1. **Синхронизация потоков с использованием мьютексов: алгоритм использования, пример программной реализации**

Инструмент управления синхронизацией потоков представляет класс Mutex или мьютекс.

int x = 0;

Mutex mutexObj = new();

// запускаем пять потоков

for (int i = 1; i < 6; i++)

{

    Thread myThread = new(Print);

    myThread.Name = $"Поток {i}";

    myThread.Start();

}

void Print()

{

    mutexObj.WaitOne();     // приостанавливаем поток до получения мьютекса

    x = 1;

for (int i = 1; i < 6; i++)

    {

        Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");

x++;

        Thread.Sleep(100);

    }

    mutexObj.ReleaseMutex();    // освобождаем мьютекс

}

1. **Синхронизация потоков с использованием мониторов: описание методов, при-мер программной реализации.**

int x = 0;

object locker = new();  // объект-заглушка

// запускаем пять потоков

for (int i = 1; i < 6; i++)

{

    Thread myThread = new(Print);

    myThread.Name = $"Поток {i}";

    myThread.Start();

}

void Print()

{

    bool acquiredLock = false;

    Monitor.Enter(locker, ref acquiredLock);

x = 1;

    for (int i = 1; i < 6; i++)

    {

        Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");

x ++;

        Thread.Sleep(100);

    }

    if (acquiredLock) Monitor.Exit(locker);

}

1. **Синхронизация потоков с использованием семафоров: описание методов, при-мер программной реализации.**

Семафоры позволяют ограничить количество потоков, которые имеют доступ к определенным ресурсам. В .NET семафоры представлены классом **Semaphore**.

Например, у нас такая задача: есть некоторое число читателей, которые приходят в библиотеку три раза в день и что-то там читают. И пусть у нас будет ограничение, что единовременно в библиотеке не может находиться больше трех читателей. Данную задачу очень легко решить с помощью семафоров:

// запускаем пять потоков

for (int i = 1; i < 6; i++)

{

    Reader reader = new Reader(i);

}

class Reader

{

    // создаем семафор

    static Semaphore sem = new Semaphore(3, 3);

    Thread myThread;

    int count = 5;// счетчик чтения

    public Reader(int i)

    {

        myThread = new Thread(Read);

        myThread.Name = $"Читатель {i}";

        myThread.Start();

    }

    public void Read()

    {

        while (count > 0)

        {

            sem.WaitOne();  // ожидаем, когда освободиться место

            Console.WriteLine($"{Thread.CurrentThread.Name} входит в библиотеку");

            Console.WriteLine($"{Thread.CurrentThread.Name} читает");

            Thread.Sleep(1000);

            Console.WriteLine($"{Thread.CurrentThread.Name} покидает библиотеку");

            sem.Release();  // освобождаем место

            count--;

            Thread.Sleep(1000);

        }

    }

}

1. **Механизмы синхронизации потоков. Оператор lock (замок). Пример про-граммной реализации.**

У нас запускаются пять потоков, которые вызывают метод Print и которые работают с общей переменной x. И мы предполагаем, что метод выведет все значения x от 1 до 5. И так для каждого потока. Однако в реальности в процессе работы будет происходить переключение между потоками, и значение переменной x становится непредсказуемым. Например, в моем случае я получил следующий консольный вывод (он может в каждом конкретном случае различаться): 1 1 1 1 1 6 7 7 9 11 14 11 16…

int x = 0;

object locker = new();  // объект-заглушка

// запускаем пять потоков

for (int i = 1; i < 6; i++)

{

    Thread myThread = new(Print);

    myThread.Name = $"Поток {i}";

    myThread.Start();

}

void Print()

{

    lock (locker)

    {

        x = 1;

        for (int i = 1; i < 6; i++)

        {

            Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");

            x++;

            Thread.Sleep(100);

        }

    }

}

Вывод с применением lock: 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 …

1. **Библиотека параллельных задач TPL: основные методы, программная реализа-ция на примере расчета математической константы.**

метод Start() собственно запускает задачу.

new Task(() => Console.WriteLine("Hello Task!")).Start();

Второй способ заключается в использовании статического метода Task.Factory.StartNew(). Этот метод также в качестве параметра принимает делегат Action, который указывает, какое действие будет выполняться. При этом этот метод сразу же запускает задачу

Task task = Task.Factory.StartNew(() => Console.WriteLine("Hello Task!"));

Пример расчета числа пи:

public static void Main()

{

var iterations = int.Parse(Console.ReadLine() ?? "100");

Console.WriteLine(CalculatePi(iterations));

}

private static double CalculatePi(int iterations)

{

var cpus = Environment.ProcessorCount;

var tasks = new Task<double>[cpus];

for (var i = 0; i < tasks.Length; i++)

{

var startInclusive = (int)Math.Round((double)i \* iterations / cpus);

var endExclusive = (int)Math.Round((double)(i + 1) \* iterations / cpus);

tasks[i] = Task.Run(() => CalculatePi(startInclusive, endExclusive));

}

return tasks.Sum(t => t.Result);

}

private static double CalculatePi(int startInclusive, int endExclusive)

{

var sum1 = 0.0;

for (var i = startInclusive; i < endExclusive; i += 2)

{

sum1 += 1.0 / (2 \* i + 1);

}

var sum2 = 0.0;

for (var i = startInclusive + 1; i < endExclusive; i += 2)

{

sum2 -= 1.0 / (2 \* i + 1);

}

return 4\*(sum1 + sum2);

}

1. **Производительность вычислений: закон Гроша, гипотеза Минского, закон Мура, закон Амдала.**

**Закон Гроша**

Производительность компьютера увеличивается как квадрат стоимости. Если компьютер A стоит в два раза дороже, чем компьютер B, то вы должны ожидать, что компьютер A в четыре раза быстрее, чем компьютер B.

**Гипотеза Минского**

**Минский** сформулировал **гипотезу**: производительность параллельной системы растёт (примерно) пропорционально логарифму числа процессоров.

**Закон Мура**

Количество транзисторов, размещаемых на кристалле интегральной схемы, удваивается каждые 24 месяца.

**Закон Амдала**

Суммарное время выполнения задачи на параллельной системе не может быть меньше времени выполнения самого медленного фрагмента.

Ускорение процесса вычислений при использовании *N* процессоров ограничивается величиной:

 где  – доля последовательных вычислений.