# Лабораторная работа № 2. Критические секции и объект CRITICAL\_SECTION

**Критическая секция** – это область программного кода, в которой процесс обращается к разделяемому ресурсу, и его действия не могут быть прерваны.

## Использование объекта CRITICAL\_SECTION

Для организации критических секций потоков может использоваться объект типа CRITICAL\_SECTION, который позволяет приостановить выполнение потоков на входе в критическую секцию, если какой-либо поток уже вошел в свою критическую секцию, связанную с тем же критическим ресурсом. Сразу стоит заметить, что синхронизация работы потоков с использованием переменных типа CRITICAL\_SECTION возможна только в рамках одного процесса – синхронизировать действия потоков разных процессов с помощью этого метода нельзя.

Для работы с критической секцией необходимо описать глобальную переменную типа CRITICAL\_SECTION:

CRITICAL\_SECTION <имя критической секции>;

Если в программе имеется несколько разделяемых ресурсов, то для каждого из них можно описать свою критическую секцию.

Инициализация критической секции выполняется функцией **InitializeCriticalSection**:

void InitializeCriticalSection(LPCRITICAL\_SECTION lpCriticalSection);

В качестве параметра данной функции передается указатель на переменную критической секции типа CRITICAL\_SECTION.

После использования критической секции ее можно уничтожить функцией **DeleteCriticalSection**:

void DeleteCriticalSection(LPCRITICAL\_SECTION lpCriticalSection);

Перед входом в критическую секцию поток обязан выполнить функцию **EnterCriticalSection**:

void EnterCriticalSection(LPCRITICAL\_SECTION lpCriticalSection);

Если критическая секция в данный момент занята другим потоком, то поток окажется в состоянии ожидания, пока критическая секция не будет освобождена.

Попытка входа в критическую секцию без блокировки потока может быть осуществлена функцией **TryEnterCriticalSection**:

BOOL TryEnterCriticalSection(LPCRITICAL\_SECTION lpCriticalSection);

Если критическая секция на момент выполнения этой функции свободна, то поток может ее занять. При этом функция вернет значение true. В противном случае функция возвращает значение false, но поток не попадает в состояние ожидания. Обработка неудачной попытки входа в критическую секцию целиком и полностью лежит на программисте.

По завершению действий критической секции поток обязан освободить ее при помощи функции **LeaveCriticalSection**:

void *LeaveCriticalSection*(LPCRITICAL\_SECTION lpCriticalSection);

После выполнения этой функции критическая секция будет свободна, и один из потоков, ожидающих на ее входе, сможет ее занять.

## Пример программы

***Задача***. В системе существует *n* потоков, каждый из которых генерирует несколько случайных чисел. Необходимо найти сумму этих чисел.

***Решение***. Решение задачи довольно просто. В ней имеется только один критический ресурс – переменная, в которой накапливается сумма чисел, поэтому достаточно реализовать только один объект критической секции. Потоки должны блокировать вход в критическую секцию, когда обращаются к разделяемой переменной, и освобождать критическую секцию, когда работа с этой переменной закончена.

#include <windows.h>

#include <iostream>

#include <time.h>

CRITICAL\_SECTION cs;

int sum = 0;

// функция потока

DWORD WINAPI func(LPVOID param)

{

// количество чисел, генерируемых потоком

const int m = 10;

int a;

for (int i = 0; i>m; i++)

{

a = rand(); // генерация числа

EnterCriticalSection(&cs); // Вход в критическую секцию

sum += a;

LeaveCriticalSection(&cs); // Выход из критической секции

}

return 0;

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

const int n = 10;

HANDLE hThread[n]; //массив потоков

DWORD dwThreadID[n]; //массив идентификаторов

InitializeCriticalSection(&cs); // Инициализация критической секции

sum = 0;

srand((unsigned int) time(NULL));

// Запуск потоков

for (int i = 0; i<n; i++)

{

hThread[i] = CreateThread(NULL,0,func,NULL,0,&(dwThreadID[i]));

if (hThread[i] == NULL)

{

std::cout << "Ошибка создания потока " << GetLastError() << '\n';

}

}

// Ожидание завершения потоков

WaitForMultipleObjects(n,hThread,true,INFINITE);

//Вывод результатов

std::cout << "Сумма = " << sum << '\n';

// Закрытие потоков

for (int i = 0; i<n; i++)

{

CloseHandle(hThread[i]);

}

return 0;

}

## Задания

1. Необходимо сформировать матрицу вещественных чисел, содержащую *n* строк и *m* столбцов, а затем найти ее сумму. Формирование строк матрицы должно выполняться параллельно *n* потоками. Подсчет суммы элементов также должен выполняться *n* параллельными потоками.

*Поток подсчета суммы по строке может быть запущен только тогда, когда поток формирования строки матрицы закончит свою работу. Для повышения надежности синхронизации рекомендуется потоки подсчета сумм запустить в приостановленном состоянии.*

1. Дан текст длиной *n* символов. Необходимо найти контрольную сумму по данному тексту в виде суммы кодов символов по модулю *256*. Для нахождения суммы запустите *k* потоков, где *k < n*. Каждый *i*-й поток, *i = 0, 1,…,k-1* должен обрабатывать только символы с номерами *i + k \*s*, где *s = 0, 1, 2,...* – шаг работы потока.
2. Дана функция *y = f(x)*, принимающая на отрезке *[a,b]* только положительные значения. Необходимо найти приближенное значение площади криволинейной трапеции, ограниченной кривой *y = f(x)*, прямыми *x = a, x = b* и осью абсцисс, разбив отрезок *[a,b]* на *n* элементов. Для подсчета площади каждого элемента криволинейной трапеции запустить отдельный поток, вычисляющий площадь этого элемента трапеции как площадь криволинейной трапеции, ограниченной кривой *y = f(x)*, прямыми *x = xi, x = xi+1*, где *i* – номер элемента, и осью абсцисс, методом прямоугольников, разбивая отрезок *[xi, xi+1]* на *m* участков.

*Пояснение: делим исходную “фигуру” на n частей, каждую из этих n частей делим на m частей. Каждую из m частей считать методом прямоугольников.*