Оглавление

ЛАБОРАТОРНАЯ РАБОТА №3 Программирование синхронизации процессов в ОС Windows

Цель работы: *разработать программу, осуществляющую решение поставленной задачи с помощью многопоточного приложения средствами Win API*

1. Теоретические сведения

1.1 Алгоритмы синхронизации

Основные теоретические по алгоритмам синхронизации изложены в лабораторной работе №2.

1.2 Аппаратная поддержка взаимоисключений

Основные теоретические сведения по алгоритмам синхронизации изложены в лабораторной работе №2.

Атомарные операции

Атомарные операции обычно имеют более-менее близкое соответствие с командами процессора. Так, например, они могут сводиться к операциям с блокировкой шины (префикс lock) и специальным командам (типа cmpxchg) процессора. ОС Windows предоставляет функции для увеличения (InterlockedIncrement, InterlockedIncrement64) или уменьшения (InterlockedDecrement, InterlockedDecrement64) значения целочисленных переменных и изменения их значений (InterlockedExchange, InterlockedExchange64, InterlockedExchangeAdd, InterlockedExchangePointer), в том числе со сравнением (InterlockedCompareExchange, InterlockedCompareExchangePointer).

```
unsigned _ _stdcall ThreadProc( void *param )
{
  int  i;
  for ( i = 0; i < ASIZE; i++ ) InterlockedIncrement(array+i);
  return 0;
}
```

Еще несколько функций предназначены для работы с односвязными LIFO списками. Функция InitializeSListHead подготавливает начальный указатель на LIFO список, функция InterlockedPushEntrySList добавляет новую запись в список, функция InterlockedPopEntrySList извлекает из списка последнюю добавленную запись и функция InterlockedFlushSList очищает список. Операции изменения указателей в списке являются атомарными.

Критические секции

Термин "критическая секция" обозначает некоторый фрагмент кода, который должен выполняться в исключительном режиме - никакие другие потоки и процессы не должны выполнять этот же фрагмент в то же время. В некоторых операционных системах для однопроцессорных машин вход в критическую секцию просто блокирует работу планировщика до выхода из секции. Этот подход не всегда эффективен: нахождение потока в критической секции не должно влиять на возможность исполнения кода, не принадлежащего именно данной критической секции, другими потоками, особенно на

многопроцессорных машинах. В Windows предусмотрен специальный тип данных, называемый CRITICAL_SECTION и предназначенный для реализации критических секций. В приложении может существовать произвольное количество данных этого типа, реализующих различные критические секции; равно как несколько секций кода могут использовать один общий объект CRITICAL_SECTION.

Существуют несколько основных функции для работы с критическими секциями:

- перед использованием критическая секция должна быть инициализирована с помощью функции InitializeCriticalSection. Объект CRITICAL_SECTION, принадлежащий пользовательскому процессу, может использовать в своей реализации объекты ядра для ожидания;
- для окончательного освобождения ресурсов, после использования критической секции она должна быть удалена с помощью функции DeleteCriticalSection;
- функция EnterCriticalSection, которая соответствует входу в критическую секцию, при необходимости с ожиданием, не ограниченным по времени;
- TryEnterCriticalSection, которая позволяет при необходимости не входить в секцию, если она занята;
- функция LeaveCriticalSection, которая соответствует выходу из этой секции, возможно с пробуждением потоков, ожидающих ее освобождения. При этом система исключает вход в критическую секцию всех остальных потоков процесса, в то время как поток, уже вошедший в данную секцию, может входить в нее рекурсивно - надо лишь, чтобы число выходов из секции соответствовало числу входов:

```
#include <stdio.h>
#include <process.h>
#define _WIN32_WINNT 0x0403
#include <windows.h>

#define THREADS         10
#define ASIZE           10000000
static  LONG               array[ASIZE];
static  CRITICAL_SECTION     CS;

unsigned _ _stdcall ThreadProc( void *param )
{
  int  i;
  for ( i = 0; i < ASIZE; i++ ) {
    EnterCriticalSection( &CS );
    array[i]++;
    LeaveCriticalSection( &CS );
  }
  return 0;
}

int main( void )
{
  HANDLE       hThread[THREADS];
  unsigned     dwThread;
  int          i, errs;
  InitializeCriticalSectionAndSpinCount( &CS, 100 );
  for ( i = 0; i < THREADS; i++ )
    hThread[i] = (HANDLE)_beginthreadex(
```

```
      NULL, 0, ThreadProc, (void*)i, 0, &dwThread );
  WaitForMultipleObjects( THREADS, hThread, TRUE, INFINITE );
  for ( i = 0; i < THREADS; i++ ) CloseHandle( hThread[i] );
  for ( errs=i=0; i<ASIZE; i++ )
    if ( array[i] != THREADS ) errs++;
  if ( errs ) printf("Detected %d errors!\n", errs );
  DeleteCriticalSection( &CS );
  return 0;
}
```

Кроме того, эффективность критических секций на многопроцессорных машинах может быть повышена, если перед началом ожидания занятой секции, вместо перехода к ожиданию в режиме ядра, выполнить предварительный кратковременный цикл с опросом состояния секции. Если секция занимается другим потоком на небольшое время, то такой цикл позволяет дождаться ее освобождения, не переходя в режим ядра. Для этого предназначены функции: InitializeCriticalSectionAndSpinCount и SetCriticalSectionSpinCount. Они позволяют задавать число опросов состояния занятой критической секции перед переходом в режим ядра для ожидания. На однопроцессорных машинах опросы не выполняются и функция InitializeCriticalSectionAndSpinCount ничем не отличается от обычной функции InitializeCriticalSection.

Синхронизация с использованием объектов ядра

В Windows для синхронизации используются самые разные объекты, применение которых существенно различается. Однако при рассмотрении синхронизации особое положение имеет момент перехода ожидаемого объекта в свободное состояние - с точки зрения ожидающего потока совершенно неважно, какие события привели к этому и какой именно объект стал свободным. Поэтому при большом разнообразии объектов, пригодных для синхронизации, существует всего несколько основных функций, осуществляющих ожидание объекта ядра:

```
DWORD WaitForSingleObject( HANDLE hHandle, DWORD dwMsecs );

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

С точки зрения операционной системы объекты ядра, поддерживающие интерфейс синхронизируемых объектов, могут находиться в одном из двух состояний: свободном (signaled) и занятом (nonsignaled). Функции проверяют состояние ожидаемого объекта или ожидаемых объектов и продолжают выполнение, только если объекты свободны. В зависимости от типа ожидаемого объекта, система может предпринять специальные действия (например, как только поток дожидается освобождения объекта исключительного владения, он сразу должен захватить его).

Функция WaitForSingleObject осуществляет ожидание одного объекта, а функция WaitForMultipleObjects может ожидать как освобождения любого из указанных объектов (bWaitAll = FALSE), так и всех сразу (bWaitAll = TRUE). Ожидание завершается либо по освобождении объекта(ов), либо по истечении указанного интервала времени (dwMsecs) в миллисекундах (бесконечное при dwMsecs = INFINITE). Код возврата функции позволяет определить причину - таймаут, освобождение конкретного объекта либо ошибка.

Если функция WaitForMultipleObjects (или ее клон) ожидает сразу все объекты группы, то до освобождения всех ожидаемых объектов одновременно никаких мер по занятию ранее освободившихся объектов функция не предпринимает.

В тех случаях, когда поток переходит в состояние ожидания, его исполнение блокируется до конца ожидания. Для реализации APC (и функций завершения ввода-вывода) необходимо было предусмотреть в операционной системе возможность приостановки потока с вызовом асинхронных процедур. Это связано с тем, что система должна гарантировать выполнение функций в контексте конкретного потока для соблюдения норм безопасности. Ожидание оповещения - это такое состояние ожидания, которое может быть завершено либо по достижении таймаута, либо при освобождении указанного объекта, либо после обработки APC. При этом в контексте потока, находящегося в состоянии ожидания оповещения, обрабатывается APC вызов и только затем завершается состояние ожидания.

Для перехода в ожидание оповещения предусмотрены функции SleepEx, WaitForMultipleObjectsEx, WaitForSingleObjectEx и SignalObjectAndWait.

Еще несколько функций предназначены для разработки GUI-приложений Win32: MsgWaitForMultipleObjects и MsgWaitForMultipleObjectsEx выполняют ожидание указанных объектов и, кроме того, могут контролировать состояние очереди потока, предназначенной для обработки оконных сообщений. Функция WaitForInputIdle ожидает, пока GUI-приложение не закончит свою инициализацию и не перейдет в ожидание в цикле обработки сообщений.

Несколько типов объектов в Win32 предназначены только для взаимной синхронизации потоков: события (event), семафоры (semaphore), объекты исключительного владения (мьютексы, mutex, mutual exclusion) и ожидающие таймеры (waitable timer). Для изменения их состояния предусмотрены специальные функции, так что потоки могут явным образом управлять состоянием таких объектов, обеспечивая взаимную синхронизацию. Эти объекты являются базовыми примитивами, на которых часто строятся более сложные синхронизирующие объекты.

При проектировании составных объектов иногда возникает задача изменения состояния одного из примитивных объектов при переходе к ожиданию другого. Если такие операции выполнять поочередно, то между вызовами функции, изменяющей состояние объекта, и функции ожидания возможно срабатывание планировщика и переключение потоков; то есть время, проходящее между изменением состояния одного объекта и началом ожидания другого, оказывается непредсказуемым. Чтобы избежать такой ситуации, предусмотрена функция SignalObjectAndWait, переводящая один объект в свободное состояние и ожидающая другой.

Стандартные синхронизирующие объекты могут быть именованными (функции, создающие эти объекты, имеют параметр, указывающий их имя), а могут быть "безымянными", если вместо имени указать NULL. Попытка создания именованных объектов с совпадающими именами приведет или к получению нового описателя существующего объекта (если типы существующего объекта и вновь создаваемого одинаковы), или к ошибке (если типы разные). Именованные объекты обычно используют для межпроцессного взаимодействия, а безымянные - для взаимодействия потоков в одном процессе.

События

Обычно событие - некоторый объект, который может находиться в одном из двух состояний: занятом или свободном. Переключение состояний осуществляется явным вызовом соответствующих функций; при этом любой процесс/поток, имеющий необходимые права доступа к объекту "событие", может изменить его состояние.

В Windows различают события с ручным и с автоматическим сбросом (тип и начальное состояние задаются при создании события функцией CreateEvent). События с ручным сбросом ведут себя обычным образом: функция SetEvent переводит событие в свободное (сигнальное) состояние, а функция ResetEvent - в занятое (не сигнальное). События с автоматическим сбросом переводятся в занятое состояние либо явным вызовом функции ResetEvent, либо ожидающей функцией WaitFor....

```
#define DEFAULT_SECURITY (LPSECURITY_ATTRIBUTES)NULL

VOID CALLBACK ApcProc( ULONG_PTR dwData )
{
  SetEvent( (HANDLE)dwData );
}
int main( void )
{
  HANDLE  hEvent;
  hEvent = CreateEvent( DEFAULT_SECURITY, TRUE, FALSE, NULL );
  QueueUserAPC(ApcProc, GetCurrentThread(), (ULONG_PTR)hEvent);
  WaitForSingleObject(hEvent,100);
    /* код завершения WAIT_TIMEOUT */
  SleepEx( 100, TRUE );
    /* APC процедура освобождает событие */
  WaitForSingleObject(hEvent,100);
    /* код завершения WAIT_OBJECT_0 */
  WaitForSingleObject(hEvent,100);
    /* код завершения WAIT_OBJECT_0 */
  CloseHandle( hEvent );
  return 0;
}
```

В примере создается событие с ручным сбросом в занятом состоянии. При первом вызове функции WaitForSingleObject событие все еще занято, поэтому выход из функции осуществляется по таймауту. При втором вызове уже успевает сработать APC и событие свободно - поэтому функция ожидания завершится с успехом. К третьему вызову состояние события не меняется, поэтому результат аналогичный. Если в приведенном примере изменить событие с ручным сбросом на событие с автоматическим сбросом (второй параметр функции CreateEvent должен быть FALSE), то при втором обращении к WaitForSingleObject функция завершится также с кодом WAIT_OBJECT_0, но при этом событие автоматически будет переведено в занятое состояние. В результате третий вызов функции WaitForSingleObject завершится по таймауту.

Поведение событий с ручным и автоматическим сбросом особенно различаются в случае, если несколько потоков ждут одного события: для события с ручным сбросом, при установке его в свободное состояние, выполнение могут продолжить все ожидающие потоки; а для события с автоматическим сбросом - только один, так как событие будет сразу переведено в занятое состояние. В некоторых случаях бывает надо перевести событие в свободное состояние, чтобы ожидающие на данный момент времени потоки могли продолжить свое выполнение, после чего снова вернуть в занятое. Вместо пары вызовов SetEvent...ResetEvent,

во время выполнения которых планировщик вполне может переключиться на другие потоки (в итоге время нахождения события в свободном состоянии непредсказуемо), целесообразно использовать функцию PulseEvent, которая как бы выполняет сброс и установку события, но в рамках одной операции.

Семафоры

Семафор представляет собой счетчик, который считается свободным, если значение счетчика больше нуля, и занятым при нулевом значении. При создании семафора задаются его максимально допустимое и начальное состояния. Ожидающие функции WaitFor... уменьшают значение свободного семафора на 1, если счетчик ненулевой, или переходят в режим ожидания до тех пор пока кто-либо не увеличит значение семафора. Увеличение счетчика осуществляется функцией ReleaseSemaphore:

```
#define DEFAULT_SECURITY (LPSECURITY_ATTRIBUTES)NULL

int main( void )
{
  HANDLE   hSem;
  LONG     lPrev;
  hSem = CreateSemaphore( DEFAULT_SECURITY, 0, 5, NULL );
  WaitForSingleObject( hSem, 100 );
    /* код завершения WAIT_TIMEOUT */
  ReleaseSemaphore( (HANDLE)dwData, 1, &lPrev );
  WaitForSingleObject( hSem, 100 );
    /* код завершения WAIT_OBJECT_0 */
  WaitForSingleObject( hSem, 100 );
    /* код завершения WAIT_TIMEOUT */
  ReleaseSemaphore( (HANDLE)dwData, 2, &lPrev );
  WaitForSingleObject( hSem, 100 );
    /* код завершения WAIT_OBJECT_0 */
  WaitForSingleObject( hSem, 100 );
    /* код завершения WAIT_OBJECT_0 */
  WaitForSingleObject( hSem, 100 );
    /* код завершения WAIT_TIMEOUT */
  CloseHandle( hSem );
  return 0;
}
```

В примере создается семафор в первоначально занятом состоянии, поэтому первое ожидание завершается по таймауту. После этого его счетчик увеличивается на 1, и следующее ожидание завершается успехом, при этом счетчик семафора уменьшается и он снова становится занятым. В результате третье ожидание завершается по таймауту. После этого счетчик увеличивается на 2, и два последующих ожидания завершаются успехом, после чего счетчик оказывается снова нулевым и третье ожидание завершается по таймауту.

К сожалению, средств для проверки текущего значения счетчика без изменения состояния семафора нет: функция ReleaseSemaphore позволяет узнать предыдущее значение, но при этом обязательно увеличит его значение хотя бы на 1 (попытка увеличить на 0 или на отрицательную величину рассматривается как ошибка), а ожидающая функция обязательно уменьшит счетчик, если семафор был свободен. Поэтому для определения значения счетчика надо использовать что-то вроде приведенного ниже примера:

```
LONG  lCounter;
lCounter = 0;
if ( WaitForSingleObject( hSem, 0 ) == WAIT_OBJECT_0 )
  ReleaseSemaphore( hSem, 1, &lCounter );
/* теперь переменная lCounter содержит значение счетчика */
```

Семафоры предназначены для ограничения числа потоков, имеющих одновременный доступ к какому-либо ресурсу.

Мьютексы

Объекты исключительного владения могут быть использованы в одно время не более чем одним потоком. В этом отношении мьютексы подобны критическим секциям, с той оговоркой, что работа с ними выполняется в режиме ядра (при использовании критических секций переход в режим ядра необязателен) и что мьютексы могут быть использованы для межпроцессного взаимодействия, тогда как критические секции реализованы для применения внутри процесса. Для захвата мьютекса используется ожидающая функция WaitFor..., а для освобождения - функция ReleaseMutex. При создании мьютекса функцией CreateMutex можно указать, чтобы он создавался сразу в занятом состоянии:

```
#include <process.h>
#include <windows.h>
#define DEFAULT_SECURITY  (LPSECURITY_ATTRIBUTES)NULL

unsigned __stdcall TProc( void *pdata )
{
  WaitForSingleObject( (HANDLE)pdata, 2000 );
  WaitForSingleObject( (HANDLE)pdata, 2000 );
  Sleep( 1000 );
  ReleaseMutex( (HANDLE)pdata );
  ReleaseMutex( (HANDLE)pdata );
  return 0;
}
int main( void )
{
  unsigned  id;
  HANDLE    hMutex, hThread;
  hMutex = CreateMutex( DEFAULT_SECURITY, TRUE, NULL );
  hThread = (HANDLE)_beginthreadex(
    (void*)0, 0, TProc, (void*)hMutex, 0, &id
  );
  Sleep( 1000 );
  ReleaseMutex( hMutex );
  WaitForSingleObject( hThread, INFINITE );
  CloseHandle( hThread );
  CloseHandle( hMutex );
  return 0;
}
```

Ожидающие таймеры

Эти объекты предназначены для выполнения операций через заданные промежутки времени или в заданное время. Таймеры бывают периодическими или однократными, также их разделяют на таймеры с ручным сбросом и синхронизирующие:

```
#include <windows.h>
#include <stdio.h>

int main()
{
  HANDLE            hTimer = NULL;
  LARGE_INTEGER     liDueTime;
  hTimer = CreateWaitableTimer(NULL, TRUE, "WaitableTimer");
  /* задать срабатывание через 5 секунд */
  liDueTime.QuadPart=-50000000;
  SetWaitableTimer( hTimer, &liDueTime, 0, NULL, NULL, 0 );
  WaitForSingleObject( hTimer, INFINITE );
  return 0;
}
```

Таймеры могут служить в качестве синхронизирующих объектов, как в данном примере, а могут вызывать указанную разработчиком функцию, если поток в нужное время находится в ожидании оповещения. Следует подчеркнуть, что ожидающие таймеры обладают ограниченной точностью работы. При необходимости точно планировать время выполнения (например, в случае обработки потоков мультимедиа данных) надо использовать специальный таймер, предназначенный для работы с мультимедиа (см. функции timeGetSystemTime, timeBeginPeriod и др.).

1.3 Реализация многопоточности средствами Win API

Создание процессов

Для создания процессов используются функции CreateProcess, CreateProcessAsUser, CreateProcessWithLogonW и CreateProcessWithTokenW. Функция CreateProcess создает новый процесс, который будет исполняться от имени текущего пользователя потока, вызвавшего эту функцию. Функция CreateProcessAsUser позволяет запустить процесс от имени другого пользователя, который идентифицируется его маркером безопасности (security token); однако вызвавший эту функцию поток должен принять меры к правильному использованию реестра, так как профиль нового пользователя не будет загружен. Функции CreateProcessWithTokenW и CreateProcessWithLogonW позволяют при необходимости загрузить профиль пользователя и, кроме того, функция CreateProcessWithLogonW сама получает маркер пользователя по известному учетному имени, домену и паролю.

```
BOOL CreateProcess(
  LPCTSTR lpApplicationName,    // имя исполняемого файла
  LPTSTR lpCommandLine,         // командная строка
  LPSECURITY_ATTRIBUTES  lpProcessAttributes,  // атрибуты доступа к процессу
  LPSECURITY_ATTRIBUTES lpThreadAttributes, // атрибуты доступа к потоку
  BOOL bInheritHandles,   // флаг наследования дескрипторов
  DWORD dwCreationFlags, // флаги создания и флаги класса приоритета
  LPVOID lpEnvironment,   // указатель на параметры настройки окружения
  LPCTSTR lpCurrentDirectory, // путь к текущему каталогу
```

```
  LPSTARTUPINFO lpStartupInfo,  // указатель на стуктуру отображения нового процесса
  LPPROCESS_INFORMATION lpProcessInformation  // указатель на структуру с информацией о созданном процессе
);

typedef struct _PROCESS_INFORMATION {
HANDLE hProcess;    // дескриптор нового процесса
HANDLE hThread;     // дескриптор первичного потока нового процесса
DWORD dwProcessId;  // идентификатор нового процесса
DWORD dwThreadId;   // идентификатор первичного потока нового процесса
} PROCESS_INFORMATION;

#include <windows.h>
#define DEFAULT_SECURITY  (LPSECURITY_ATTRIBUTES)NULL

int main( void )
{
  STARTUPINFO         si;
  PROCESS_INFORMATION  pi;
  memset( &si, 0, sizeof(si) );
  memset( &pi, 0, sizeof(pi) );
  si.cb = sizeof(si);
  CreateProcess(
    NULL, "cmd.exe", DEFAULT_SECURITY, DEFAULT_SECURITY,
    FALSE, NORMAL_PRIORITY_CLASS, NULL, NULL, &si, &pi
  );
  CloseHandle( pi.hThread );
  WaitForSingleObject( pi.hProcess, INFINITE );
  CloseHandle( pi.hProcess );
  return 0;
}
```

При создании процесса ему можно передать описатели каналов (CreatePipe), предназначенные для перенаправления stdin, stdout и stderr. Описатели каналов должны быть наследуемыми.

Для завершения процесса рекомендуется применять функцию ExitProcess, которая завершит процесс, сделавший этот вызов. В крайних случаях можно использовать функцию TerminateProcess, которая может завершить процесс, заданный его описателем. Этой функцией пользоваться не рекомендуется, так как при таком завершении разделяемые библиотеки будут удалены из адресного пространства уничтожаемого процесса без предварительных уведомлений - это может привести в некоторых случаях к утечке ресурсов.

Завершение процессов

```
VOID ExitProcess( UINT uExitCode);
BOOL TerminateProcess(HANDLE hProcess, UINT uExitCode);
```

Создание потоков

Каждый поток начинает выполнение с некоторой входной функции. В первичном потоке используется одна из функций: WinMain() или main(). Для создания вторичного потока необходимо определить его входную функцию:

```
DWORD WINAPI ThreadFunc ( PVoid pvParam)
{
   DWORD dwResult = 0;
   …
   Return dwResult
}
```

Для создания потока используется функция:

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttributes, // атрибуты доступа
  SIZE_T dwStackSize,                       // размер стека
  LPTHREAD_START_ROUTINE lpStartAddress,    // адрес функции потока
  LPVOID lpParameter,                       // параметр функции потока
  DWORD dwCreationFlags,                    // флаги потока
  LPDWORD lpThreadId                        // идентификатор потока
);
```

При необходимости можно создать поток в виртуальном адресном пространстве другого процесса. Для этого используется следующая функция:

```
HANDLE CreateRemoteThread(
  HANDLE hProcess,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  SIZE_T dwStackSize,
  LPTHREAD_START_ROUTINE lpStartAddress,
  LPVOID lpParameter,
  DWORD dwCreationFlags,
  LPDWORD lpThreadId
);
```

Пример:

```
#include <windows.h>
#include <strsafe.h>

#define MAX_THREADS 3
#define BUF_SIZE 255

typedef struct _MyData {
    int val1;
    int val2;
} MYDATA, *PMYDATA;

DWORD WINAPI ThreadProc( LPVOID lpParam )
{
    HANDLE hStdout;
    PMYDATA pData;
```

```
    TCHAR msgBuf[BUF_SIZE];
    size_t cchStringSize;
    DWORD dwChars;

    hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
    if( hStdout == INVALID_HANDLE_VALUE )
        return 1;

    // Cast the parameter to the correct data type.

    pData = (PMYDATA)lpParam;

    // Print the parameter values using thread-safe functions.

    StringCchPrintf(msgBuf,  BUF_SIZE,  TEXT("Parameters  =  %d,
%d\n"),
        pData->val1, pData->val2);
    StringCchLength(msgBuf, BUF_SIZE, &cchStringSize);
    WriteConsole(hStdout, msgBuf, cchStringSize, &dwChars, NULL);

    // Free the memory allocated by the caller for the thread
    // data structure.

    HeapFree(GetProcessHeap(), 0, pData);

    return 0;
}

void main()
{
    PMYDATA pData;
    DWORD dwThreadId[MAX_THREADS];
    HANDLE hThread[MAX_THREADS];
    int i;

    // Create MAX_THREADS worker threads.

    for( i=0; i<MAX_THREADS; i++ )
    {
        // Allocate memory for thread data.

        pData = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY,
                sizeof(MYDATA));

        if( pData == NULL )
            ExitProcess(2);

        // Generate unique data for each thread.

        pData->val1 = i;
        pData->val2 = i+100;

        hThread[i] = CreateThread(
```

```
            NULL,              // default security attributes
            0,                 // use default stack size
            ThreadProc,        // thread function
            pData,             // argument to thread function
            0,                 // use default creation flags
            &dwThreadId[i]);   // returns the thread identifier

        // Check the return value for success.

        if (hThread[i] == NULL)
        {
            ExitProcess(i);
        }
    }

    // Wait until all threads have terminated.

    WaitForMultipleObjects(MAX_THREADS, hThread, TRUE, INFINITE);

    // Close all thread handles upon completion.

    for(i=0; i<MAX_THREADS; i++)
    {
        CloseHandle(hThread[i]);
    }
}
```

Завершение потока

Завершение потока можно организовать четырьмя способами:

- Функция потока возвращает управление (предпочтительный способ)
- Поток самоуничтожается вызовом функции ExitThread
- Один из потоков данного или стороннего процесса вызывает функцию TerminateThread (нежелательный способ)
- Завершается процесс, содержащий данный поток

Проецирование файлов

Для управления адресным пространством предназначены функции VirtualAlloc, VirtualFree, VirtualAllocEx, VirtualFreeEx, VirtualLock и VirtualUnlock. С их помощью можно резервировать пространство в адресном пространстве процесса (без передачи физической памяти из файла) и управлять передачей памяти из файла подкачки страниц указанному диапазону адресов.

Механизмы проецирования файлов в память различаются для обычных и для исполняемых файлов. Функции создания процессов (CreateProcess...) и загрузки библиотек (LoadLibrary, LoadLibraryEx), помимо специфичных действий, выполняют проецирование исполняемых файлов в адресное пространство процесса; при этом учитывается их деление на сегменты, наличие секций импорта, экспорта и релокаций и др. Обычные файлы проецируются как непрерывный блок данных на непрерывный диапазон адресов. Для явного проецирования файлов используется специальный объект ядра проекция файла (file mapping object). Этот объект предназначен для описания файла, который может быть спроецирован в память, но

реального отображения файла или его части в память при создании проекции не происходит. Описатель объекта "проекция файла" можно получить с помощью функций CreateFileMapping и OpenFileMapping. Для проецирования файла или его части в память предназначены функции MapViewOfFile, MapViewOfFileEx и UnmapViewOfFile:

```
#include <windows.h>
#define DEFAULT_SECURITY  (LPSECURITY_ATTRIBUTES)NULL
int main( void )
{
  HANDLE  hFile, hMapping;
  LPVOID  pMapping;
  LPSTR   p;
  int     i;
  hFile = CreateFile(
    "abc.dat", GENERIC_WRITE|GENERIC_READ,
    FILE_SHARE_WRITE, DEFAULT_SECURITY,
    CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL
  );
  hMapping = CreateFileMapping(
    hFile, DEFAULT_SECURITY, PAGE_READWRITE, 0, 256, NULL
  );
  pMapping = MapViewOfFile( hMapping, FILE_MAP_WRITE, 0,0, 0 );
  for ( p = (LPSTR)pMapping, i=0; i<256; i++ ) *p++ = (char)i;
  UnmapViewOfFile( hMapping );
  CloseHandle( hMapping );
  CloseHandle( hFile );
  return 0;
}
```

Межпроцессное взаимодействие с использованием проецирования файлов

Проецирование файлов используется для создания разделяемой памяти: для этого один процесс должен создать объект "проекция файла", а другой - открыть его. После этого система будет гарантировать когерентность данных в этой проекции во всех процессах, которые ее используют, хотя проекции могут размещаться в разных диапазонах адресов.

В примере ниже приводится текст двух приложений (first.cpp и second.cpp), которые обмениваются между собой данными через общий объект "проекция файла":

```
/* FIRST.CPP */
#include <windows.h>
#define DEFAULT_SECURITY  (LPSECURITY_ATTRIBUTES)NULL
int main( void )
{
  HANDLE                hMapping;
  LPVOID                pMapping;
  STARTUPINFO             si;
  PROCESS_INFORMATION    pi;
  int                *p;
  int                i;
  memset( &si, 0, sizeof(si) );
  memset( &pi, 0, sizeof(pi) );
  si.cb = sizeof(si);
```

```
  hMapping = CreateFileMapping(
    INVALID_HANDLE_VALUE, DEFAULT_SECURITY, PAGE_READWRITE,
    0, 1024, "FileMap-AB-874436342"
  );
  pMapping = MapViewOfFile( hMapping, FILE_MAP_WRITE, 0,0, 0 );
  for (p=(int*)pMapping,i=0; i<256; i++) *p++=i;
  CreateProcess(
    NULL, "second.exe", DEFAULT_SECURITY,
    DEFAULT_SECURITY, FALSE, NORMAL_PRIORITY_CLASS,
    NULL, NULL, &si, &pi);
  CloseHandle( pi.hThread );
  WaitForSingleObject( pi.hProcess, INFINITE );
  CloseHandle( pi.hProcess );
  UnmapViewOfFile( hMapping );
  CloseHandle( hMapping );
  return 0;
}

/* SECOND.CPP */
#include <windows.h>
int main( int ac, char **av )
{
  HANDLE        hMapping;
  LPVOID        pMapping;
  int           *p;
  int           i;
  hMapping = OpenFileMapping(
    FILE_MAP_READ, FALSE, "FileMap-AB-874436342"
  );
  pMapping = MapViewOfFile( hMapping, FILE_MAP_READ, 0,0, 0 );
  for ( p = (int*)pMapping, i=0; i<256; i++ )
    if ( *p++ != i ) break;
  if ( i != 256 ) { /* ОШИБКА! */ }
  UnmapViewOfFile( hMapping );
  CloseHandle( hMapping );
  return 0;
}
```

Важно отметить, что если разные процессы откроют один и тот же файл, а затем каждый создаст свой собственный объект "проекция файла", то система не будет гарантировать когерентности данных в проекциях; когерентность обеспечивается только в рамках одного объекта "проекция файла". В некоторых случаях можно воспользоваться функцией FlushViewOfFile для явного сброса данных из оперативной памяти в файл, что, однако, еще не гарантирует автоматического обновления данных в других проекциях. Именно механизм проецирования файлов является базовым средством для передачи данных между адресными пространствами процессов. Он является основой для построения многих других средств межпроцессного взаимодействия. Так, например, обмен оконными сообщениями (если в сообщении содержится указатель на данные) между разными процессами приводит к тому, что система создает внутренний объект "проекция", помещает данные в него, проецирует на второй процесс и посылает сообщение получателю с указателем на проекцию в процессе-получателе.

Межпроцессное взаимодействие с использованием общих секций

Еще одна разновидность работы с разделяемыми данными посредством проецирования файлов связана с объявлением специальных разделяемых сегментов в приложении - такие сегменты будут общими для всех копий этого приложения. В своей основе такой способ является частным случаем проецирования - с той оговоркой, что проецируется исполняемый файл (или разделяемая библиотека) и управление проекциями осуществляется декларативным способом.

В Microsoft Visual C++ для объявления разделяемого сегмента и помещаемых в него данных используются директивы #pragma и _ _declspec, как показано в примере ниже:

```
/* FSEC.CPP */
#include <windows.h>
#define DEFAULT_SECURITY  (LPSECURITY_ATTRIBUTES)NULL

#pragma section("SHRD_DATA",read,write,shared)
_ _declspec(allocate("SHRD_DATA")) int shared[ 256 ];

int main( int ac, char **av )
{
  STARTUPINFO           si;
  PROCESS_INFORMATION   pi;
  int                   i;
  memset( &si, 0, sizeof(si) );
  memset( &pi, 0, sizeof(pi) );
  si.cb = sizeof(si);
  if ( ac != 2 || strcmp( av[1], "slave" ) ) {
    for ( i = 0; i < 256; i++ ) shared[i] = 256-i;
    /* первый экземляр с общими данными */
    CreateProcess(
      NULL, "fsec.exe slave", DEFAULT_SECURITY,
      DEFAULT_SECURITY, FALSE, NORMAL_PRIORITY_CLASS,
      NULL, NULL, &si, &pi
    );
    CloseHandle( pi.hThread );

    WaitForSingleObject( pi.hProcess, INFINITE );
    CloseHandle( pi.hProcess );
  } else {
    /* второй экземляр с общими данными */
    for ( i = 0; i < 256; i++ )
      if ( shared[i] != 256-i ) break;
    if ( i != 256 ) { /* ОШИБКА! */ }
  }
  return 0;
}
```

Некоторое неудобство, связанное с необходимостью использовать одно и то же приложение, можно легко обойти, если разделяемый сегмент поместить в разделяемую библиотеку - тогда один и тот же образ библиотеки может быть подключен к разным приложениям, что позволит им обмениваться данными. В этом варианте надо только учитывать, что адреса, в которые будет помещена библиотека, в разных процессах могут отличаться.

1.4 Создание интерфейса пользователя средствами Win API

Ядро Windows:

- **USER (16, 32) .dll** – функции ввода с клавиатуры мыши, ввод через интерфейс и т.д. (взаимодействие приложений с пользователями и средой Windows).
- **KERNEL (16, 32) .dll** – функции операционной системы (память, распределение системных ресурсов, загрузка).
- **GDI (16, 32) .dll** – графический интерфейс (функции создания и отображения графических объектов).

GUI (Graphics User Interface) – стандартный графический интерфейс пользователя. Это та часть Windows, которая обеспечивает поддержку аппаратно-независимой графики.

API (Application Program Interface) — интерфейс прикладных программ (набор функций, сосредоточенных в ядре Windows и дополнительных библиотеках).

DLL (Dynamic Link Libraries) — библиотека динамической компановки. Функции API содержатся в библиотеках динамической загрузки.

DDE – динамический обмен данными.

Создание простейшего окна.

Все приложения Windows должны содержать два основных элемента: функцию WinMain(...) и функцию окна WndProc.

Функция WinMain(...) служит точкой входа в приложение. Эта функция отвечает за следующие действия:

1. регистрацию типа класса окон приложения;
2. выполнение всех инициализирующих действий;
3. создание и инициализацию цикла сообщений приложения;
4. завершение программы (обычно при получении сообщения WM_QUIT).

Функция WndProc отвечает за обработку сообщений Windows. Эта часть программы является наиболее содержательной с точки зрения выполнения поставленных перед программой задач. Если мы хотим, чтобы программа обращала на наши действия внимание, то необходимо добавить ветки case для оператора switch в оконную процедуру WndProc. Например, если мы хотим, чтобы наше приложение обращало внимание на щелчок левой кнопкой мыши – добавляем ветку case WM_LBUTTONDOWN. В настоящий момент в оконной процедуре происходит только обработка сообщения WM_DESTROY. Больше Windows-окно пока ничего делать не умеет.

Заголовочный файл windows.h нужен для любой традиционной Windows программы на C. Именно в нем содержатся разные определения констант (WM_DESTROY и т. д.).

Параметры функции WinMain:

1. hInstance (тип HINSTANCE) – является идентификатором текущего экземпляра приложения. Данное число однозначно определяет программу, работающую под управлением Windows.

2. hPrevInstance (тип HINSTANCE) – указывал ранее (Windows 3.1) на предыдущий запущенный экземпляр приложения. В современных версиях Windows он равен NULL.
3. lpCmdLine – это указатель на строку, заканчивающуюся нулевым байтом. В этой строке содержатся аргументы командной строки приложения (как правило, содержит NULL).
4. nCmdShow – этот параметр принимает значение одной из системных констант, определяющих способ изображения окна (например, SW_SHOWNORMAL, SW_SHOWMAXIMIZED или SW_SHOWMINIMIZED).

Пример1:

```
#include <windows.h>
LONG WINAPI WndProc(HWND, UINT, WPARAM,LPARAM);
int  WINAPI  WinMain (HINSTANCE  hInstance,
                      HINSTANCE  hPrevInstance,
                      LPSTR    lpCmdLine,
                       int    nCmdShow)
{
  HWND hwnd;
  MSG msg;
  WNDCLASS w;
  memset(&w,0,sizeof(WNDCLASS));
  w.style = CS_HREDRAW | CS_VREDRAW;
  w.lpfnWndProc = WndProc;
  w.hInstance = hInstance;
  w.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
  w.lpszClassName = "My Class";
  RegisterClass(&w);
  hwnd = CreateWindow("My Class", "Окно пользователя",
WS_OVERLAPPEDWINDOW,500, 300, 500, 380, NULL, NULL, hInstance, NULL);
  ShowWindow(hwnd,nCmdShow);
  UpdateWindow(hwnd);
  while(GetMessage(&msg,NULL,0,0))
  {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }
  return msg.wParam;
}

LONG WINAPI WndProc(HWND hwnd, UINT Message, WPARAM wparam, LPARAM lparam)
{
  switch (Message)
  {
    case WM_DESTROY: PostQuitMessage(0); break;
    default: return DefWindowProc(hwnd, Message, wparam, lparam);
  }
  return 0;
}
```

Регистрация класса окна

Каждое окно, которое создается в рамках приложения Windows, должно основываться на классе окна – шаблоне, в котором определены выбранные пользователем стили, шрифты, заголовки и т.д. Для всех определений класса окна используется стандартный тип структуры. Таким образом, сначала определяется структура WNDCLASS w, а затем поля структуры заполняются информацией о классе окна. У этой структуры много полей, но

большинство из них можно определить нулем. В программе это делается строкой memset(&w,0,sizeof(WNDCLASS)) ;

Рассмотрим следующий фрагмент программы:

```
w.style = CS_HREDRAW | CS_VREDRAW;
w.lpfnWndProc = WndProc;
w.hInstance = hInstance;
w.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
w.lpszClassName = "My Class";
RegisterClass(&w);
```

В этом фрагменте определяется стиль класса (все идентификаторы стилей начинаются с префикса CS_). В программе значения поля стиля задаются константами. CS_HREDRAW ¬– обеспечивает перерисовку содержимого клиентской области окна при изменении размера окна по горизонтали, CS_VREDRAW – обеспечивает перерисовку содержимого клиентской области окна при изменении размера окна по вертикали.

w.lpfnWndProc = WndProc - значение указателя на функцию окна (WndProc), которая выполняет все задачи, связанные с окном.

w.hInstance = hInstance - определяется экземпляр приложения, регистрирующий класс окна.

w.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH) - определяется кисть, используемая для закраски фона окна.

w.lpszClassName = "My Class" - указатель на строку символов, заканчивающуюся на 0, которая определяет имя класса. Класс здесь – это не класс в смысле ООП (объектно-ориентированного программирования). Термин один, но смысла два. Исторически классы окон возникли раньше, чем классы ООП.

RegisterClass(&w) - регистрация происходит при помощи вызова функции RegisterClass().

Создание окна на основе класса окна

```
hwnd = CreateWindow("My Class","Окно пользователя",
WS_OVERLAPPEDWINDOW,500,300,500,380,NULL,NULL,hInstance,NULL);
```

Первый параметр функции служит для задания класса окна. Второй – это заголовок окна. Третий – определяет стиль окна (обычное перекрывающее окно с заголовком, кнопкой вызова системного меню, кнопками минимизации и максимизации и рамкой). Следующие шесть параметров определяют положение окна на экране (по оси X и по оси Y), размеры окна по оси X и по оси Y, идентификатор родительского окна и идентификатор меню окна. Следующее поле (hInstance) содержит идентификатор экземпляра программы, далее следует информация об отсутствии дополнительных параметров (NULL). Если создание окна прошло успешно, то функция CreateWindow(...) возвращает идентификатор созданного окна, в противном случае – NULL. После того как окно создано, его надо показать и обновить. Для того чтобы вывести главное окно приложения на экран, необходимо вызвать функцию Windows ShowWindow(...).

ShowWindow(hwnd,nCmdShow) выводит окно на экран. Параметр hwnd содержит идентификатор окна, созданного при вызове CreateWindow(...). Второй параметр определяет, как окно выводится в первый момент.

Последний шаг при выводе окна на экран заключается в вызове функции Windows UpdateWindow(hwnd) , которая приводит к перерисовке клиентской области окна.

Создание цикла обработки сообщений

Теперь программа готова выполнять свою главную задачу – обрабатывать сообщения. Используется стандартный цикл C/C++ цикл while:

```
 while(GetMessage(&msg,NULL,0,0))
{
  TranslateMessage(&msg);
  DispatchMessage(&msg);
}
```

Использование функции GetMessage

Вызов этой функции позволяет получить для обработки следующее сообщение из очереди сообщений приложения. Данная функция копирует сообщение в структуру сообщения, на которую указывает указатель msg, и передает его в основной блок программы. Если значение следующего параметра NULL, то будут поступать сообщения, относящиеся к любому окну приложения. Значения последних параметров (0,0) указывает функции, что не надо применять никаких фильтров сообщений. Фильтры могут применяться для распределения получаемых сообщений по категориям, например, нажатия на клавишу или перемещение мыши. После входа в цикл обработки сообщений, выйти из него можно, получив только одно сообщение: WM_QUIT. Когда обрабатывается сообщение WM_QUIT, то возвращается значение "ложь" и цикл обработки сообщений завершается.

Использование функции TranslateMessage

Функция TranslateMessage(...) преобразует сообщения виртуальных клавиш в сообщения о символах.

Использование функции DispatchMessage

Функция DispatchMessage(...) используется для распределения текущего сообщения соответствующей функции окна.

Оконная функция WndProc

Вторая часть в любой программе под Windows – это оконная процедура. В рассматриваемом примере она маленькая (так как программа ничего не делает), но, именно эта часть и является самой главной и интересной в приложении.

Рассмотрим функцию WndProc:

```
LONG WINAPI WndProc(HWND hwnd, UINT Message, WPARAM wparam, LPARAM
lparam)
{
```

```
 switch (Message)
 {
  case WM_DESTROY: PostQuitMessage(0); break;
  default: return DefWindowProc(hwnd, Message, wparam, lparam);
 }
 return 0;
}
```

Основное назначение оконной функции – это обработка сообщений Windows. Каждое приложение получает много сообщений. Их источник может быть разным. Например, сообщения от пользователя или от самой Windows. Обработка этих сообщений происходит именно в оконной функции. Это означает, что для каждого сообщения необходимо написать свой обработчик. Если обработчика не будет, то приложение не будет обращать внимание на сообщение. У оконной функции четыре параметра. Первый из них hwnd типа HWND задает окно, которое будет обрабатывать сообщение. Второй UINT Message – это передаваемое сообщение. Два последних WPARAM wparam, LPARAM lparam задают дополнительные параметры для передаваемого сообщения. Они для каждого сообщения свои. Оконная процедура отправляет сообщение в switch, который в примере имеет только один case:

```
switch (Message)
{
 case WM_DESTROY:
 ...
```

То есть, пока рассматриваемая программа обращает внимание только на сообщение WM_DESTROY. Это сообщение окно получает только при своем уничтожении. После принятия этого сообщения необходимо вызвать функцию PostQuitMessage(...). В ответ на сообщение WM_DESTROY необходимо поместить в очередь сообщение WM_QUIT. Это и делает функция PostQiutMessage(...), посылая это сообщение в очередь и говоря, что процесс должен быть завершен. Если мы хотим, чтобы наша программа реагировала еще на что-нибудь, то надо написать еще case.

Рассмотрим далее ветку default. В ней идет вызов функции DefWindowProc(hwnd, Message, wparam, lparam) . Основное предназначение этой функции – обработка сообщений Windows, которые не обрабатываются в нашей программе (то есть, для которых нет своего case). При этом ничего не делается, но очередь из сообщений идет.

Пример2: Создание сложного окна (Delphi)

```
program Project1;
uses Windows, Messages;

const
myClassName= 'myWindow';

var   handleWnd, Label1 : THandle;
      WndClass: TWndClass;
      Msg: TMsg;

function WindowProc(Window: HWnd; AMessage, WParam,
LParam: Longint): Longint; stdcall;
begin WindowProc:= DefWindowProc(Window, AMessage, WParam, LParam);
      case AMessage of
```

```
      {WM_COMMAND: if lParam = Button1 then
      MessageBox( 0, 'Вы нажали кнопку!', 'Информация',
      MB_OK or MB_ICONINFORMATION); }
      WM_DESTROY: Halt;
      end;
end;

begin with WndClass do
      begin
      hInstance := hInstance;
      lpszClassName:= myClassName;
      style := cs_hRedraw or cs_vRedraw;
      hbrBackground:= color_btnface +1;
      lpfnWndProc := @WindowProc;
      hCursor := LoadCursor(0, idc_Arrow);
      hIcon := LoadIcon(0, IDI_EXCLAMATION);
      lpszMenuName := NIL;
      cbWndExtra := 0;
      cbClsExtra := 0;
      end;

      RegisterClass( WndClass );
      handleWnd:=     CreateWindow(myClassName,     'Нажми     кнопку',
ws_OverlappedWindow,
      400, 300, 200, 100, 0, 0, hInstance , NIL);
      if handleWnd = 0 then
      begin
      MessageBox( 0, 'Error', NIL, MB_OK );
      Exit;
      end;
      Label1:= CreateWindow( 'Label', 'Text',
      WS_VISIBLE or WS_CHILD or WM_SETTEXT,
      20, 10, 60, 23, handleWnd, 0, hInstance, nil);
      ShowWindow(handleWnd, SW_SHOW);
      UpdateWindow(handleWnd);
      while GetMessage(Msg, handleWnd, 0, 0) do
      begin
      TranslateMessage(Msg) ;
      DispatchMessage(Msg) ;
      end;
end.
```

Пример разложения в ряд функции. Графический вывод

Проиллюстрируем рассмотренный в предыдущем параграфе теоретичекий материал на примере написания Windows-приложения решения задачи о разложении в ряд некоторой функции.

Задача: Разложить в ряд Тейлора в окрестности точки 0 функции sin(x), cos(x) и вывести в окне график функции .

Напишем функции вычисления синуса и косинуса – разложения в ряд Тейлора в окрестности точки 0. Разложения в ряд имеют вид:

$$sin(x) = x - \frac{x^3}{3!} + \ldots + (-1)^{n-1} \cdot \frac{x^{2n-1}}{(2n-1)!} cos(x) = 1 - \frac{x^2}{2} + \ldots + (-1)^n \cdot \frac{x^{2n}}{(2n)!}$$

Радиус сходимости для этих рядов – ∞.

```
const double epsilon = 0.0000001;
const double pi = 3.14159265;
const double x_start = -2*pi;
const double x_end = 2*pi;

double sin_e(double arg, double eps)
{
  double result = 0, rn = arg;
  for(int i = 1; ;i++)
  {
    if(rn < eps && rn > -eps) return result;
    result += rn;
    rn = -rn*arg*arg/(i+1)/(i+2);
    i++;
  }
}

double cos_e(double arg, double eps)
{
  double result = 0, rn = 1;
  for(int i = 0; ;i++)
  {
    if(rn < eps && rn > -eps) return result;
    result += rn;
    rn = -rn*arg*arg/(i+1)/(i+2);
    i++;
  }
}
```

Для дальнейшего решения задачи – вывода графика функции в клиентском окне, будем обрабатывать сообщения, которые будут посылаться окну. Когда необходима перерисовка окна (изменились размеры окна, часть окна перекрылась другим окном и т.п.), система посылает окну сообщение WM_PAINT.

Добавим функции, которые будут выводить в клиентскую часть окна график и координатные оси, а также, необходимые ветви case в операторе switch.

Для построения изображения используем API функции операционной системы: MoveToEx(...), LineTo(...), GetClientRect(...) и т.д. Информацию об этих функциях них можно получить в справочной системе.

```
void DrawAxis(HDC hdc, RECT rectClient)
{
 HPEN penGraph = CreatePen(PS_SOLID,2,RGB(0,0,255));
 HGDIOBJ gdiOld = SelectObject(hdc, penGraph);
 MoveToEx(hdc, 0, rectClient.bottom/2, NULL);
 LineTo(hdc, rectClient.right, rectClient.bottom/2);
 LineTo(hdc, rectClient.right - 5, rectClient.bottom/2 + 2);
 MoveToEx(hdc, rectClient.right, rectClient.bottom/2, NULL);
 LineTo(hdc, rectClient.right - 5, rectClient.bottom/2 - 2);
 MoveToEx(hdc, rectClient.right/2, rectClient.bottom, NULL);
 LineTo(hdc, rectClient.right/2, rectClient.top);
 LineTo(hdc, rectClient.right/2 - 2, rectClient.top + 5);
 MoveToEx(hdc, rectClient.right/2, rectClient.top, NULL);
 LineTo(hdc, rectClient.right/2 + 2, rectClient.top + 5);
```

```
  SelectObject(hdc, gdiOld);
}

void DrawGraph(HDC hdc, RECT rectClient)
{
  HPEN penGraph = CreatePen(PS_SOLID,2,RGB(255,0,0));
  HGDIOBJ gdiOld = SelectObject(hdc, penGraph);
  double x_current = x_start;
  double step = (x_end - x_start)/rectClient.right;
  double y_start = cos_e(x_start, epsilon) + cos_e(x_start, epsilon);
  MoveToEx(hdc, 0, int(-y_start/step) + rectClient.bottom/2, NULL);
  while(x_current < x_end)
  {
    x_current += step;
    double y_next = cos_e(x_current, epsilon) + cos_e(x_current, epsilon);
    LineTo(hdc, int(x_current/step) + rectClient.right/2, int(-y_next/step) + rectClient.bottom/2);
  }
  SelectObject(hdc, gdiOld);
}

void OnPaint(HWND hwnd)
{
  PAINTSTRUCT ps;
  RECT rectClient;
  HDC hdc = BeginPaint(hwnd,&ps);
  GetClientRect(hwnd, &rectClient);
  DrawAxis(hdc, rectClient);
  DrawGraph(hdc, rectClient);
  ValidateRect(hwnd,NULL);
  EndPaint(hwnd,&ps);
}

LONG WINAPI WndProc(HWND hwnd, UINT Message, WPARAM wparam, LPARAM
lparam)
{
  switch (Message)
  {
    case WM_PAINT:
      OnPaint(hwnd);
    break;
    case WM_MOUSEMOVE:
      SetCapture(hwnd);
      SetCursor(LoadCursor(NULL,IDC_ARROW));
      ReleaseCapture();
    break;
    case WM_WINDOWPOSCHANGED:
      UpdateWindow(hwnd);
    break;
    case WM_DESTROY: PostQuitMessage(0); break;
    default: return DefWindowProc(hwnd, Message, wparam, lparam);
  }
  UpdateWindow(hwnd);
```

```
  return 0;
}
```
Листинг 3.1.

Функция DrawGraph(...) выводит на экран график, функция DrawAxis(...) выводит координатные оси. При выводе графика функции на экран используем преобразование системы координат окна приложения в логическую систему координат и наоборот. Подробно этот вопрос рассмотрен в следующей лекции. В результате в клиентской области окна получим искомый график функции :рис. 3.1

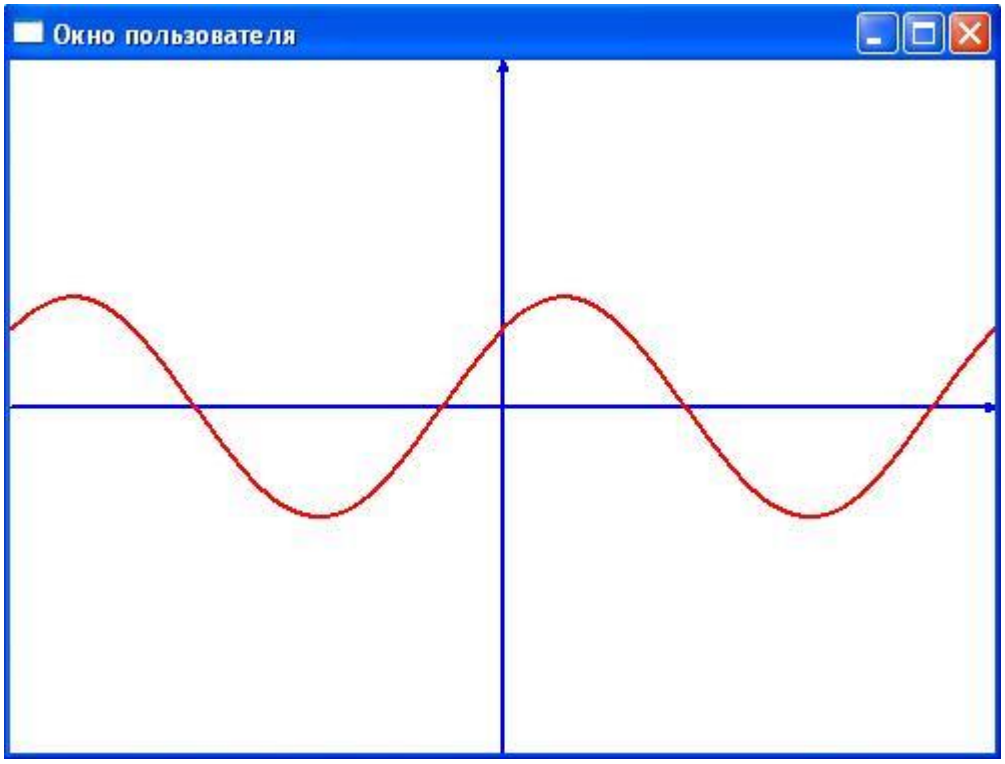

Рис. 3.1. Построение графика функции

2. Индивидуальные задания

Разработать однопоточное и многопоточное приложение, вычисляющее параллельно с заданной точностью объём или площадь поверхности фигуры вращения относительно оси OX на заданном отрезке. На экране отобразить график заданной функции.

Исходных данные:

- отрезок [A; B];
- точность вычислений;
- количество используемых потоков (от 1 до 10).

Сравнить результаты многопоточного решения с однопоточным решением, построить графики зависимости времени решения от точности для линейного и трёхпоточного решения и для точности 0.00001 график зависимости времени решения от числа используемых потоков. Сделать выводы.

Для синхронизации потоков использовать указанные средства синхронизации, для вычисления площади поверхности или объёма использовать указанный метод согласно варианта задания (таблица 3.1).

Интерфейс, создание и синхронизацию потоков реализовать средствами Win API.

Таблица 3.1 Варианты заданий

Вариант	f(x)	Что вычислять	Средство синхронизации потоков	Метод вычисления
1	$f(x)=e^{-x}\sin x$	площадь	Атомарные операции	Центральные прямоугольники
2	$f(x)=e^{-x}\cos x$	объём	Мьютексы	Левые прямоугольники
3	$f(x)=e^{\cos x}\sin x$	площадь	Семафоры	Правые прямоугольники
4	$f(x)=e^{\sin x}\cos x$	объём	Атомарные операции	Метод Симпсона
5	$f(x)=e^{x}arctgx$	площадь	Семафоры	Метод Монте-Карло
6	$f(x)=e^{-x}arctgx$	объём	Атомарные операции	Метод 3/8 Симпсона
7	$f(x)=x^{3}\cos x$	площадь	Критические секции	Двухточечная квадратура Гаусса-Лежандра
8	$f(x)=x^{3}\sin x$	объём	Семафоры	Трёхточечная квадратура Гаусса-Лежандра
9	$f(x)=x^{3}2^{\sqrt[3]{x\sin x\cos x}}$	площадь	Мьютексы	Центральные прямоугольники
10	$f(x)=x^{2}2^{\sqrt[3]{x\sin x\cos x}}$	объём	Атомарные операции	Левые прямоугольники
11	$f(x)=x^{2}2^{\sqrt[3]{x^{2}\sin x\cos x}}$	площадь	Атомарные операции	Правые прямоугольники
12	$f(x)=x^{2}e^{\cos x}$	объём	Мьютексы	Метод Симпсона
13	$f(x)=x^{3}e^{\sin x}$	площадь	Мьютексы	Метод Монте-Карло
14	$f(x)=x^{3}e^{\cos x}$	объём	Мьютексы	Метод 3/8 Симпсона
15	$f(x)=x^{2}e^{\sin x}$	площадь	События	Двухточечная квадратура Гаусса-Лежандра
16	$f(x)=2^{x}e^{\sin x}$	объём	Атомарные операции	Трёхточечная квадратура Гаусса-Лежандра
17	$f(x)=2^{x\sqrt{e^{\sin x}}}$	площадь	События	Центральные прямоугольники
18	$f(x)=2^{x\sqrt{e^{\sin x}}}$	объём	Семафоры	Левые прямоугольники
19	$f(x)=2^{x\sqrt{e^{\sin x\cos x}}}$	площадь	События	Правые прямоугольники

20	$f(x) = x^2 e^{\sin x \cos x}$	объём	Критические секции	Метод Симпсона
21	$f(x) = x^3 e^{\sin x \cos x}$	площадь	Атомарные операции	Метод Монте-Карло
22	$f(x) = x^{e^{\sin x \cos x}}$	объём	Мьютексы	Метод 3/8 Симпсона
23	$f(x) = \frac{e^{x \sin x}}{x^2 + 1}$	площадь	Семафоры	Двухточечная квадратура Гаусса-Лежандра
24	$f(x) = 2^x e^{\cos x}$	площадь	Мьютексы	Трёхточечная квадратура Гаусса-Лежандра
25	$f(x) = x^3 2^{\sqrt[3]{x \sin x}}$	объём	Мьютексы	Центральные прямоугольники
26	$f(x) = x^2 2^{\sqrt[3]{x \sin x}}$	объём	Атомарные операции	Левые прямоугольники
27	$f(x) = x^3 2^{\sqrt[3]{x \cos x}}$	площадь	Критические секции	Правые прямоугольники
28	$f(x) = x^3 2^{\sqrt[3]{x \sin x}}$	объём	Семафоры	Метод Симпсона
29	$f(x) = \frac{2^{x \cos x \sin x}}{\sin^2 x + \cos^2 x}$	площадь	События	Метод Монте-Карло
30	$f(x) = \frac{e^{x \cos x}}{x^2 + 1}$	объём	Атомарные операции	Метод 3/8 Симпсона

Вопросы к защите

1. Процессы и потоки в ОС Windows
2. Создание процессов и потоков в Win API
3. Создание и управление интерфейсом средствами Win API
4. Синхронизация потоков в ОС Windows
5. Синхронизация процессов в ОС Windows
6. Атомарные операции
7. Семафоры
8. Критические секции
9. События
10. Уметь внести исправления в программу для реализации любого алгоритма из вопросов 6-9