**Системное программное обеспечение**

**Курсовая работа**

Съестов Дмитрий Вячеславович

P3217

Задание: написать программу, состоящую из основного потока и потоков А, В и С.

Массив М содержит 10 млн элементов. Потоки А и В копируют его и сортируют локальную копию, записывая результат в массив М1 через каждые 100 тыс. итераций.

* Поток А использует метод Шелла с шагами 105, 104, 103, 102, 10, 1.

Идея метода состоит в сортировке элементов, находящихся на определённом расстоянии друг от друга.

Сложность – в худшем случае O(n2), в лучшем O(n \* log2n)

* Поток В использует метод слияния.

Метод заключается в разбиении массива на участки, которые сортируются рекурсивно или непосредственно, если они достаточно малы.

Сложность – O(n \* logn)

* Поток С выполняет поиск случайно выбранного элемента в массиве М1 при каждом его обновлении.

Для синхронизации потоков используются четыре метода:

* **Семафор**: объект, ограничивающий число потоков, которые могут войти в участок кода
* **Мьютекс**: аналогичен одноместному семафору, но является объектом ядра и может быть освобождён только тем потоком, который владеет им.
* **Критическая секция:** в C# критические секции реализованы через оператор lock, блокирующий объект-заглушку. Связаны с процессом, в отличие от мьютексов.
* **Событие:** объект, оповещающий ожидающие потоки (которых может быть несколько) о возникновении некого события. Как и мьютекс, могут быть использованы несколькими процессами.

**Базовый класс**

internal abstract class ThreadRunner

{

public int[] M;

public volatile int[] M1;

//Элемент для поиска

protected int searchElement;

//Количество итераций между обновлениями

public readonly int ItersToUpdate;

protected readonly Thread ThreadA;

protected readonly Thread ThreadB;

protected readonly Thread ThreadC;

public bool ShellFinished;

public bool MergeFinished;

protected AutoResetEvent[] AutoEvents;

protected AutoResetEvent SearchEvent;

protected ThreadRunner(int[] m, int x)

{

M = m;

searchElement = x;

ItersToUpdate = m.Length / 100;

M1 = new int[m.Length];

ShellFinished = false;

MergeFinished = false;

AutoEvents = new AutoResetEvent[3];

for (int i = 0; i < 3; i++)

AutoEvents[i] = new AutoResetEvent(false);

ThreadA = new Thread(ThreadAProc);

ThreadB = new Thread(ThreadBProc);

ThreadC = new Thread(ThreadCProc);

ThreadA.Name = “Поток А”;

ThreadB.Name = «Поток В»;

ThreadC.Name = «Поток С»;

SearchEvent = new AutoResetEvent(false);

}

public TimeSpan Start()

{

var startTime = DateTime.Now;

ThreadA.Start();

ThreadB.Start();

ThreadC.Start();

WaitHandle.WaitAll(AutoEvents);

return DateTime.Now – startTime;

}

public abstract void UpdateArray(ref int counter, int[] m1);

private void ThreadAProc()

{

Sorter.ShellSort(M, this);

AutoEvents[0].Set();

}

private void ThreadBProc()

{

Sorter.MergeSort(M, this);

AutoEvents[1].Set();

}

private void ThreadCProc()

{

while (!ShellFinished && !MergeFinished)

{

if (!SearchEvent.WaitOne(500)) continue;

int index = Array.IndexOf(M1, searchElement);

Console.WriteLine($”Элемент {searchElement} {(index == -1 ? «не найден» : $»найден на позиции {index}»)}.»);

}

AutoEvents[2].Set();

}

}

**Семафоры, мьютексы и события**

/// <summary>

/// Запускает потоки А, В и С, используя для синхронизации производный от WaitHandle класс.

/// </summary>

internal class WaitHandleThreadRunner : ThreadRunner

{

private readonly WaitHandle waitHandle;

public WaitHandleThreadRunner(int[] m, int x, WaitHandle waitHandle) : base(m, x)

{

this.waitHandle = waitHandle;

}

private void Exit()

{

switch (waitHandle)

{

case Semaphore s:

s.Release(1);

break;

case Mutex m:

m.ReleaseMutex();

break;

case EventWaitHandle e:

e.Set();

break;

}

}

public override void UpdateArray(ref int counter, int[] m1)

{

counter++;

if (counter < ItersToUpdate) return;

counter = 0;

waitHandle.WaitOne();

Console.WriteLine(Thread.CurrentThread.Name + “ обновляет массив…”);

Array.Copy(m1, M1, m1.Length);

Exit();

SearchEvent.Set();

}

**Критические секции**

/// <summary>

/// Запускает потоки А, В и С, используя для синхронизации критическую секцию.

/// </summary>

class LockThreadRunner : ThreadRunner

{

private readonly object locker;

public LockThreadRunner(int[] m, int x) : base(m, x)

{

locker = new object();

}

public override void UpdateArray(ref int counter, int[] m1)

{

counter++;

if (counter < ItersToUpdate) return;

counter = 0;

lock(locker)

{

Console.WriteLine(Thread.CurrentThread.Name + “ обновляет массив…”);

Array.Copy(m1, M1, m1.Length);

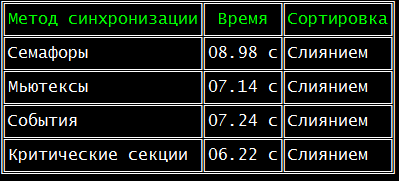
}

SearchEvent.Set();

}

}

**Результаты**

****

Каждый из этих методов имеет свои преимущества:

* Семафоры могут быть многоместными.
* Мьютексы могут использоваться несколькими процессами.
* События могут оповещать несколько ожидающих потоков.
* Критические секции работают быстрее всего. Также в C# это самый удобный способ, потому что поддерживается на уровне языковых конструкций.

**Циклическая блокировка**

Попробуем ещё один метод блокировки – спинлок. Он представляет собой флаг и холостой цикл, ожидающий его установки. В библиотеке .NET существует структура для эффективной реализации циклической блокировки – SpinWait.

internal class SpinThreadRunner : ThreadRunner

{

private volatile bool locked;

public SpinThreadRunner(int[] m, int x) : base(m, x)

{

locked = false;

}

public override void UpdateArray(ref int counter, int[] m1)

{

counter++;

if (counter < ItersToUpdate) return;

counter = 0;

var spinWait = new SpinWait();

while (locked)

{

Thread.MemoryBarrier();

spinWait.SpinOnce();

}

locked = true;

Console.WriteLine(Thread.CurrentThread.Name + " обновляет массив...");

Array.Copy(m1, M1, m1.Length);

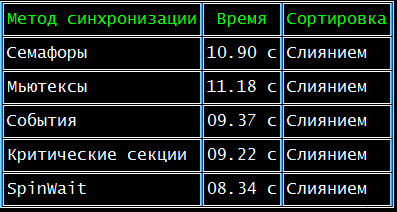
locked = false;

SearchEvent.Set();

}

}

**Результат**



Как видно, спинлок оказался быстрее критических секций.

Особенности этого метода:

+ Оптимизация. Модификация переменных намного быстрее переключения контекста процессором, особенно когда блокировка происходит очень часто.

+ В C# SpinWait – структура, а следовательно, циклическая блокировка не создаёт нагрузки на сборщик мусора.

- В зависимости от реализации циклическая блокировка может сильно нагружать процессор, если длится долго.