|  |
| --- |
| **Лабораторная работа. Синхронизация нитей внутри процесса в ОС Windows.** |
|  |  |
| **Цель работы**: изучить возможности применения механизмов синхронизации нитей одного процесса в операционной системе Windows.    **Темы для предварительной проработки**: процессы и нити в операционных системах, жизненный цикл процесса, средства взаимодействия процессов IPC, интерфейс прикладного программирования API WIN32.    **Теоретические сведения**    *1. Процессы, нити и волокна в ОС Windows. Дополнительные данные.*  Процессом (process) называется экземпляр программы, загруженной в память. Этот экземпляр может создавать нити (thread), которые представляют собой последовательность инструкций на выполнение. Важно понимать, что выполняются не процессы, а именно нити. Причем любой процесс имеет хотя бы одну нить. Эта нить называется главной (основной) нитью приложения.  Так как практически всегда нитей гораздо больше, чем физических процессоров для их выполнения, то нити на самом деле выполняются не одновременно, а по очереди (распределение процессорного времени происходит именно между нитями). Но переключение между ними происходит так часто, что кажется, будто они выполняются параллельно.  В зависимости от ситуации нити могут находиться в трех состояниях. Во-первых, нить может выполняться, когда ей выделено процессорное время, т.е. она может находиться в состоянии активности. Во-вторых, она может быть неактивной и ожидать выделения процессора, т.е. быть в состоянии готовности. И есть еще третье, тоже очень важное состояние - состояние блокировки. Когда нить заблокирована, ей вообще не выделяется время. Обычно блокировка ставится на время ожидания какого-либо события. При возникновении этого события нить автоматически переводится из состояния блокировки в состояние готовности. Например, если одна нить выполняет вычисления, а другая должна ждать результатов, чтобы сохранить их на диск. Вторая могла бы использовать цикл типа "**while( !isCalcFinished ) continue;**", но легко убедиться на практике, что во время выполнения этого цикла процессор занят на 100 % (это называется активным ожиданием). Таких вот циклов следует по возможности избегать, в чем оказывает неоценимую помощь механизм блокировки. Вторая нить может заблокировать себя до тех пор, пока первая не установит событие, сигнализирующее о том, что чтение окончено.    *2. Синхронизация нитей в ОС Windows.*  В Windows реализована *вытесняющая многозадачность* - это значит, что в любой момент система может прервать выполнение одной нити и передать управление другой. Ранее, в Windows 3.1, использовался способ организации, называемый *кооперативной многозадачностью*: система ждала, пока нить сама не передаст ей управление и именно поэтому в случае зависания одного приложения приходилось перезагружать компьютер.  Все нити, принадлежащие одному процессу, разделяют некоторые общие ресурсы - такие, как адресное пространство оперативной памяти или открытые файлы. Эти ресурсы принадлежат всему процессу, а значит, и каждой его нити. Следовательно, каждая нить может работать с этими ресурсами без каких-либо ограничений. Но... Если одна нить еще не закончила работать с каким-либо общим ресурсом, а система переключилась на другую нить, использующую этот же ресурс, то результат работы этих нитей может чрезвычайно сильно отличаться от задуманного. Такие конфликты могут возникнуть и между нитями, принадлежащими различным процессам. Всегда, когда две или более нитей используют какой-либо общий ресурс, возникает эта проблема.  Пример. Несинхронизированная работа нитей: если временно приостановить выполнение нити вывода на экран (пауза), фоновая нить заполнения массива будет продолжать работать.    #include <windows.h>  #include <stdio.h>  int a[5];  HANDLE hThr;  unsigned long uThrID;  void Thread( void\* pParams )  {              int i, num = 0;              while (1)              {                          for (i=0; i<5; i++) a[i] = num;                          num++;              }  }  int main( void )  {              hThr=CreateThread(NULL,0,(LPTHREAD\_START\_ROUTINE)Thread,NULL,0,&uThrID);              while(1)                          printf("%d %d %d %d %d\n", a[0], a[1], a[2], a[3], a[4]);              return 0;  }    Именно поэтому необходим механизм, позволяющий потокам согласовывать свою работу с общими ресурсами. Этот механизм получил название механизма *синхронизации нитей* (thread synchronization).  Этот механизм представляет собой набор объектов операционной системы, которые создаются и управляются программно, являются общими для всех нитей в системе (некоторые - для нитей, принадлежащих одному процессу) и используются для координирования доступа к ресурсам. В качестве ресурсов может выступать все, что может быть общим для двух и более нитей - файл на диске, порт, запись в базе данных, объект GDI, и даже глобальная переменная программы (которая может быть доступна из нитей, принадлежащих одному процессу).  Объектов синхронизации существует несколько, самые важные из них - это взаимоисключение (**mutex**), критическая секция (**critical section**), событие (**event**) и семафор (**semaphore**). Каждый из этих объектов реализует свой способ синхронизации. Также в качестве объектов синхронизации могут использоваться сами процессы и нити (когда одна нить ждет завершения другой нити или процесса); а также файлы, коммуникационные устройства, консольный ввод и уведомления об изменении.  Любой объект синхронизации может находиться в так называемом *сигнальном состоянии*. Для каждого типа объектов это состояние имеет различный смысл. Нити могут проверять текущее состояние объекта и/или ждать изменения этого состояния и таким образом согласовывать свои действия. При этом гарантируется, что когда нить работает с объектами синхронизации (создает их, изменяет состояние) система не прервет ее выполнения, пока она не завершит это действие. Таким образом, все конечные операции с объектами синхронизации являются атомарными (неделимыми.    *3. Работа с объектами синхронизации*  Чтобы создать тот или иной объект синхронизации, производится вызов специальной функции WinAPI типа **Create...** (напр. **CreateMutex**). Этот вызов возвращает дескриптор объекта (**HANDLE**), который может использоваться всеми нитями, принадлежащими данному процессу. Есть возможность получить доступ к объекту синхронизации из другого процесса - либо унаследовав дескриптор этого объекта, либо, что предпочтительнее, воспользовавшись вызовом функции открытия объекта (**Open...**). После этого вызова процесс получит дескриптор, который в дальнейшем можно использовать для работы с объектом. Объекту, если только он не предназначен для использования внутри одного процесса, обязательно присваивается имя. Имена всех объектов должны быть различны (даже если они разного типа). Нельзя, например, создать событие и семафор с одним и тем же именем.  По имеющемуся дескриптору объекта можно определить его текущее состояние. Это делается с помощью т.н. *ожидающих* функций. Чаще всего используется функция **WaitForSingleObject**. Эта функция принимает два параметра, первый из которых - дескриптор объекта, второй - время ожидания в мсек. Функция возвращает **WAIT\_OBJECT\_0**, если объект находится в сигнальном состоянии, **WAIT\_TIMEOUT** - если истекло время ожидания, и **WAIT\_ABANDONED**, если объект-взаимоисключение не был освобожден до того, как владеющая им нить завершилась. Если время ожидания указано равным нулю, функция возвращает результат немедленно, в противном случае она ждет в течение указанного промежутка времени. В случае, если состояние объекта станет сигнальным до истечения этого времени, функция вернет **WAIT\_OBJECT\_0**, в противном случае функция вернет **WAIT\_TIMEOUT**. Если в качестве времени указана символическая константа **INFINITE**, то функция будет ждать неограниченно долго, пока состояние объекта не станет сигнальным.  Очень важен тот факт, что обращение к ожидающей функции *блокирует* текущую нить, т.е. пока нить находится в состоянии ожидания, ей не выделяется процессорного времени.    ***Критические секции***  Объект-критическая секция помогает программисту выделить участок кода, где нить получает доступ к разделяемому ресурсу, и предотвратить одновременное использование ресурса. Перед использованием ресурса нить *входит в критическую секцию* (вызывает функцию **EnterCriticalSection**). Если после этого какая-либо другая нить попытается войти в ту же самую критическую секцию, ее выполнение приостановится, пока первая нить не *покинет секцию* с помощью вызова **LeaveCriticalSection**. Используется только для нитей одного процесса. Порядок входа в критическую секцию не определен.  Существует также функция **TryEnterCriticalSection**, которая проверяет, занята ли критическая секция в данный момент. С ее помощью нить в процессе ожидания доступа к ресурсу может не блокироваться, а выполнять какие-то полезные действия.  Пример. Синхронизация нитей с помощью критических секций.    #include <windows.h>  #include <stdio.h>  CRITICAL\_SECTION cs;  int a[5];  HANDLE hThr;  unsigned long uThrID;  void Thread( void\* pParams )  {              int i, num = 0;              while (1)              {                          EnterCriticalSection( &cs );                          for (i=0; i<5; i++) a[i] = num;                          num++;                          LeaveCriticalSection( &cs );              }  }  int main( void )  {              InitializeCriticalSection( &cs );              hThr=CreateThread(NULL,0,(LPTHREAD\_START\_ROUTINE)Thread,NULL,0,&uThrID);              while(1)              {                          EnterCriticalSection( &cs );                          printf("%d %d %d %d %d\n", a[0], a[1], a[2], a[3], a[4]);                          LeaveCriticalSection( &cs );              }              return 0;  }    ***Взаимоисключения***  Объекты-взаимоисключения (мьютексы, mutex - от MUTual EXclusion) позволяют координировать взаимное исключение доступа к разделяемому ресурсу. Сигнальное состояние объекта (т.е. состояние "установлен") соответствует моменту времени, когда объект не принадлежит ни одной нити и его можно "захватить". И наоборот, состояние "сброшен" (не сигнальное) соответствует моменту, когда какая-либо нить уже владеет этим объектом. Доступ к объекту разрешается, когда нить, владеющая объектом, освободит его.  Две (или более) нити могут создать мьютекс с одним и тем же именем, вызвав функцию **CreateMutex**. Первая нить действительно создает мьютекс, а следующие - получают дескриптор уже существующего объекта. Это дает возможность нескольким нитям получить дескриптор одного и того же мьютекса, освобождая программиста от необходимости заботиться о том, кто в действительности создает мьютекс. Если используется такой подход, желательно установить флаг ***bInitialOwner*** в **FALSE**, иначе возникнут определенные трудности при определении действительного создателя мьютекса.  Несколько нитей могут получить дескриптор одного и того же мьютекса, что делает возможным взаимодействие между процессами. Можно использовать следующие механизмы такого подхода:   * Дочерний процесс, созданный при помощи функции **CreateProcess** может наследовать дескриптор мьютекса в случае, если при создании мьютекса функцией **CreateMutex** был указан параметр ***lpMutexAttributes***. * Нить может получить дубликат существующего мьютекса с помощью функции **DuplicateHandle**. * Нить может указать имя существующего мьютекса при вызове функций **OpenMutex** или **CreateMutex**.   Для того чтобы объявить взаимоисключение принадлежащим текущей нити, надо вызвать одну из ожидающих функций. Нить, которой принадлежит объект, может его "захватывать" повторно сколько угодно раз (это не приведет к самоблокировке), но столько же раз она должна будет его освобождать с помощью функции **ReleaseMutex**.  Для синхронизации нитей одного процесса более эффективно использование критических секций.  Пример. Синхронизация нитей с помощью мьютексов.    #include <windows.h>  #include <stdio.h>  HANDLE hMutex;  int a[5];  HANDLE hThr;  unsigned long uThrID;  void Thread( void\* pParams )  {              int i, num = 0;              while (1)              {                          WaitForSingleObject( hMutex, INFINITE );                          for (i=0; i<5; i++) a[i] = num;                          num++;                          ReleaseMutex( hMutex );              }  }  int main( void )  {              hMutex=CreateMutex( NULL, FALSE, NULL );              hThr=CreateThread(NULL,0,(LPTHREAD\_START\_ROUTINE)Thread,NULL,0,&uThrID);              while(1)              {                          WaitForSingleObject( hMutex, INFINITE );                          printf("%d %d %d %d %d\n", a[0], a[1], a[2], a[3], a[4]);                          ReleaseMutex( hMutex );              }              return 0;  }    ***События***  Объекты-события используются для уведомления ожидающих нитей о наступлении какого-либо события. Различают два вида событий - с ручным и автоматическим сбросом. Ручной сброс осуществляется функцией **ResetEvent**. События с ручным сбросом используются для уведомления сразу нескольких нитей. При использовании события с автосбросом уведомление получит и продолжит свое выполнение только одна ожидающая нить, остальные будут ожидать дальше.  Функция **CreateEvent** создает объект-событие, **SetEvent** - устанавливает событие в сигнальное состояние, **ResetEvent** - сбрасывает событие. Функция **PulseEvent** устанавливает событие, а после возобновления ожидающих это событие нитей (всех при ручном сбросе и только одной при автоматическом), сбрасывает его. Если ожидающих нитей нет, **PulseEvent** просто сбрасывает событие.  Пример. Синхронизация нитей с помощью событий.    #include <windows.h>  #include <stdio.h>  HANDLE hEvent1, hEvent2;  int a[5];  HANDLE hThr;  unsigned long uThrID;  void Thread( void\* pParams )  {              int i, num = 0;              while (1)              {                          WaitForSingleObject( hEvent2, INFINITE );                          for (i=0; i<5; i++) a[i] = num;                          num++;                          SetEvent( hEvent1 );              }  }  int main( void )  {              hEvent1=CreateEvent( NULL, FALSE, TRUE, NULL );              hEvent2=CreateEvent( NULL, FALSE, FALSE, NULL );              hThr=CreateThread(NULL,0,(LPTHREAD\_START\_ROUTINE)Thread,NULL,0,&uThrID);              while(1)              {                          WaitForSingleObject( hEvent1, INFINITE );                          printf("%d %d %d %d %d\n", a[0], a[1], a[2], a[3], a[4]);                          SetEvent( hEvent2 );              }              return 0;  }    ***Семафоры***  Объект-семафор - это фактически объект-взаимоисключение со счетчиком. Данный объект позволяет "захватить" себя определенному количеству нитей. После этого "захват" будет невозможен, пока одна из ранее "захвативших" семафор нитей не освободит его. Семафоры применяются для ограничения количества нитей, одновременно работающих с ресурсом. Объекту при инициализации передается максимальное число нитей, после каждого "захвата" счетчик семафора уменьшается. Сигнальному состоянию соответствует значение счетчика больше нуля. Когда счетчик равен нулю, семафор считается не установленным (сброшенным).  Функция **CreateSemaphore** создает объект-семафор с указанием и максимально возможного начального его значения, **OpenSemaphore** – возвращает дескриптор существующего семафора, захват семафора производится с помощью *ожидающих* функций, при этом значение семафора уменьшается на единицу, **ReleaseSemaphore** - освобождение семафора с увеличением значения семафора на указанное в параметре число.  Пример. Синхронизация нитей с помощью семафоров.    #include <windows.h>  #include <stdio.h>  HANDLE hSem;  int a[5];  HANDLE hThr;  unsigned long uThrID;  void Thread( void\* pParams )  {              int i, num = 0;              while (1)              {                          WaitForSingleObject( hSem, INFINITE );                          for (i=0; i<5; i++) a[i] = num;                          num++;                          ReleaseSemaphore( hSem, 1, NULL );              }  }  int main( void )  {              hSem=CreateSemaphore( NULL, 1, 1, "MySemaphore1" );              hThr=CreateThread(NULL,0,(LPTHREAD\_START\_ROUTINE)Thread,NULL,0,&uThrID);              while(1)              {                          WaitForSingleObject( hSem, INFINITE );                          printf("%d %d %d %d %d\n", a[0], a[1], a[2], a[3], a[4]);                          ReleaseSemaphore( hSem, 1, NULL );              }              return 0;  }    **Защищенный доступ к переменным**  Существует ряд функций, позволяющих работать с глобальными переменными из всех нитей, не заботясь о синхронизации, т.к. эти функции сами за ней следят – их выполнение атомарно. Это функции **InterlockedIncrement**, **InterlockedDecrement**, **InterlockedExchange**, **InterlockedExchangeAdd** и **InterlockedCompareExchange**. Например, функция **InterlockedIncrement** атомарно увеличивает значение 32-битной переменной на единицу, что удобно использовать для различных счетчиков.    Для получения полной информации о назначении, использовании и синтаксисе всех функций WIN32 API необходимо воспользоваться системой помощи MS SDK, входящей в состав сред программирования Borland Delphi или CBuilder, а также MSDN, поставляемым в составе системы программирования Visual C.    **Постановка задачи**: разработать консольное приложение в среде программирования Visual C, которое обязательно использует нити и решает поставленную задачу в соответствии с указанным в индивидуальном задании механизме синхронизации.    **Порядок выполнения работы**    1. Ознакомиться с постановкой задачи и исходными данными. В соответствии с номером по журналу определить вариант задачи.  2. Согласно рекомендациям, приведенным в исходных данных, сконструировать структуру программы.  3. Составить тексты программ и утвердить их у преподавателя.  4. Пункты 1 - 3 должны быть выполнены предварительно до проведения данной лабораторной работы. Утверждение преподавателем текстов программ является допуском к лабораторной работе.  5. Набрать текст программы.  6. Выполнить компиляцию программы.  7. Провести анализ и исправление обнаруженных синтаксических ошибок в тексте программы и повторить пункты 6. и 7. При устранении всех синтаксических ошибок перейти к выполнению пункта 8.  8. Получить решение и, в случае обнаружения логических ошибок, описать и устранить их. Продемонстрировать преподавателю окончательный вариант программы и ее работу.    **Содержание отчета**    1. Тема лабораторной работы.  2. Цель работы.  3. Индивидуальное задание.  4. Метод и алгоритм решения задачи.  5. Текст программы.  6. Результаты работы программы.  7. Выводы по работе.    **Задачи**    I. Задача «производители-потребители». Решается проблема работы с нескольких процессов (нитей) с одним буфером. Часть процессов являются «производителями»: в случайные моменты времени производят запись информации в буфер. Часть процессов являются «потребителями»: в случайные моменты читают информацию из буфера (после чтения информация в буфере теряется). Необходимо организовать такую работу, чтобы не было коллизий при совместной работе «производителей» и «потребителей».  II. Задача «читатели-писатели». Имеются данные, совместно используемые нескольким процессами (нитями). Имеется несколько процессов (нитей), которые только читают эти данные («читатели») и несколько других, которые только записывают данные (или изменяют их) («писатели»). При этом должны удовлетворяться следующие условия:  – любое число «читателей» могут одновременно читать данные;  – записывать данные в определенный момент может только один «писатель».  – когда «писатель» записывает данные, ни один читатель не может его читать.  III. Задача об «обедающих философах». Имеется некоторое количество процессов (нитей) – «философов», которые «сидят вокруг стола с едой» - выполняются в операционной системе. На столе находится «столовые приборы» (тарелки и вилки) – «ресурсы», количество которых равно количеству «философов». Каждый философ некоторое время размышляет, затем пытается взять 2 рядом лежащие (слева и справа от него) вилки. Если это удается, то он принимается за еду. Время приема пищи – произвольно. Решение должно избегать возможности наступления тупика (круговое ожидание) и голодания.    **Индивидуальные задания**  15. Задача I. Механизм синхронизации – события. Буфер – циклический. | |