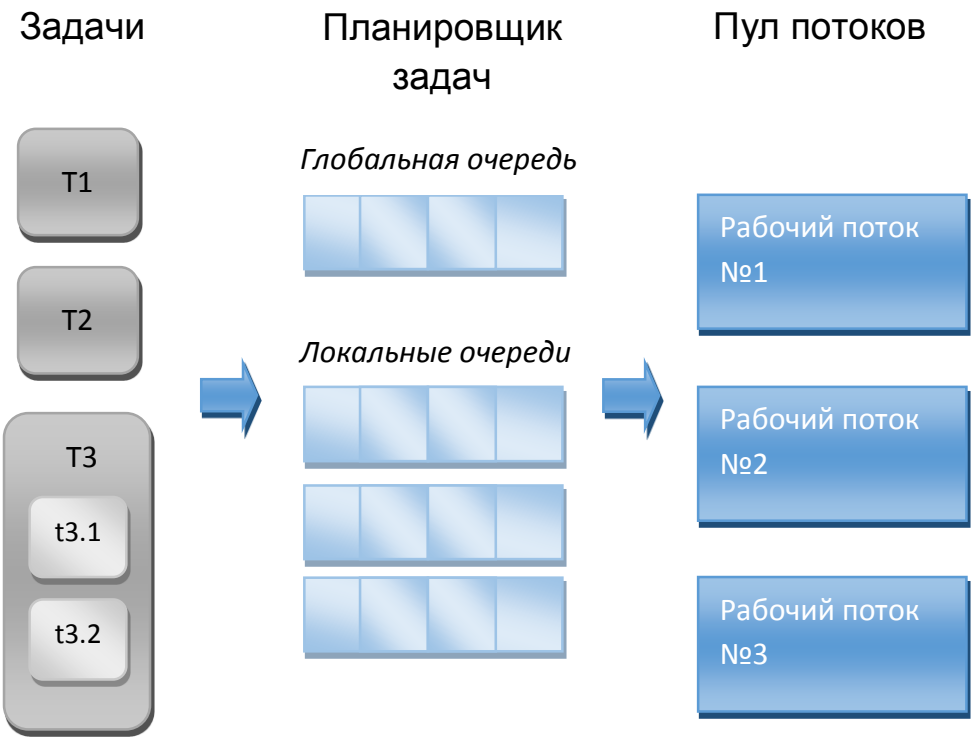
Планировщик задач

Планировщик задач играет связующую роль между задачами (рабочими элементами) и потоками. Множество задач приложения выполняется одними и теми же рабочими потоками, число которых динамически оптимизируется планировщиком с учетом возможностей вычислительной системы, фактической загруженности системы и прогрессом выполнения задач. Планировщик задач включает в себя очереди задач (одна глобальная и множество локальных очередей), стратегии распределения задач и рабочие потоки, которые фактически выполняют задачи.



Организация параллельного приложения с помощью пула потоков позволяет избежать накладных расходов, связанных с запуском и завершением потоков для каждого фрагмента в программе, допускающего распараллеливание.

Таким образом, приложение состоит из основного потока, в котором могут запускаться новые пользовательские потоки, и набора рабочих потоков. Рабочие потоки существуют на протяжении всей жизни приложения; изначально находятся в «спящем» состоянии, но могут быть задействованы для обработки пользовательских задач. Рабочие потоки включаются в обработку, если в коде программы есть работа с объектами библиотеки Task Parallel Library (Task, Parallel, PLINQ) или непосредственная работа с пулом потоков ThreadPool.

В следующем фрагменте проверяем, действительно ли рабочие потоки осуществляют обработку пользовательских задач:

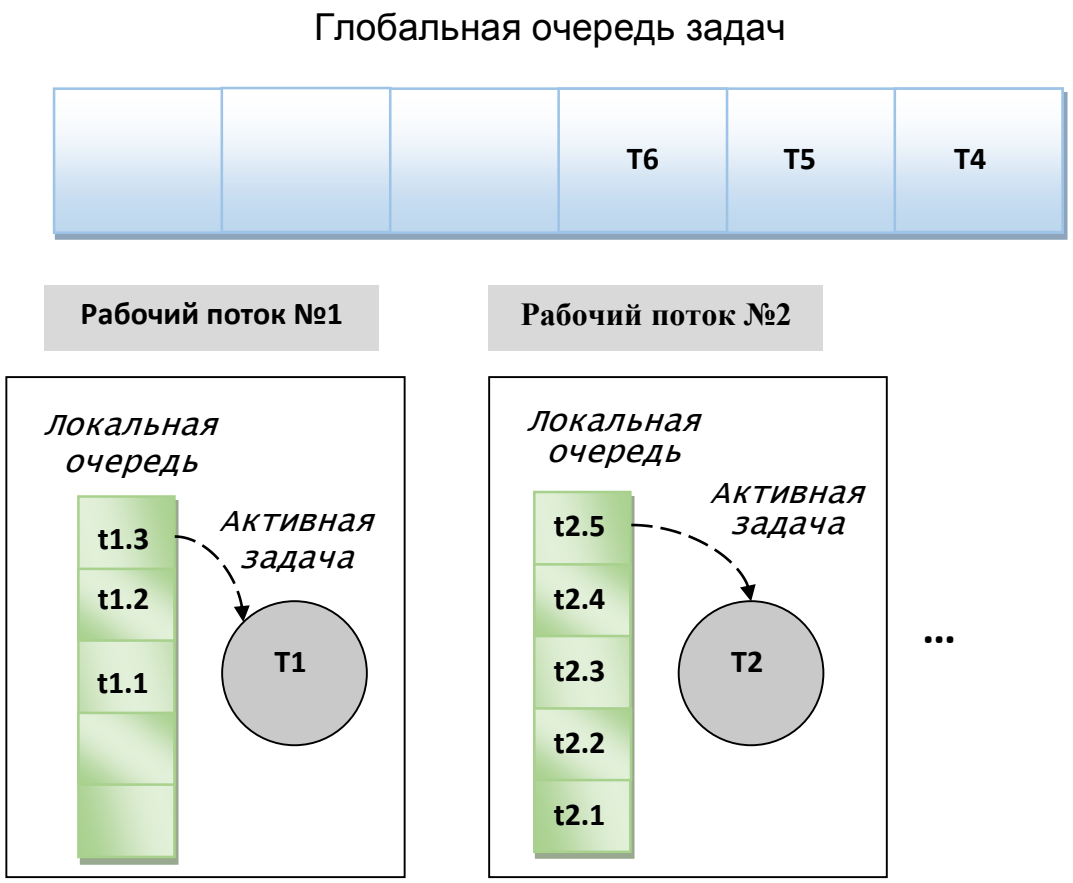
```
static void Main()
{
     Action IsPool = () => {
      Console.WriteLine(Thread.CurrentThread.IsThreadPoolThread);
     }

     Console.WriteLine("Paralle.Invoke");
     Parallel.Invoke(IsPool, IsPool, IsPool, IsPool, IsPool);

     Console.WriteLine("Paralle.For");
     Parallel.For(0, 5, i => IsPool());

     Console.WriteLine("Task");
     Task.Factory.StartNew(IsPool).Wait();

     Console.WriteLine("Thread");
     new Thread(() => IsPool()).Start();
     }
```

Итак, получаем, что задача `Task` выполняется в пуле, методы `Parallel.Invoke`, `Parallel.For` не явно создают задачи, которые тоже выполняются в пуле. Метод Main и пользовательский поток не являются рабочими потоками пула. Часть работы, определяемой в методах `Parallel.Invoke`, `Parallel.For`, может выполнять в основном потоке.

Пул потоков состоит из глобальной очереди, организованной по принципу FIFO, и множества локальных очередей, организованных по принципу LIFO. Дисциплина очереди FIFO (*first-in, first-out*) предполагает, что первый добавленный элемент первым поступит на обработку. Рабочие потоки обращаются к глобальной очереди и получают задачу на обработку. Доступ нескольких потоков к одному ресурсу – очереди задач – требует применения синхронизации для избегания проблемы гонки данных. Синхронизированный доступ потоков снижает быстродействие обработки. Чем меньше «вычислительная нагрузка» задач, тем чаще потоки соревнуются за доступ к очереди.

Для уменьшения накладных расходов, связанных с необходимостью синхронизации доступа потоков к глобальной очереди, в структуру пула потоков введены локальные очереди. Каждая локальная очередь соответствует одному рабочему потоку.

Локальные очереди предназначены для вложенных задач. Задачи, которые объявляются и запускаются из пользовательского потока, являются задачами верхнего уровня (*top-most level tasks*). Такие задачи помещаются в глобальную очередь и распределяются по рабочим потокам. При выполнении задачи верхнего уровня рабочий поток добавляет в свою локальную очередь все вложенные задачи.



Вложенные задачи помещаются в локальную очередь рабочего потока, в котором выполняется родительская задача. Локальные очереди организованы по принципу LIFO (*last-in, first-out*). Первой вложенной задачей, которая будет выполняться рабочим потоком, будет та задача, которая добавлена в очередь последней. Такой порядок объясняется возможной локальностью данных. Последняя добавленная вложенная задача использует общие данные в ближайшем окружении.

```
Task t = Task.Factory.StartNew(() =>
{
     // готовим данные для подзадачи 1
     data1 = f1(data);
     // добавляем в очередь подзадачу 1
     Task t1 = Task.Factory.StartNew(() => { work(data1); });

     // готовим данные для подзадачи 2
     data2 = f2(data);
     Task t2 = Task.Factory.StartNew(() => { work(data2); });

     // готовим данные для подзадачи 3
     data3 = f3(data);
     Task t3 = Task.Factory.StartNew(() => {work(data3); });

     // ожидаем завершения подзадач
     Task.WaitAll(t1, t2, t3);
     });
```

Благодаря стратегии планировщика *inlined execution* (см. ниже), не стоит опасаться блокировки потока, если последняя вложенная задача ссылается на предыдущую подзадачу (ожидает завершения или события из предыдущей задачи).

Опция `PreferFairness`

Если обратный порядок выполнения задач не желателен, то можно использовать опцию `PreferFairness` при создании задач. Такие задачи помещаются в глобальную очередь. Организация глобальной очереди по принципу FIFO позволяет надеяться, что вложенные задачи, созданные с опцией `PreferFairness`, будут выполняться в порядке добавления.

Рассмотрим следующий фрагмент:

```
static void Main()
{
     Task tMain = Task.Factory.StartNew(() => {
          t1 = Task.Factory.StartNew(() => ..);
          t2 = Task.Factory.StartNew(() => ..);
          t3 = Task.Factory.StartNew(() => ..,
                    TaskCreationOptions.PreferFairness);
          t4 = Task.Factory.StartNew(() => ..,
                    TaskCreationOptions.PreferFairness);
     });
}
```

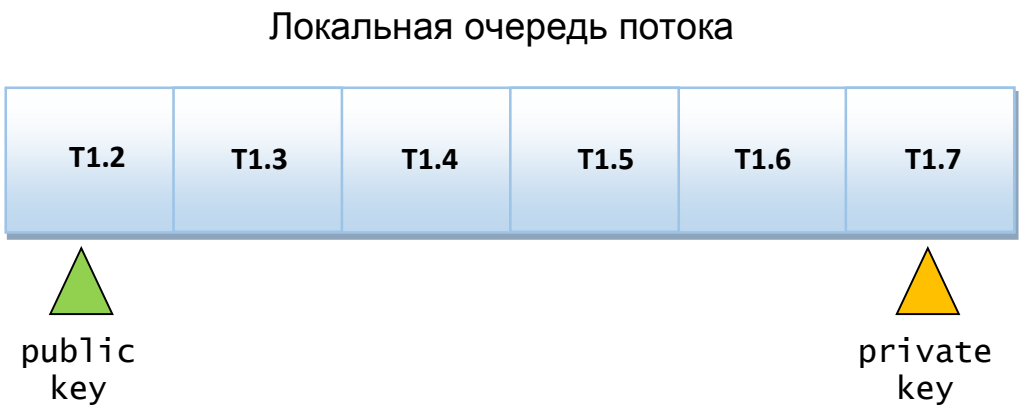
В этом фрагменте – одна задача верхнего уровня, которая помещается в глобальную очередь. При обработке задачи tMain в одном из рабочих потоков вложенные задачи t1 и t2 помещаются в локальную очередь данного рабочего потока. Задачи t3 и t4 созданы с опцией PreferFairness, поэтому добавляются в глобальную очередь. Порядок обработки вложенных задач локальной очереди – обратный порядку добавления. Сначала будет обрабатываться задача t2, затем t1. Задачи t3 и t4 обрабатываются в порядке добавления, т.е. сначала t3, затем t4. В каком рабочем потоке фактически будет выполняться задачи t1, t2, t3 и t4 зависит от загруженности других рабочих потоков (см. механизм WorkStealing). Но можно сказать, что для задач t1 и t2 созданы предпосылки последовательного выполнения в том же рабочем потоке, что и родительская задача tMain. Для задач t3 и t4 созданы предпосылки параллельного выполнения в других рабочих потоках пула. Следует не забывать, что выполнение вложенных задач в других рабочих потоках может быть сопряжено с накладными расходами при использовании данных родительской задачи.

Таким образом, опция `PreferFairness` применяется не для изменения порядка выполнения задач (для этого проще изменить порядок добавления вложенных задач в локальную очередь), а для обеспечения условий параллельного выполнения вложенных задач в разных рабочих потоках.

Оптимизация выполнения задач выполняется с помощью трех стратегий планировщика: стратегия заимствования задач (*work-stealing*), стратегия динамического изменения числа рабочих потоков (*thread-injection*), стратегия *inlined execution*.

Стратегия Work Stealing

Стратегия *work-stealing* заключается в том, что свободный рабочий поток при условии отсутствия ожидающих задач в глобальной очереди заимствует задачу у одного из загруженных рабочих потоков. Для решения проблемы синхронизации доступ к каждой локальной очереди осуществляется с помощью двух ключей: *private key* – указатель на последний добавленный элемент (задачу), который используется только одним рабочим потоком - держателем очереди; *public key* – указатель на первый добавленный элемент, который используется сторонними рабочими потоками для заимствования (*work-stealing*).



Синхронизация реализована только для открытого ключа, так как существует возможность обращения нескольких свободных рабочих потоков к локальной очереди занятого потока. Механизм заимствования задач вносит коррективы в обработку вложенных задач. Если заимствование задач не происходит, то вложенные задачи выполняются последовательно в одном рабочем потоке в порядке обратном добавлению. При заимствовании вложенные задачи могут выполняться параллельно в разных рабочих потоках.

Стратегия Inlined Execution

Стратегия *inlined execution* применяется в случае блокировки выполняющейся задачи. Если выполняющаяся задача блокируется вызовом ожидания завершения задачи, которая находится в локальной очереди этого же рабочего потока, то планировщик выполняет задачу из очереди. Такая стратегия позволяет избежать взаимоблокировки рабочего потока (*dead-lock*), которая может возникнуть при определенном порядке вложенных задач в локальной очереди. Стратегия *inlined execution* применяется только для задач в локальной очереди блокированного рабочего потока.

Рассмотрим следующий фрагмент:

```
static void Main()
{
     Task tParent = Task.Factory.StartNew(() =>
          {
               tChild1 = Task.Factory.StartNew(() => ..);
               tChild2 = Task.Factory.StartNew(() => ..);

               tChild1.Wait();
          });
}
```

В задаче `tParent` поочередно добавляются в локальную очередь `tChild1`, `tChild2`. Предполагается, что сначала должна завершить свою работу родительская задача, затем управление будет передано задаче `tChild2`, и только после этого задаче `tChild1`. Но оператор ожидания `Wait` блокирует родительскую задачу. Без стратегии планировщика `Inlined thread` текущий рабочий поток был бы заблокированным. Но в действительности планировщик (среда выполнения) фиксирует вызов Wait и передает управление ожидаемой задаче `tChild1`.

Стратегия *Inlined execution* работает при вызовах `Wait` для конкретной задачи в локальной очереди и при вызове `WaitAll` для ожидания нескольких задач локальной очереди. Вызов `WaitAny` не приводит к передаче управления вложенным задачам. Это связано с тем, что в случае метода WaitAny возникает ситуация неопределенности – кому передать управление? Вложенная задача, которая будет выбрана первой для выполнения и будет победительницей. Метод ожидания WaitAny изначально предполагает *соревновательность* выполнения задач с целью выявления победителя – задачи, первой завершившей выполнение.

Стратегия Thread-Injection

Стратегия *thread-injection* применяется для оптимизации числа рабочих потоков. Применяются два механизма динамического изменения числа рабочих потоков. Первый механизм применяется с целью избежать блокировки рабочих потоков: планировщик добавляет еще один рабочий поток, если не наблюдает прогресса при выполнении задачи. При этом планировщик не различает, находится ли поток действительно в заблокированном состоянии, ожидая какого-либо события, или поток загружен полезной длительной работой (*long-running task*). В случае выполнения «длинных» задач такая стратегия планировщика не улучшает, а даже ухудшает производительность. При росте числа потоков возникает конкуренция за физические ядра вычислительной системы, появляются накладные расходы на переключение контекстов. Чтобы избежать неправильных действий планировщика рекомендуется делать более «короткие» задачи, а длинные вычислительно-ёмкие задачи запускать с опцией `LongRunning`:

```
     Task t = Task.Factory.StartNew(VeryLongComputing,
                                    TaskCreationOptions.LongRunning);
```

Эта опция вынуждает планировщик выделить для задачи собственный поток и не контролировать прогресс выполнения данной задачи. Поток «длинной» задачи не входит в группу рабочих потоков пула, поэтому планировщик не применяет стратегии оптимизации работы этого потока и не использует поток для выполнения других задач, ожидающих в пуле.

Второй механизм стратегии *thread-injection* добавляет или удаляет рабочие потоки в зависимости от результатов выполнения предыдущих задач. Если увеличение числа потоков привело к росту производительности, то планировщик увеличивает число потоков. Если производительность не растет, то число потоков сокращается. Эвристический алгоритм стремится найти оптимальное число потоков при текущей загрузке вычислительной системы. Уменьшение объема задач также помогает планировщику найти оптимальный вариант. Второй механизм дополняет первый и позволяет избежать неограниченного добавления рабочих потоков без улучшения производительности.