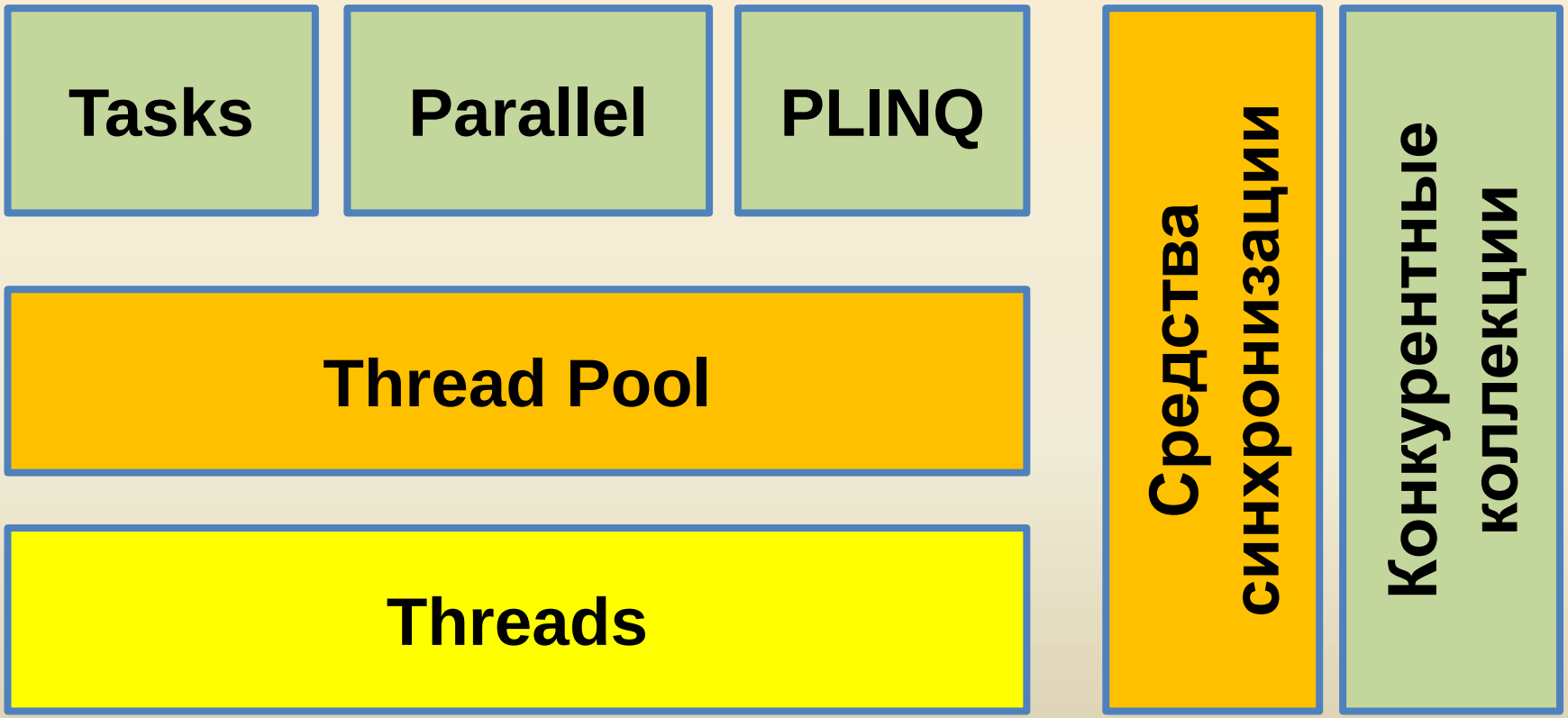
Среда Framework .NET
для параллельного программирования



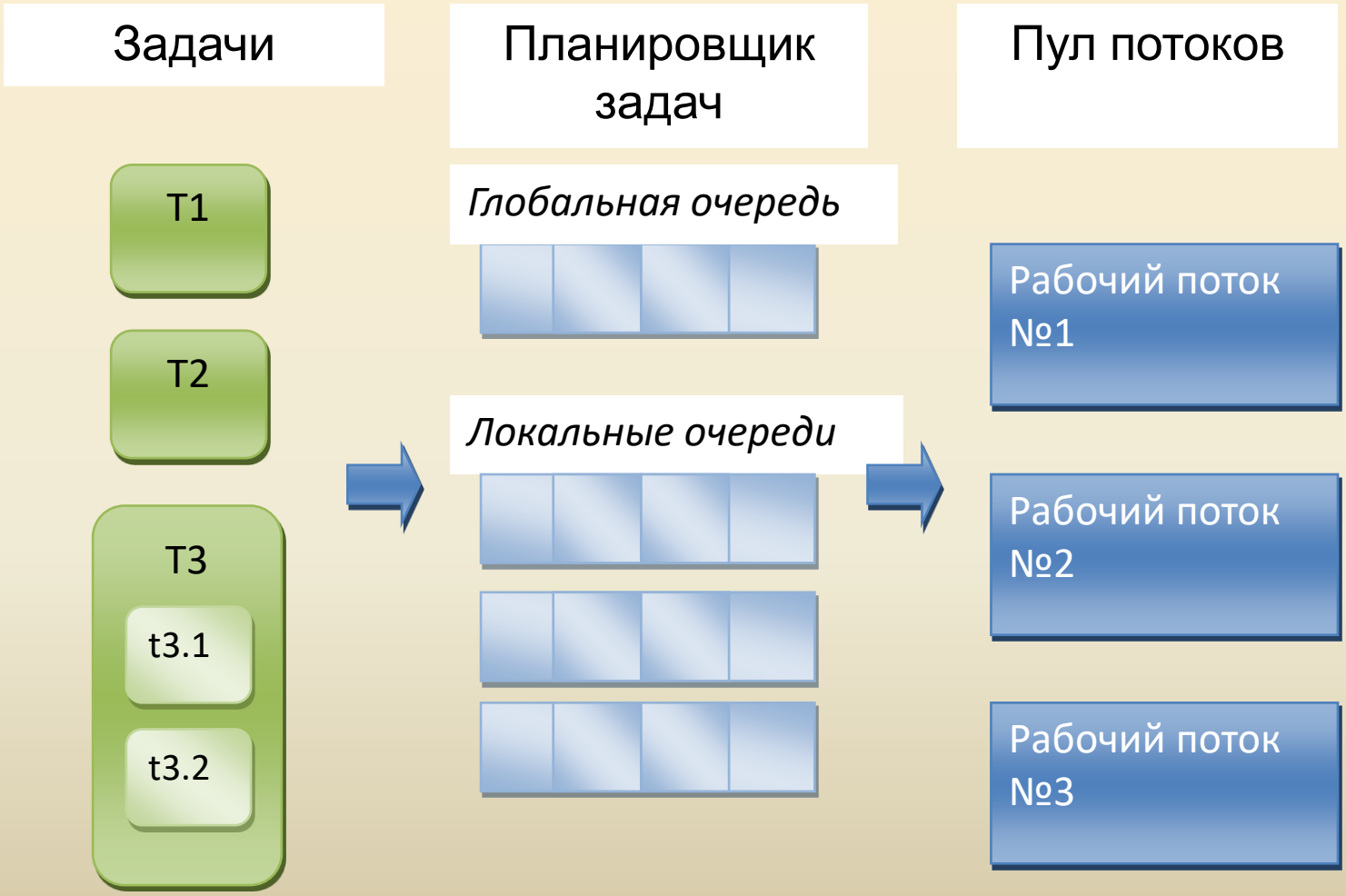
Task Parallel Library

Императивный параллелизм	• Parallel.Invoke • Parallel.For • Parallel.ForEach
Декларативный параллелизм	• Технология Parallel Language Integrated Query (PLINQ)
Асинхронный параллелизм	• Task, Task<T> • async/await • DownloadStringAsync, ..

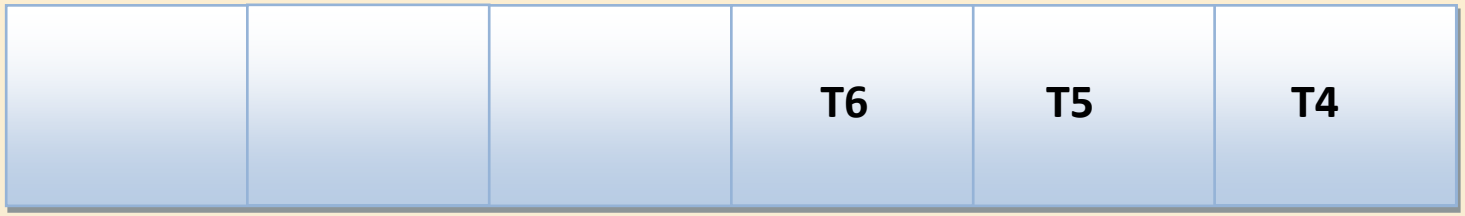
Tasks

- Задачи являются основным строительным блоком библиотеки **Task Parallel Library**
- Задачи представляют собой рабочие элементы, которые могут выполняться параллельно
- В качестве рабочих элементов могут выступать методы, делегаты, лямбда-выражения
- Для выполнения задач используются рабочие потоки пула
- Распределение пользовательских задач по потокам осуществляется планировщиком (**TaskSheduler**)

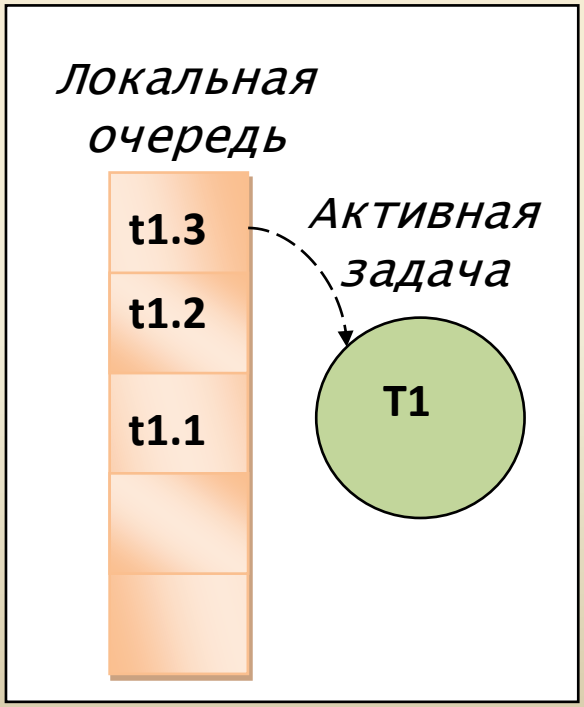
Организация планировщика



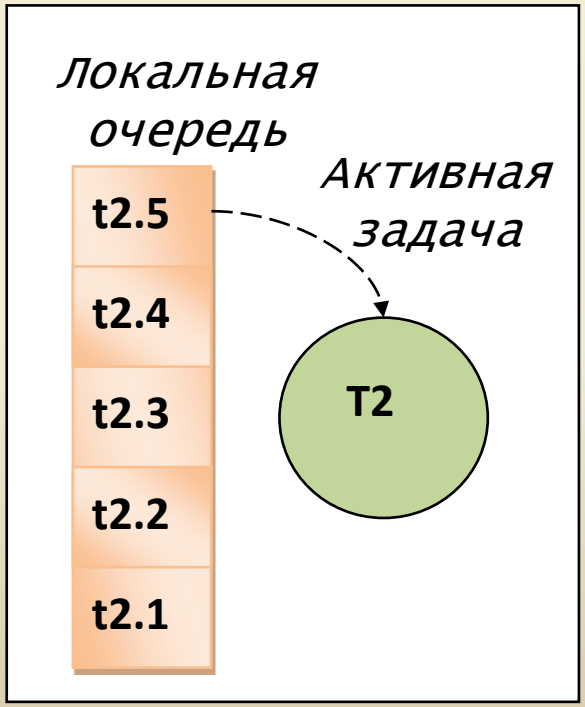
Глобальная очередь задач



Рабочий поток №1



Рабочий поток №2



...

Сценарии параллелизма Parallel

- Параллельные циклы (Parallel.For, Parallel.ForEach) и параллельный запуск нескольких независимых задач (Parallel.Invoke)
- Реализация сценариев построена на задачах (tasks
- Императивность: оператор, следующий за вызовом метода класса Parallel, будет выполняться только после завершения всех задач, неявно созданных в методе.

Метод Parallel.Invoke

void **Parallel.Invoke**(params Action[] *actions*)

actions – набор действий

void **Parallel.Invoke**(ParallelOptions *options*, params Action[] *actions*)

actions – набор действий

options – опции планировщика

Встроенный делегатный тип Action

- Action

  void f()

- Action<int>

  void f(int a)

- Action<string, double>

  void f(string s, double d)

Action

```
Action doNothing = Console.WriteLine;

Action<string, string> ShowMyName =
    (name, lastName) =>
    {
        Console.WriteLine(name + " " + lastName);
    };
```

Parallel.Invoke. Варианты запуска

```
// Методы класса
Parallel.Invoke(f1, f2, f3, f4);

// Статические методы
Parallel.Invoke(Console.Write, ConsoleWriteLine);

// Лямбда-выражения
Parallel.Invoke(
    () => { Console.WriteLine(“first”); },
    () => { Console.WriteLine(“second”); });
```

Императивность Parallel.Invoke

- Порядок обработки действий заранее не определен.

- Гарантируется, что код после Parallel.Invoke начнет выполнение только после завершения обработки всех параллельных действий

```
Console.WriteLine(“Starting..”);
Parallel.Invoke(f1, f2, f3);
Console.WriteLine(“Finished..”);
```

Parallel.Invoke vs. Task

Parallel.Invoke(FuncOne, FuncTwo)

```
// Используем задачи
Task taskTwo = Task.Factory.StartNew(FuncTwo);
Task taskOne = Task.Factory.StartNew(FuncOne);
Task.WaitAll(taskOne, taskTwo);
```

Кроме синтаксических различий есть и различия в реализации..

ParallelOptions

Опции: максимальная степень параллелизма, токен отмены, планировщик.

```
ParallelOptions pOptions = new ParallelOptions()
        {
                maxDegreeOfParallelism = 4,
                cancellationToken = cToken,
                TaskScheduler = tScheduler
        };

Parallel.Invoke(pOptions, actions);
```

Parallel.For, Parallel.ForEach

- Сценарии позволяют распараллелить обработку итераций или обработку элементов какой-либо структуры данных перечислимого типа (массив, список)
- Методы содержат ряд перегрузок, позволяющих настраивать параллелизм циклической обработки

Сигнатуры Parallel.For

- Всего 22+ перегрузки
  - Работа с итерационным индексом типа **int** или **long**
  - Возможность передачи токена отмены
  - Возможность передачи опций планировщика
  - Перегрузка для пакетной обработки
  - Перегрузка для агрегирующих вычислений

Базовая версия Parallel.For

```
ParallelLoopResult Parallel.For(
                // от..
                        int fromInclusive,
                // до ..
                        int toExclusive,
                // тело цикла
                        Action<int> body);
```

```
// Метод f соответствует сигнатуре Action<int>
Parallel.For(0, N, f);

// Статический метод
Parallel.For(0, N, Console.WriteLine);

// Лямбда-выражение
Parallel.For(0, N, i =>
    {
        c[i] = a[i] + b[i];
    });
```

for => Parallel.For

```
// Последовательный цикл
for(int i=0; i<N; i++)
        b[i] = k * a[i];

// Параллельный цикл
Parallel.For(0, N, i =>
{
        b[i] = k * a[i];
});
```

C = A x B

```
// for(int i =0; i < N; i++)
Parallel.For(0, N, i =>
{
    for (int j = 0; j < N; j++)
        for (int k = 0; k < N; k++)
            c[i][j] += a[i][k] * b[k][j];
});
```

Parallel.For

- Порядок обработки итераций не определен
- Предполагается независимость обработки итераций
- Управление возвращается только после завершения обработки всех итераций

Parallel.ForEach

- Сигнатура базовой версий

```
ParallelLoopResult Parallel.ForEach<TSource>(
        // Последовательность элементов
        IEnumerable<TSource> source,
        // Обработчик элементов
        Action<TSource> body
);
```

foreach => Parallel.ForEach

```
var names = new string[] { “Charlez”, “Jeffry”, “Dino” };

foreach(var s in names)
    Console.WriteLine(s);

Parallel.ForEach(names, s => Console.WriteLine(s));
// .. или ..
Parallel.ForEach(names, Console.WriteLine);
```

for => Parallel.ForEach

```
// Обработка множества вещественных чисел
double[] doubles = ..
Parallel.ForEach(doubles, d => Computation(d));

// Неединичный шаг цикла
var steppedNumbers = Enumerable.Range(0, n)
                                        .Select(x => x*3);
Parallel.ForEach(steppedNumbers,
                        item => Computation(item));
```

break

```
for(int i = 0; i < N; i++)
{
    DoSomething(i);
    if(Condition(i))
        break;
}
```

В последовательной версии – при любых условиях будут выполнены все итерации до той, на которой осуществляется преждевременный выход из цикла

break

- Досрочное завершение параллельного цикла реализуется двумя способами – метод Break или метод Stop

Досрочный выход из цикла

- Для досрочного прекращения обработки цикла применяются два метода объекта ParallelLoopState

- Метод **Break**, вызванный на i*-ой итерации позволяет отменить выполнение итераций с индексами j > i*, которые еще начали обрабатываться

- Метод **Stop** позволяет отменить выполнение всех итераций, которые еще не начали обрабатываться

Parallel break

Для возможности преждевременного выхода из цикла требуется обработчик итераций типа

Action<int, ParallelLoopState>

```
Parallel.For(0, N, (int i, ParallelLoopState state) =>
{
    if(i == 50)
        // state.Stop();
        state.Break();
    DoSomething(i);
});
```

ParallelLoopState

- Методы Break и Stop не прерывают выполняющиеся (активные) итерации
- При выполнении каждой итерации существует возможность получить информацию о запросе досрочного выхода из цикла

```
class ParallelLoopState {
        public bool IsExceptional { get; }
        public bool IsStopped { get; }
        public long LowestBreakIteration? { get; }
        public bool ShouldExitCurrentIteration { get; }
        public void Stop();
        public void Break();
}
```

ParallelLoopResult

Возвращаемое значение методов ParallelFor имеет тип ParallelLoopResult и используется в случае возможного досрочного выхода из цикла

```
struct ParallelLoopResult {
    // все итерации были завершены?
    public bool IsCompleted { get ; }
    // номер меньшей итерации с Break
    public long? LowestBreakIteration { get; }
}
```

ParallelLoopResult

```
ParallelLoopResult res = Parallel.For(..);
If(!res.IsCompleted)
    if(!res.LowestBreakIteration.HasValue)
        // вызов Stop
    else
        // Вызов Break на итеарции
        //       res.LowestBreakIteration.Value
else
    // Цикл выполнен полностью
```

Декомпозиция в Parallel.For/Parallel.ForEach

- Реализованы статическая и динамическая схемы разделения элементов
- По умолчанию реализуется стандартная динамическая схема разделения
- Для реализации альтернативных версий разделения необходимо использовать объект «разделитель», создаваемый методом Partitioner.Create
- Альтернативные версии декомпозиции поддерживаются в Parallel.ForEach

Статические схемы разделения

- Partitioner.Create(0, N)

  Статическая схема, размер блока выбирается исполняющей средой
- Partitioner.Create(0, N, size)

  Размер блока равен size

Обработчик элементов в этом случае имеет тип

Action<Tuple<int, int>>

Аргумент обработчика содержит информацию о диапазоне элементов текущего блока:

Item1 – начальный индекс диапазона

Item2 – конечный индекс диапазона

Статическая декомпозиция

```
Parallel.ForEach(Partitioner.Create(0, N),
        range =>
        {
                for(int i= range.Item1; i<range.Item2; i++)
                        DoSomething(i);
        });
// Размер блока фиксируется программистом
Parallel.ForEach(Partitioner.Create(0, N, N/2),
        range =>
        {
                for(int i= range.Item1; i<range.Item2; i++)
                        DoSomething(i);
        });
```

Динамическая схема

- Стандартная схема реализует динамическую декомпозицию с переменным размером блока

- Дополнительно поддерживается «сбалансированная» динамическая схема

```
Parallel.ForEach(Partitioner.Create(data, true),
                                    item => DoSomething(item));
```

Механизм согласованной отмены задач

- В библиотеке TPL (Parallel, Task, PLINQ) используется унифицированная схема согласованной отмены:
  - Управляющий поток, владеющий объектом CancellationTokenSource, может выдать сигнал отмены с помощью метода Cancel
  - При поступлении сигнала отмены планировщик отменяет еще не запланированные действия
  - Параллельные задачи (итерации в Parallel.For) могут контролировать поступление сигнала отмены и осуществлять преждевременное завершение

Механизм согласованной отмены задач

```
// Владелец токена отмены
    var cts = new CancellationTokenSource();
// Сигнал отмены
    var token = cts.Token;

    cts.CancelAfter(100);
    ..
    Parallel.For(0, N,
        new ParallelOptions() {CancellationToken=token},
        i => DoSomething(i));
```

Агрегирующие вычисления

- Вычисления с редукцией: сумма ряда, произведение, поиск максимума, средне-арифметическое, ..

```
int sum = 0;
for(int i=0; i< N; i++)
        sum += a[i];
```

Parallel.For для агрегирования

```
int sum = 0;
Parallel.For(0, N, i => {
    sum += a[i];
});
Parallel.For(0, N, i=> {
    lock(“critical”) sum += a[i];
});
Parallel.For(0, N, i => {
    Interlocked.Add(ref sum, a[i]);
});
```

Parallel.For with aggr

- В оптимальной версии параллельного цикла с редукцией для каждого потока создается свой локальный накопитель; на каждой итерации осуществляется обновление локального накопителя без средств синхронизации; в завершении обработки осуществляется агрегирование локальных накопителей.

- В версии Parallel.For необходимы три обработчика:

  Инициализация локальных накопителей: Func<T>

  Обработчик итераций: Func<int, ParallelLoopState, T, T>

  Финальный обработчик: Action<T>

Parallel for с агрегированием

```
Parallel.For(
    // индексы массива
    0, N,
    // инициализация локальных накопителей
    () => 0.0,
    // обработчик итераций
    (i, state, local) => local + a[i],
    // конечная редукция локальных накопителей
    local => { Interlocked.Add(ref sum, local)); }
);
```

```
// Пакетная обработка итераций

Parallel.ForEach(
    //
    Partitioner.Create(0, N),
    // Начальная инициализация
    () => 0.0,
    // Обработчик цикла
    (range, state, partial) =>
    {
        for(int i=range.Item1; i< range.Item2; i++)
            partial += ar[i];
        return partial;
    },
    // финальный этап
    partial =>Interlocked.Add(ref sum, partial)
);
```