Конкурентность

Цель конкурентности – предотвратить взаимоблокировки задач путем переключения между ними, когда одна из задач вынуждена ждать внешнего ресурса. Типичный пример – обработка нескольких сетевых запросов.

Вариант реализации конкурентности – запуск запросов по очереди, пока запросы не обработаются. Здесь, однако, возникает проблема со скоростью и эффективностью.

**Решение проблемы** – запустить запросы одновременно, а затем переключаться между ними по мере получения ответов.

Параллелизм

**Пример:** Есть 10 рабочих. Не хотелось бы, чтобы 1 рабочий работал, а остальные 9 сидели и ничего не делали.

Чтобы повысить эффективность, требуется разделить работу между ними, чтобы работа выполнялась быстрее.

Параллелизм использует ту же концепцию, которая применяется к аппаратным ресурсам. Речь идет о максимальном использовании этих ресурсов путем запуска процессов или потоков, использующих все ядра процессора, которыми располагает компьютер.

Обе эти концепции полезны для одновременной обработки нескольких задач, но требуется выбрать правильный метод для конкретной задачи.

Многопоточность

Поток похож на последовательную программу: у него есть начало, процесс работы и окончание работы. В момент выполнения потока существует единственная точка выполнения, однако поток не программа — поток не может выполнятся сам по себе.

Поток выполнения — одна или несколько функций, которые возможно выполнить, независимо от остальной части программы.

Процессы и потоки связаны друг с другом, но при этом имеют существенные различия.

Процесс — экземпляр программы во время выполнения, независимый объект, которому выделены системные ресурсы (например, процессорное время и память). Каждый процесс выполняется в отдельном адресном пространстве: один процесс не может получить доступ к переменным и структурам данных другого. Если процесс хочет получить доступ к чужим ресурсам, необходимо использовать межпроцессное взаимодействие. Это могут быть конвейеры, файлы, каналы связи между компьютерами и многое другое.

Поток использует то же самое пространства стека, что и процесс, а множество потоков совместно используют данные своих состояний. Как правило, каждый поток может работать (читать и писать) с одной и той же областью памяти, в отличие от процессов, которые не могут просто так получить доступ к памяти другого процесса. У каждого потока есть собственные регистры и собственный стек, но другие потоки могут их использовать.

Поток — определенный способ выполнения процесса. Когда один поток изменяет ресурс процесса, это изменение сразу же становится видно другим потокам этого процесса.

[Join](https://learn.microsoft.com/ru-ru/dotnet/api/system.threading.thread.join?view=net-7.0) — это метод синхронизации, который блокирует вызывающий поток (т. е. поток, вызывающий метод), пока не завершится поток, метод которого [Join](https://learn.microsoft.com/ru-ru/dotnet/api/system.threading.thread.join?view=net-7.0) вызывается. Используйте этот метод, чтобы убедиться, что поток был завершен. Вызывающий объект будет блокироваться на неопределенный срок, если поток не завершается. В следующем примере Thread1 поток вызывает [Join()](https://learn.microsoft.com/ru-ru/dotnet/api/system.threading.thread.join?view=net-7.0" \l "system-threading-thread-join) метод Thread2, который приводит Thread1 к блокировке, пока не Thread2 завершится.

Первоначально в среде .NET Framework нельзя было передавать аргумент потоку, когда он начинался, поскольку у метода, служившего в качестве точки входа в поток, не могло быть параметров. Но этот недостаток был впоследствии устранен, и теперь аргумент может быть передан потоку. Для этого придется воспользоваться другими формами метода Start(), конструктора класса Thread, а также метода, служащего в качестве точки входа в поток.

Аргумент передается потоку в следующей форме метода Start().

*public void Start(object параметр)*

Объект, указываемый в качестве аргумента параметр, автоматически передается методу, выполняющему роль точки входа в поток. Следовательно, для того чтобы передать аргумент потоку, достаточно передать его методу Start().

У каждого потока имеется свой приоритет, который отчасти определяет, насколько часто поток получает доступ к ЦП. Вообще говоря, низкоприоритетные потоки получают доступ к ЦП реже, чем высокоприоритетные.

Когда порожденный поток начинает выполняться, он получает приоритет, устанавливаемый по умолчанию. Приоритет потока можно изменить с помощью свойства Priority, являющегося членом класса Thread. Данное свойство может принимать одно из значений из перечислимого ThreadPriority:

* ThreadPriority.Highest
* ThreadPriority.AboveNormal
* ThreadPriority.Normal
* ThreadPriority.BelowNormal
* ThreadPriority.Lowest

По умолчанию для потока устанавливается значение приоритета ThreadPriority.Normal.