanced
Tools
GUI
Language
English
Author: Christoph von Wittich
Date: 10.02.2004
Settings
Syntax:
Microsoft Visual Basic.NET
Always on top
Minimize to systray
Save Window size/position
OK
Cancel


Открыть
Папка:
ApiViewer 2004
Имя
Дата изменения
Тип
Размер
Localizations
PlugIns
unins000.dat
Win16api.apv
Win32api.apv
WinCEapi.apv
Имя файла:
Win32api.apv
Открыть
Тип файлов:
All Api-files
Отмена
Только чтение
Win32api.apv
ApiViewer 2003 file
Version: 3.00
Date: 08.02.2004
Declarations: 6180
Constants: 52933
Types: 469
Enumerations: 4
ActiveVB


ApiViewer 2004 [Win32api.apv]
File
Edit
Extras
?
Declarations
Subs and Functions
CreateIpForwardEntry
CreateIpNetEntry
CreateItemMoniker
CreateJobObject
CreateMailslot
CreateMappedBitmap
CreateMDIWindow
CreateMenu
CreateMetaFile
CreateMultiProfileTransform
CreateMutex
CreateMut
Private
Public
Functions: 4834
Subs: 1346
Constants: 52933
Types: 469
} 6180
Declare Function CreateMutex Lib "kernel32.dll" Alias "CreateMutexA" (_
ByRef lpMutexAttributes As SECURITY_ATTRIBUTES, _
ByVal bInitialOwner As Int32, _
ByVal lpName As String) As Int32