Параллельное программирование: WinAPI и OpenMP

ЛЕКЦИЯ 7

Калинина А.П.

Литература

1. И. Одинцов Профессиональное программирование. Системный подход. – «БХВ-Петербург» - 2004. – 610 с.

2. Джин Бэкон, Тим Харрис Операционные системы. Параллельные и распределенные системы. – bhv «Питер» - 2004 – 799 с.

3. Материалы тренинга Intel для преподавателей, апрель 2006

Два стиля параллельного программирования

Существуют два стиля параллельного программирования:

- Модули пассивны, а потоки могут вызывать процедуры для выполнения кода модулей
  - Пример: WinAPI
- Модули активны и содержат постоянные, заранее определенные процессы
  - Пример: OpenMP

Необходимость изучения Windows Threads

- Это «родные» потоки Windows, надстройкой над которыми является стандарт OpenMP
- В OpenMP используется параллелелизм «вилочного» типа: в начале параллельной секции (участок кода) потоки одновременно начинают работу, выход из параллельной секции требует завершения работы всех потоков.
- Параллелелизм «вилочного» типа – существенное ограничение на выбор задач, производительность вычислений в которых от распараллеливания подобного типа возрастет
- Организация более дифференцированного подхода к режиму различных потоков требует другой технологии, например, WinAPI
- С другой стороны, методом сравнения OpenMP и WinAPI, можно выделить свойства, общие для технологий многопоточного программирования

Содержание по WINAPI

- Функции Win32 Threading API, применяющиеся для
  - Создания потоков
  - Уничтожения потоков
- Синхронизации доступа к разделяемым переменным
- Простые модели для программирования координации действий потоков

Win32* «HANDLE» – тип данных для обращения к любому объекту Windows

- К каждому объекту в Windows можно обратиться с помощью переменной типа «**HANDLE**»
  - Указатель на объекты ядра
  - Потоки, процессы, файлы, события, мьютексы, семафоры, и т.д.

Функция создания объекта возвращает «**HANDLE**»

- Управление объектом можно осуществлять через его «HANDLE»
- Напрямую обращаться к объектам нельзя

Создание потока Win32*

```
HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES ThreadAttributes,
    DWORD StackSize,
    LPTHREAD_START_ROUTINE StartAddress,
    LPVOID Parameter,
    DWORD CreationFlags,
    LPDWORD ThreadID );
```

CreateThread(): ее предназначение

- На предыдущем слайде показан заголовок функции CreateThread (), которая создает Win32 поток
- Этот поток начинает выполнение функции, описываемой третьим и четвертым параметрами
- Данная функция возвращает «HANDLE», который используется для обращения к ее потоку

CreateThread(): первый параметр – «атрибуты безопасности» `ThreadAttributes`

- Каждый объект ядра имеет атрибуты безопасности
- Первый параметр в CreateThread() позволяет программисту определить атрибуты безопасности для потока
- Система защиты объектов Windows определяет совокупности процессов с разрешенным или запрещенным доступом к данным объектам
- Значение «NULL» устанавливает значения атрибутов безопасности «по умолчанию»

CreateThread(): второй параметр –объем стека потока - `Stacksize`

- Параметр **`Stacksize`** позволяет пользователю определить размер стека потока
- Значение ‘0’ позволяет установить объем стека «по умолчанию», равное одному мегабайту

CreateThread(): третий параметр – имя функции, с выполнения которой поток начнет работу - `StartAddress`

- Третий параметр, **`StartAddress`** – это имя функции
- В дальнейшем эту функцию будем называть «функция потока», «потоковая функция», «функция для многопоточного выполнения»
- С выполнения функции с этим именем поток и начнет свою работу
- Это функция с глобальной видимостью, объявляемая как DWORD WINAPI.

CreateThread(): четвертый параметр - `Parameter`

- Потоковой функции требуется только один параметр типа LPVOID («указатель на VOID»). Значение этого параметра для потока может быть установлено с помощью четвертого параметра.

- Если потоковой функции требуется больше, чем одно значение, можно инкапсулировать их в одну структуру, которую и передать в качестве четвертого параметра.

- При этом самым первым действием, выполненным в потоковой функции, должна быть декомпозиция этой структуры на отдельные компонентные части.

CreateThread(): пятый параметр – «режим старта» `CreationFlags`

- CreationFlags позволяет определить «режим старта» потока, который создан, но выполнение которого «приостановлено».

- «По умолчанию» (для этого нужно установить значение параметра, равное ‘0’) работа потока начинается сразу, как только он создается системой.

CreateThread(): шестой параметр – `ThreadId`

- **`ThreadId`** – параметр «уникальности», который обеспечивает то, что каждое потоковое задание выполняется своим потоком

- может использоваться повторно до тех пор, пока данный поток существует.

Если создать поток не удалось…

- Если выполнение **`CreateThread()`** не завершилось созданием потока, будет возвращено «FALSE»

- Причина «неудачи» может быть установлена с помощью вызова **`GetLastError()`**.

- **`GetLastError()`**- РЕКОМЕНДУЕТСЯ ВЫПОЛНЯТЬ ВСЕГДА ДЛЯ ЛЮБОГО ОШИБОЧНОГО КОДА

Альтернативы CreateThread() – меньше преимуществ…

- Существуют альтернативные функции создания потоков с помощью Microsoft C library.
- Это функции “**_beginthread**” и “**_beginthreadex**”.
- “**_beginthread**” лучше не применять, так как
  - Не включает в себя «атрибуты безопасности»
  - Не включает в себя «флаги» установки «режима старта»
  - Не возвращает идентификатор (номер) потока
- “**_beginthreadex**” обладает теми же аргументами, что и **CreateThread**.
- В MSDN – дополнительная информация.

LPTHREAD_START_ROUTINE `StartAddress` – третий параметр в `CreateThread()` – подробнее...

CreateThread() ожидает указателя на глобальную функцию

- Тип возвращаемого этой функцией значения DWORD
- Вызывает стандартные WINAPI
- Функция обладает единственным формальным параметром типа LPVOID (void *) – «указатель на void» - четвертый параметр в CreateThread()

```
DWORD WINAPI MyThreadStart(LPVOID p);
```

Поток начинает работу с выполнения этой функции

Чтобы применить явные потоковые функции, необходимо…

- Выделить участки кода, которые требуется выполнять параллельно
- Инкапсулировать выделенный код в потоковую функцию

  - Если код, предназначенный для распараллеливания, уже является функцией, то управление выполнением этой функции с помощью входных параметров должно быть переписано таким образом, чтобы координировать работу нескольких потоков.

- Добавить вызов CreateThread, чтобы сделать выполнение этой функции многопоточным

Уничтожение потоков

- Необходимо освободить ресурсы операционной системы
  - Потоковые «HANDLEs» оставляют занятой зарезервированную память
  - Если потоки закончили работу, необходимо освободить ресурсы до того, как программа завершит свою работу
  - Непрерывное создание новых потоков без освобождения ресурсов потоков, которые выполнили свою работу, приведет к «утечке памяти»
- Завершение процесса сделает это «за Вас»

```
BOOL CloseHandle(HANDLE hObject);
```

Пример: создание потока

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

DWORD WINAPI helloFunc(LPVOID arg ) {
    printf("Hello Thread\n");
    return 0;
}

main() {
    HANDLE hThread =
        CreateThread(NULL, 0, helloFunc,
                     NULL, 0, NULL );
    }
```


Что будет?

Что будет…

Реализуется одна из двух возможностей:

1) Сообщение “Hello Thread” появится на экране
2) На экране не появится ничего. Это гораздо вероятнее, чем первая возможность.

- В главном потоке осуществляется управление ресурсами всего процесса (выполняющийся экземпляр программы), а когда главный поток закончит свою работу (а значит, процесс завершится), все потоки будут уничтожены.
- Таким образом, если выполнение вызова CreateThread завершится до того, как операционная система создаст потоки и начнет выполнение их заданий, все потоки будут преждевременно уничтожены в связи с завершением процесса

Что делать, чтобы потоки выполнили задание…

Чтобы избежать создания приложений, которые только

- «плодят потоки»,
- а эти потоки уничтожаются до того, как они совершат хоть какую-то полезную работу,

необходимо применить какой-нибудь механизм, который не позволит процессу завершится, пока потоки не выполнят свою работу

Может, подождать в цикле, пока каждый поток выполнит свою работу?...

```
#include <stdio.h>
#include <windows.h>
BOOL threadDone = FALSE ;

DWORD WINAPI helloFunc(LPVOID arg ) {
    printf("Hello Thread\n");
    threadDone = TRUE ;
    return 0;
}
main() {
    HANDLE hThread = CreateThread(NULL, 0, helloFunc,
                                  NULL, 0, NULL );
  while (!threadDone);//"потраченные зря" циклы процессора
}
```


Это не лучший выход!

Почему не лучший выход – слишком «дорого»...

- Конечно, в этом случае поток не будет уничтожен до того, как выполнит всю свою работу, но...
  - Пока сообщение не напечатано (а это событие труднопредсказуемое, зависящее от случайных факторов), главный поток находится в состоянии ожидания, постоянно и непрерывно проверяя в цикле, не изменилось ли значение threadDone в результате создания потока.
  - Однако, если выполнение программы осуществляется на однопроцессорной машине, или машине с гипертредингом, главный поток выполнит тысячи или даже миллионы циклов процессора к тому времени, как будут потоки будут созданы и выполнят свои задания
- Итак, вполне очевидно – этот выход из положения не лучший

Как «подождать поток»

Ожидание одного объекта (потока)

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

Осуществляет «ожидание потока» (блокирование) до тех пор, пока

- Закончится временной промежуток
- Существует поток (handle не станет signaled)
  - Используется **INFINITE** («бесконечное») ожидание, пока поток не будет уничтожен

~~Не требует циклов CPU~~

Это уже лучше… Сначала – немного о “handle” -ax

Один поток ждет завершения работы другого потока

Любой «HANDLE» может быть в одном из двух состояний:

- Сигнализирующем (signaled)
- Не сигнализирующем (non-signaled)

«HANDLE» потока находится в состоянии «сигнализирует» (signaled), если он завершил работу, и «не сигнализирует» (non-signaled) в противном случае.

Это уже лучше...Немного о `WaitForSingleObject`

- WaitForSingleObject будет блокировать завершение других потоков, пока данный поток не закончит свою работу (handle is signaled);
- Второй параметр – это временной предел для ожидания.
- Если время ожидания превышено, код становится доступен для выполнения другими потоками, независимо от того, в каком состоянии находится HANDLE.
- Чтобы установить условием окончания ожидания завершение работы потока (handle signaled), нужно установить время ожидания «бесконечность» INFINITE (определенное значение константы).
- Любое другое значение временного предела приведет к ошибке. Нужно проверить код ошибки, чтобы установить причину завершения работы функции

- WaitForSingleObject может быть применен не только для потока, но и для «события», «мьютекса» и т.д.

Ждать, пока все не закончат....

Количество ожидаемых объектов (потоков) не более 64

```
DWORD WaitForMultipleObjects(
    DWORD nCount,
    CONST HANDLE *lpHandles, // array
    BOOL fWaitAll, // wait for one or all
    DWORD dwMilliseconds)
```

Дождаться всех: **fWaitAll==TRUE**

Дождаться хоть одного: **fWaitAll==FALSE**

- Завершает работу, если хотя бы один поток свое задание выполнил

Комментарии к `WaitForMultipleObjects`

- **`nCount`** - количество ожидаемых «HANDLES» из всего массива HANDLEs. Должно быть nCount <= 64.
- Эти **`nCount`** элементов начинают свою работу последовательно с адреса **`lpHandles до lpHandles[nCount-1]`**.
- **`fWaitAll`** определяет, ждать ли всех **`nCount`** объектов или одного из них. Если **TRUE,** то **`WaitForMultipleObjects`** «ждет всех», иначе «ждет одного».
- Четвертый параметр такой же, как в **`WaitForSingleObject`**.

Комментарии к WaitFor* функциям

Параметром является “HANDLE”

В качестве “HANDLE” могут быть рассмотрены различные типы объектов

Эти объекты могут быть в двух состояниях

- «Сигнализирует» (Signaled) («свободен» - перевод из [2])
- «Не сигнализирует» (Non-signaled) («занят» - перевод из [2])

Смысл понятия «signaled» или «non-signaled» зависит от типа объекта

WaitFor* функции заставляют ждать объекты, которые находятся в состоянии “signaled”

Выполнение функции определяется типом объекта, описываемого “HANDLE”

- Поток: «сигнализирующий» (signaled) означает «завершивший работу» (здесь: всю, которая была до «WaitFor*»)

Объекты ядра (диспетчерские объекты для планирования работы потоков) Windows 2000 [2]

- поток ядра
- мьютекс ядра
- мутант ядра
- событие ядра
- пара событий ядра
- семафор ядра
- таймер ядра

Состояние объектов синхронизации Widows 2000 [2]

Тип объекта	Устанавливается в состояние *«свободен» (signaled),* когда...	Воздействие на ожидающие потоки
Процесс	Завершается выполнение последнего потока	Все освобождаются
Поток	Завершается выполнение потока	Все освобождаются
Событие	Поток устанавливает событие	Все освобождаются
Пара событий	Выделенный поток клиента или сервера устанавливает событие	Освобождается другой выделенный поток
Семафор	Счетчик семафора доходит до нуля	Все освобождаются
Таймер	Наступает заданное время или истекает временной интервал	Все освобождаются
Мутант	Поток освобождает мутант	Освобождается один поток

Задание 1 - напечатать “HelloThreads”

Использовать предыдущий пример для вывода сообщения

- “Hello Thread” – от каждого потока
- Каждому потоку – сообщить свой номер
  - Применить цикл for для создания потоков **(CreateThread)**

Должно быть напечатано

```
Hello from Thread #0
Hello from Thread #1
Hello from Thread #2
Hello from Thread #3
```

Пример ошибки

- Ошибка: значение i будет различным для различных потоков
- К тому моменту, когда потоки начнут выполнять задание, i будет другим, чем когда выполнялся вызов CallThread

```
DWORD WINAPI threadFunc(LPVOID pArg) {
  int* p = (int*)pArg;
  int myNum = *p;
  printf( "Thread number %d\n", myNum);
}
. . .
// from main():
for (int i = 0; i < numThreads; i++) {
  hThread[i] =
     CreateThread(NULL, 0, threadFunc, &i, 0, NULL);
}
```

Временная диаграмма Hello Threads

Это один из вариантов – наиболее вероятный: оба потока напечатают номер 2. Но, может быть, первый поток напечатает 1, а второй – 2 – это менее вероятно.

Этот тип ошибки называется *«гонки данных»* или *“data race”* – более, чем один поток имеют доступ к одной и той же переменной

Время	*Главный поток*	*Поток 0*	*Поток 1*
T_0	i = 0	---	----
T_1	Создать (&i)	---	---
T_2	i++ (i == 1)	начинает	---
T_3	Создать (&i)	p = pArg	---
T_4	i++ (i == 2)	myNum = *p myNum = 2	начинает
T_5	ожидает	print(2)	p = pArg
T_6	ожидает	exit	myNum = *p myNum = 2

Условия возникновения гонки данных (Race Conditions)

Одновременный доступ к одной переменной для всех потоков

- Конфликты чтение-запись (Read/Write conflict)
- Конфликты запись-запись (Write/Write conflict)

Наиболее частая ошибка

Не всегда легко обнаружить

Как избежать гонки данных

Использовать переменные, являющиеся локальными для каждого потока

- Описывать переменные в пределах потоковой функции
- Память – резервировать в стеке потока(Allocate on thread’s stack)
- Запоминать для потока (TLS (Thread Local Storage))

Управлять общим доступом к критическим участкам

- Доступ «одного» и синхронизация
- «Замки», семафоры, события, критические секции, мьютексы...(Lock, semaphore, event, critical section, mutex)

Решение – “Local” Storage

Каждый поток запоминает то значение i, которое было в момент его создания - **tNum[i]**

```
DWORD WINAPI threadFunc(LPVOID pArg)
{
  int myNum = *((int*)pArg);
  printf( “Thread number %d\n”, myNum);
}
. . .

// from main():
for (int i = 0; i < numThreads; i++) {
  tNum[i] = i;
  hThread[i] =
     CreateThread(NULL, 0, threadFunc, &tNum[i],
                  0, NULL);
}
```

Мьютекс [2]

Mutual exclusion – взаимное исключение

- Синхронизационный объект, используемый несколькими потоками для обеспечения целостности общего ресурса (как правило, данных) путем взаимоисключающего доступа
- Ресурс, защищенный *мьютексом*, доступен в настоящий момент времени только одному процессу
- Перед тем, как обратиться к такому ресурсу, процесс блокирует его мьютекс, а закончив с ним работать, снимает блокировку
- Если мьютекс уже заблокирован другим потоком, то запросивший блокировку поток может либо дождаться освобождения ресурса, либо остаться в активном состоянии и перейти к другим операциям в зависимости от того, какая из процедур была вызвана для блокирования мьютекса

Мьютекс (Win32* Mutexes)

- Объект ядра, **`CreateMutex(…)`** возвращает ”HANDLE” мьютекса
- Используется «**`WaitForSingleObject`**» для «закрытия на замок» или «блокирования» мьютекса
- Если мьютекс не заблокирован, “HANDLE” мьютекса в состоянии «сигнализирует» (“Signaled” или «свободен» [2]),
- При выполнении функции «**`WaitForSingleObject`**» , мьютекс оказывается захваченным одним из потоков и переходит в состояние “non-signaled” («занят» [2])
- Блокировка мьютекса снимается с помощью операции **`ReleaseMutex(…)`**

```
CreateMutex(…) //создать новый мьютекс
WaitForSingleObject // «ждать и не подпускать» (wait & lock)
ReleaseMutex(…) // освободить (unlock)
```

Используется для координации действий множества процессов

Критическая секция (Win32* Critical Section) – действия в “main”

«Легковесный мьютекс», но только внутри одного процесса

Очень популярная и часто применяемая конструкция

Новый тип данных

```
CRITICAL_SECTION cs;
```

Операторы создания и уничтожения – в главной программе

```
•InitializeCriticalSection(&cs)
• DeleteCriticalSection(&cs);
```

Критическая секция – действия в потоковой функции

До «защищаемого кода»

```
EnterCriticalSection(&cs)
```

- Блокирует работу других потоков, если уже есть поток в критической секции
- Разрешает работу «кому-нибудь», если «никого» в критической секции нет

«После» защищаемого кода

```
LeaveCriticalSection(&cs)
```

Пример критической секции : генерация случайных чисел – “main”

```
InitializeCriticalSection(&g_cs);
for (int i = 0; i < numThreads; i++)
{
 tNum[i] = i;
 hThread[i] =
 CreateThread(NULL, 0, threadFunc, &tNum[i], 0, NULL);
}
WaitForMultipleObjects(numThreads, hThread, TRUE, INFINITE);

DeleteCriticalSection(&g_cs);
```

Критическая секция – в потоковой функции – генерация случайных чисел

```
for( int i = start_local; i <= finish_local; i+=2 )
{
  if( TestForPrime(i) )
   {
     EnterCriticalSection(&g_cs);
      globalPrimes[gPrimesFound++] = i;
         LeaveCriticalSection(&g_cs);
    }
 }
```

Семафоры [1]

- *Семафор* – это защищенная переменная, значение которой можно запрашивать и менять только при помощи специальных операций P и V и при инициализации.
- Концепция семафоров была предложена Дейкстрой в начале 60гг 20 века. Применяют три основные типа семафоров:

  - *Двоичные* (бинарные) семафоры, принимающие только два значения {0,1}
  - *Считающие* семафоры. Их значения – целые неотрицательные числа
  - *Общие* семафоры. Принимают все множество целых чисел.

Семафоры - Win32* Semaphores

Объекты синхронизации, использующие счетчик

Этот счетчик представляет число доступных ресурсов

- Ввел и сформулировал Edsger Dijkstra (1968)

Две операции для семафора

- Ждать [P(s)]:
  - Поток ждет, пока s > 0, при этом s уменьшается s = s - 1
- Одному продолжить [V(s)]: s = s + 1

Семафор «свободен» (signaled) (объекты синхронизации Windows 2000, [2]), когда «показания счетчика» доходят до нуля

Семафор “is in signaled state” («свободен», по переводу [2]), если s>0 ([3], Win32*)

Создание семафора (Win32* Semaphore)

```
HANDLE CreateSemaphore(
    LPSECURITY_ATTRIBUTES lpEventAttributes,
    LONG lSemInitial,  //Начальное значение счетчика
    LONG lSemMax,      //Максимальное значение
счетчика
    LPCSTR lpSemName); //
```

Значение **lSemMax** должно быть больше или равно 1

Значение **lSemInitial** должно быть

- Больше или равно 0,
- Меньше или равно **lSemMax**, **и**
- Не может выйти за границы диапазона

Операции ждать – продолжить (Wait and Post)

WaitForSingleObject («ожидание одного») – на семафоре

- Пока не будет == 0, поток ждет
- Уменьшает счетчик на 1, пока он положителен

Увеличение переменной семафора – операция «продолжить» (Post operation)

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

- Увеличивает переменную счетчика посредством **cReleaseCount**
- возвращает предыдущее значение через **lpPreviousCount**

Предназначение семафоров

Управлять доступом к структурам данных конечного размера

- Queues, stacks, deques
- Счетчик применяется для нумерации доступных элементов

Управлять доступом к конечному числу ресурсов

- File descriptors, tape drives…

Контролировать число активных потоков в области

Бинарный семафор [0,1] может работать как мьютекс

«Минусы семафоров» Semaphore

«Потеря владельца»

Любой поток может освободить семафор раньше, чем другой начнет его «ждать»

- Хорошая практика – избегать семафоров

No concept of abandoned semaphore

- If thread terminates before post, semaphore increment may be lost
- Зависание

Бинарный семафор как мьютекс: аналогия с кабинетом врача: свободно - занято

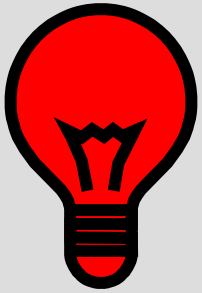
Сценарий:
Врач может принять только одного больного (один поток может выполнять защищенный код). Очередь больных ждет за дверью.

Начало работы, никого нет, кабинет свободен,

горит зеленая лампочка

Бинарный семафор инициализируется «1»:

```
hSem1 = CreateSemaphore(NULL, 1, 1, NULL);
```



Когда в кабинет заходит больной (один из потоков устанавливает “0” и начинает выполнять защищенный код), загорается красная лампочка, больной обслуживается, очередь ждет за дверью

Значение семафора устанавливается в «0»,

```
WaitForSingleObject(hSem1, INFINITE);
```

Бинарный семафор как мьютекс: аналогия с кабинетом врача: свободно - занято

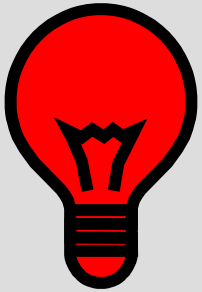
«Продолжение приема у врача» - «семафор» регулирует прием у врача (выполнение потоками защищенного кода)

Прием больного завершен (поток выполнил защищенный код, установил “1”), он вышел из кабинета, врач может принять следующего больного, загорелась зеленая лампочка

- выполняется

```
ReleaseSemaphore(hSem1, 1, NULL);
```

значение семафора снова устанавливается в «1»



В кабинет заходит больной (следующий поток установил “0” и приступил к выполнению защищенного кода), загорается красная лампочка, новый больной обслуживается, очередь ждет за дверью

Значение семафора устанавливается в «0»,

```
WaitForSingleObject(hSem1, INFINITE);
```

Пример: бинарный семафор как мьютекс

Генерация простых чисел:

Семафор используется для контроля доступа к записи простых чисел в общую переменную – вместо критической секции

Генерация простых чисел – “main”

```
HANDLE hSem1;
```

```
hSem1 = CreateSemaphore(NULL, 1, 1, NULL);// Binary
semaphore
   for (int i = 0; i < numThreads; i++)
   {
   tNum[i] = i;
   hThread[i] = CreateThread(NULL, 0, threadFunc,
                    &tNum[i], 0, NULL);
   }
   WaitForMultipleObjects(numThreads, hThread,
      TRUE, INFINITE);
```

Генерация простых чисел: потоковая функция

```
for( int i = start_local; i <= finish_local; i+=2 )
{
 if( TestForPrime(i) )
 {
   WaitForSingleObject(hSem1, INFINITE);
globalPrimes[gPrimesFound++] = i;
   ReleaseSemaphore(hSem1, 1, NULL);
  }
}
```

Генерация простых чисел: потоковая функция с критической секцией для последующего сравнения с OPenMP – (распределение работы между потоками то же, что и «для семафора»)

```
for( int i = start_local; i <= finish_local; i+=2 )
{
  if( TestForPrime(i) )
   {
     EnterCriticalSection(&g_cs);
      globalPrimes[gPrimesFound++] = i;
       LeaveCriticalSection(&g_cs);
    }
 }
```

Генерация простых чисел: потоковая функция – распределение работы между потоками (WINAPI)

```
start_number = 3;

int kvant = 8;

finish_number = 200000;

int start_local, finish_local;

start_local = start_number + myNum*kvant;

int i_kvant = (finish_number - start_number)/ numThreads;

int one = kvant*numThreads;

int j_limit = 1 + (finish_number - start_number)/one;
```

Генерация простых чисел: потоковая функция – распределение работы между потоками (WINAPI, продолжение)

```
for(int j=1; j<=j_limit; j++)
{
 finish_local = start_local + kvant - 1;
 if (finish_local>finish_number)
       finish_local = finish_number ;
 if(start_local <= finish_number)
  for( int i = start_local; i <= finish_local; i+=2 )
          {...}
 }
```

Генерация простых чисел: OpenMP– но это не полная аналогия WINAPI! – ЗАТО МЕНЬШЕ КОДА!

И задачу создания потоков (главная программа WinAPI), и распределение заданий для потоков (потоковые функции) выполняет одна прагма

#pragma omp parallel for schedule(static, 8)

```
#pragma omp parallel for schedule(static, 8)
{
    for( int i = start; i <= end; i += 2 ){
        if( TestForPrime(i) )
#pragma omp critical
            globalPrimes[gPrimesFound++] = i; }
                                            }
```

Задание

1. Изучить примеры реализации критической секции и бинарного семафора WinAPI на примере программ генерации простых чисел, присоединенных к лекции №6
2. Получить ускорение параллельной программы для всех примеров и сравнить со случаем OpenMP(проект программы присоединен к лекции №4)
3. Создать вариант программы «Преобразование Фурье» (проект присоединен к лекции №5) на основе Windows Threads