Белгородский Государственный Технологический Университет им. В. Г. Шухова

Кафедра программного обеспечения вычислительной техники  
и автоматизированных систем

## Лабораторная работа №3 по теме: «Синхронизация потоков»

Выполнил:  
студент группы ПВ-31  
Адаменко И. И.

Проверил:  
старший преподаватель  
Гарибов А. И.

Белгород  
2015

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

## Задание

1. Составить программу, которая включает функцию, с помощью которой приложение приостановит свою работу до завершения другого приложения.
2. Исследовать использование мьютексов и событий на конкретном примере. Использовать приложения LAB\_OC\_4\_1 и LAB\_OC\_4\_2, приведённые в папке ЛАБ\_ОС\_4. Разработать аналогичные приложения с использованием семафоров для синхронизации потоков.

## Отчёт

### Основные понятия

Приложение Windows — это совокупность исполняемых программ и вспомогательных файлов.

Процессом называется исполняемый экземпляр приложения.

Каждый исполняемый экземпляр — это отдельный процесс со своей собственной областью памяти.

Если быть более точным, процессом называется исполняемый экземпляр приложения и комплект ресурсов, отводящийся данному исполняемому приложению.

Поток — это внутренняя составляющая процесса, которой операционная система выделяет процессорное время для выполнения кода. Именно потоки исполняют программный код, а не процессы. Каждый процесс должен иметь как минимум один поток. Конечно, основное назначение потоков — дать процессу возможность поддерживать несколько ветвей управления, то есть выполнять больше действий одновременно.

Приоритет потоков:

В старой 16-разрядной Windows существовал только один поток. Более того, в данной системе был реализован метод кооперативной многозадачности, который состоит в том, что каждое приложение само отвечает за высвобождение единственного системного потока, после чего могут выполняться другие приложения. Если программа выполняла задачи, требующие значительного времени, такие как форматирование гибкого диска, все другие загруженные приложения должны были ждать. А если программа, написанная с ошибками, входила в бесконечный цикл, то вся система становилась непригодной для использования, требовала перезагрузки с выключением питания.

Уровни приоритета потоков:

Временной интервал в Windows называется квантом. Каждый поток получает процессорное время, выделяемое квантами времени. Продолжительность кванта времени зависит от аппаратуры и может фактически меняться от потока к потоку. Например, базовое значение в Windows 95 составляет 20 мс, для Windows NT Workstation (на базе процессора Pentium) — 30 мс, для Windows NT Server — 180

мс.

Windows выделяет кванты времени потокам в системе следующим образом:

* Если существуют какие-либо потоки с приоритетом 31, которые требуют процессорного времени (то есть не находятся в состоянии ожидания), операционная система перебирает эти потоки (независимо от того, каким процессам они принадлежат), выделяя по очереди каждому из них кванты времени.
* Если пользователь нажимает клавишу, относящуюся к процессу, потоки которого простаивают, операционная система временно выделяет процессорное время соответствующему потоку, чтобы он мог обработать нажатие клавиши.
* Приоритет 0 зарезервирован исключительно за специальным системным потоком, который называется потоком нулевой страницы. Он освобождает незадействованные области памяти. Существует также поток idle, который работает с уровнем приоритета 0, опрашивая систему в поисках какой-нибудь работы.
* Если поток с произвольным приоритетом выполняется в тот момент, когда потоку с большим приоритетом потребовалось процессорное время (например, он получает сообщение о том, что пользователь щёлкнул мышью), операционная система немедленно вытесняет поток с меньшим приоритетом и отдаёт процессорное время потоку с большим. Таким образом, поток может не успеть завершить выделенный ему квант времени.
* Для того чтобы перейти с одного потока на другой, система осуществляет переключение контекста. Это процедура, сохраняющая состояние процессора (регистров и стека) и загрузки соответствующих значений другого потока.

Назначение приоритета потока:

Назначение потоку приоритета происходит в два этапа. Каждому процессу в момент создания присваивается класс приоритета. Узнать класс приоритета можно с помощью функции GetPriorityClass, а изменить с помощью функции SetPriorityClass.

Диапазон приоритета от 1 до15 известен как диапазон динамического приоритета, а диапазон от 16 до31 — как диапазон приоритета реального времени.

В Windows NT приоритет потока, находящийся в динамическом диапазоне, может временно повышаться операционной системой в различные моменты времени.

Соответственно, нижний уровень приоритета потока (установленный программистом с помощью API функции) называется уровнем его базового приоритета. API-функция Windows NT SetProcessPriorityBoost может использоваться для разрешения или запрещения временных изменений приоритета. Она не поддерживается в Windows 9х.

Бывают также случаи, когда кванты времени, выделяемые потоку, временно увеличиваются.

Стремясь плавно выполнять операции, Windows будет повышать приоритет потока или увеличивать продолжительность его кванта времени при следующих условиях:

* если поток принадлежит приоритетному процессу, то есть процессу, окно которого активно и имеет фокус ввода;
* если поток первым вошёл в состояние ожидания;
* если поток выходит из состояния ожидания;
* если поток совсем не получает процессорного времени.

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** — это обобщённый термин, относящийся к процессу взаимодействия и взаимосвязи потоков. Синхронизация потоков требует привлечения в качестве посредника самой операционной системы. Потоки не могут взаимодействовать друг с другом без её участия.

В Win32 существует несколько методов синхронизации потоков. Бывает, что в конкретной ситуации, один метод более предпочтителен, чем другой.

Многие объекты ядра, включая процесс, поток, файл, мьютекс, семафор, уведомление об изменении файла и событие, могут находиться в одном из двух состояний — «свободно» и «занято». Смысл всей этой «сигнализации» в том, чтобы поток мог приостанавливать свою работу до того момента, когда заданный объект перейдёт в состояние «свободно». Например, поток одного процесса может временно прекратить работу до завершения другого, просто подождав, когда объект ядра этого другого процесса перейдёт в состояние «свободно».

Посредством вызова функций WaitForSingleObject и WaitForMultipleObjects поток приостанавливает своё выполнение до того момента, когда заданный объект (или объекты) перейдёт в состояние «свободно».

Функция WaitForSingleObject переводит вызывающий поток в состояние ожидания до того момента, когда она передаст ему своё возвращаемое значение.

Критические секции:

Это единственный метод синхронизации потоков, который не требует привлечения ядра Windows. Критическая секция не является объектом ядра. Однако этот метод может использоваться только для синхронизации потоков одного процесса.

Критические секции — это объекты, используемые для блокировки доступа всех потоков приложения, кроме одного, к некоторым важным данным в один момент времени.

Мьютексы:

Мьютекс — 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 определённом смысле мьютекс похож на выключатель, которым может пользоваться любой поток по принципу «первым пришёл — первым обслужили».

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.

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 создаёт 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 произойдёт, система выведет из состояния ожидания поток В, который теперь владеет информацией, что поток А завершил выполнение своей задачи.

Семафоры:

В семафорах применён принцип действия мьютексов, но с добавлением одной существенной детали. В них заложена возможность подсчёта ресурсов, что позволяет заранее определённому числу потоков одновременно войти в синхронизуемый участок кода. Для создания семафора используется функция CreateSemaphore.

Ждущие таймеры:

Ждущий таймер 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 трёх режимах. В режиме «ручного сброса» таймер переходит в установленное состояние при истечении заданной задержки и остаётся 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 остаётся 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 одного потока. Наконец, ждущий таймер может выполнять функции интервального таймера, который перезапускается с заданной задержкой после каждого срабатывания объекта.

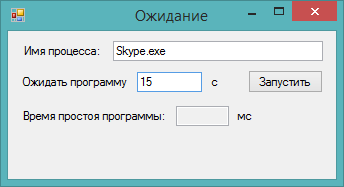
Главная особенность, отличающая ждущие таймеры от системных, — то, что ждущие таймеры могут совместно использоваться несколькими приложениями. Например, можно приостановить несколько приложений в фоновом режиме так, чтобы они «просыпались» каждые несколько часов для выполнения некоторой операции.

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

Далее перечислены функции, предназначенные для работы со ждущими таймерами:

* CancelWaitableTimer — останавливает работу ждущего таймера. Таймер остаётся в текущем состоянии.
* СгеаteWaitableTimer — создаёт объект ждущего таймера. Если таймер с заданным именем уже существует, он открывается.
* OpenWaitableTimer — открывает существующий ждущий таймер.
* SetWaitableTimer — запускает ждущий таймер с заданной продолжительностью и интервалом срабатывания.

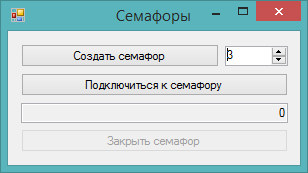
### Результат. Первая часть



Код:

|  |  |
| --- | --- |
| 001 | **public** **static** **void** pauseProcess(**IntPtr** handle, **UInt32** maxPauseMilliseconds, **bool** isPaused) |
| 002 | { |
| 003 | **if** (!isPaused) |
| 004 | { |
| 005 | **const** **UInt32** WAIT\_TIMEOUT = 0x00000102; |
| 006 | **const** **UInt32** WAIT\_ABANDONED = 0x00000080; |
| 007 | **const** **UInt32** WAIT\_OBJECT\_0 = 0x00000000; |
| 008 |  |
| 009 | **UInt32** result = WaitForSingleObject(handle, maxPauseMilliseconds); |
| 010 | **if** (result == WAIT\_TIMEOUT) |
| 011 | { |
| 012 | **throw** **new** **Exception**(*"Время вышло, но объект до сих пор занят"*); |
| 013 | } **else** **if** (result == WAIT\_ABANDONED) { |
| 014 | **throw** **new** **Exception**(*"Объект не был освобожден потоком-владельцем"*); |
| 015 | } **else** **if** (result != WAIT\_OBJECT\_0) { |
| 016 | **throw** **new** **Exception**(*"Ошибка выполнения функции WaitForSingleObject"*); |
| 017 | } |
| 018 | win32APIOld.CloseHandle(handle); |
| 019 | } |
| 020 | } |
| 021 |  |
| 022 | **public** **static** **IntPtr** getProcessHandle(**String** nameSearchProcess) |
| 023 | { |
| 024 | **UInt32**[] processIds = getProcessIds(); |
| 025 | **UInt32** currentProcessId = win32APIOld.GetCurrentProcessId(); |
| 026 |  |
| 027 | **IntPtr** tmp; |
| 028 | **String** tmpStr; |
| 029 | **for** (**int** i = 0; i < processIds.Length; i++) |
| 030 | { |
| 031 | **if** (processIds[i] == currentProcessId) **continue**; |
| 032 |  |
| 033 | tmp = win32APIOld.OpenProcess((**int**)win32APIOld.ProcessAccessFlags.All, |
| 034 | **false**, |
| 035 | (**int**)processIds[i]); |
| 036 |  |
| 037 | tmpStr = getProcessName(tmp); |
| 038 | **if** (tmpStr.Equals(nameSearchProcess)) |
| 039 | { |
| 040 | **return** tmp; |
| 041 | } |
| 042 | **else** |
| 043 | { |
| 044 |  |
| 045 | win32APIOld.CloseHandle(tmp); |
| 046 | } |
| 047 | } |
| 048 |  |
| 049 | **throw** **new** **Exception**(*"Не найден соответствующий процесс"*); |
| 050 | } |
| 051 |  |
| 052 | **private** **static** **UInt32**[] getProcessIds() |
| 053 | { |
| 054 | **UInt32** arraySize = 120; |
| 055 | **UInt32** arrayBytesSize = arraySize \* sizeof(**UInt32**); |
| 056 | **UInt32**[] processIds = **new** **UInt32**[arraySize]; |
| 057 | **UInt32** bytesCopied; |
| 058 |  |
| 059 | **bool** success = EnumProcesses(processIds, arrayBytesSize, **out** bytesCopied); |
| 060 |  |
| 061 | **if** (success && bytesCopied > 0) |
| 062 | { |
| 063 | bytesCopied >>= 2; |
| 064 | **UInt32**[] result = **new** **UInt32**[bytesCopied]; |
| 065 |  |
| 066 | **for** (**int** i = 0; i < bytesCopied; i++) |
| 067 | { |
| 068 | result[i] = processIds[i]; |
| 069 | } |
| 070 |  |
| 071 | **return** result; |
| 072 | } |
| 073 | **else** |
| 074 | { |
| 075 | **return** **new** UInt32[0]; |
| 076 | } |
| 077 | } |
| 078 |  |
| 079 | **private** **static** **String** getProcessName(**IntPtr** handle) |
| 080 | { |
| 081 | **String** result = ""; |
| 082 | **UInt32** t = getProcessModule(handle); |
| 083 |  |
| 084 | **if** (t != 0) |
| 085 | { |
| 086 | **StringBuilder** bl = **new** StringBuilder(255); |
| 087 | **UInt32** size = GetModuleBaseName(handle, **new** IntPtr(t), bl, 255); |
| 088 | result = bl.ToString().Substring(0, (**int**)size); |
| 089 | } |
| 090 |  |
| 091 | **return** result; |
| 092 | } |
| 093 |  |
| 094 | **private** **static** **UInt32** getProcessModule(**IntPtr** handle) |
| 095 | { |
| 096 | **UInt32** arraySize = 1000; |
| 097 | **UInt32** arrayBytesSize = arraySize \* sizeof(**UInt32**); |
| 098 | **UInt32**[] result = **new** **UInt32**[arraySize]; |
| 099 | **UInt32** bytesCopied; |
| 100 |  |
| 101 | EnumProcessModules(handle, result, arrayBytesSize, **out** bytesCopied); |
| 102 | bytesCopied >>= 2; |
| 103 |  |
| 104 | **if** (bytesCopied > 0) |
| 105 | { |
| 106 | **return** result[0]; |
| 107 | } |
| 108 | **else** |
| 109 | { |
| 110 | **return** 0; |
| 111 | } |
| 112 | } |

### Результат. Вторая часть



Код:

|  |  |
| --- | --- |
| 001 | **public** **static** **IntPtr** createSemaphore(**decimal** maxValue, **String** nameSemaphore) |
| 002 | { |
| 003 | **return** CreateSemaphore(**IntPtr**.Zero, (**int**)maxValue, (int)maxValue, nameSemaphore); |
| 004 | } |
| 005 |  |
| 006 | **public** **static** **IntPtr** openSemaphore(**String** nameSemaphore) |
| 007 | { |
| 008 | **return** OpenSemaphore((**uint**)SyncObjectAccess.SEMAPHORE\_ALL\_ACCESS, **false**, nameSemaphore); |
| 009 | } |
| 010 |  |
| 011 | **public** **static** **bool** releaseSemaphore(**IntPtr** semaphore) |
| 012 | { |
| 013 | **return** ReleaseSemaphore(semaphore, 1, **IntPtr**.Zero); |
| 014 | } |
| 015 |  |
| 016 | **public** **static** **bool** closeSemaphore(**IntPtr** semaphore) |
| 017 | { |
| 018 | releaseSemaphore(semaphore); |
| 019 | **return** CloseHandle(semaphore); |
| 020 | } |
| 021 |  |
| 022 | **public** **static** **bool** pause(**IntPtr** semaphore) |
| 023 | { |
| 024 | **const** **UInt32** WAIT\_TIMEOUT = 0x00000102; |
| 025 | **const** **UInt32** WAIT\_ABANDONED = 0x00000080; |
| 026 | **const** **UInt32** WAIT\_OBJECT\_0 = 0x00000000; |
| 027 |  |
| 028 | **UInt32** result = WaitForSingleObject(semaphore, 0xFFFFFFFF); |
| 029 | **if** (result == WAIT\_TIMEOUT) |
| 030 | { |
| 031 | **throw** **new** **Exception**(*"Время вышло, но объект до сих пор занят"*); |
| 032 | } **else** **if** (result == WAIT\_ABANDONED) { |
| 033 | **throw** **new** **Exception**(*"Ошибка мьютекса"*); |
| 034 | } **else** **if** (result != WAIT\_OBJECT\_0) { |
| 035 | **throw** **new** **Exception**(*"Ошибка выполнения функции WaitForSingleObject"*); |
| 036 | } |
| 037 | **return** **true**; |
| 038 | } |