Многопоточность и асинхронность

**Однопоточное приложение** – это приложение, которое использует только один поток.

Соответственно **многопоточное приложение** – это приложение создающее и использующее несколько потоков

По умолчанию все приложения являются однопоточными, т.е. всегда есть минимум один поток, такой поток называется основным или main потоком. Дополнительные потоки называются рабочими потоками.

**Поток** – это абстракция операционной системы, позволяющая выполнять некие куски кода параллельно.

Важно понимать, что у каждого потока свой стек и регистры. Что не позволяет двум потокам взаимодействовать между собой.

Однако в приложении есть еще один объект памяти, называемый кучей, который является памятью приложения и по этой причине он общий для всех потоков.

Грубо, можно сказать, что куча – это память приложения, а стек – это память потока.

Основной целью многопоточности является более оптимальная загрузка CPU, выделяя в отдельные потоки работу которая может быть исполнена параллельно с основной деятельностью вашей программы или просто для ускорения некоторых алгоритмов.

Давайте посмотрим на один из примеров, когда многопоточность может ускорить исполнение программы:

Представим, что у нас есть большой массив над которым мы хотим выполнить какую либо операцию. Мы можем сделать это в одном потоке, вот только в большинстве случаев на современных устройствах по 4 ядра. Поэтому мы можем данный массив поделить на 4 части и выполнить действие в разных потоках над 1 / 4 массива. Прошу обратить внимание, что сложность алгоритма не поменялась, но поменялась нагрузка на систему.

Тем не менее многопоточность имеет свои проблемы, с которыми приходится сталкиваться. Одним из таких примеров может стать доступ к общему ресурсу из двух или более потоков. Если два или более потоков попробуют изменить общий ресурс из кучи, может возникнуть так называемая **гонка потоков**

Представим, что у нас есть 2 потока, которые пытаются увеличить значение переменной на единицу и допустим в текущий момент там лежит значение 33. И если 2 потока решать просуммировать это значение одновременно, то в итоге мы получим 34. Давайте посмотрим, как это может произойти.

Два потока получают к себе в регистр значение из кучи.

Увеличивают это значение на 1

И сохраняют значение обратно в кучу.

Проблема в том, что между действиями «Получить значение» и «Инкремент» есть некий промежуток времени. Т.е. мы не можем гарантировать, что поток один закончит свою операцию, а потом её начнет выполнять поток 2.

Получается, что данная операция не атомарная.

**Атомарная** **операция** — операция, которая либо выполняется целиком, либо не выполняется вовсе; операция, которая не может быть частично выполнена и частично не выполнена.

Что бы избежать гонки потоков используются **примитивы синхронизации**.

Такие как семафор или мютекс, на них я подробно останавливается не буду – это будет домашнее задание.

Основная цель примитивов синхронизации – это разрешить доступ к общему ресурсу только определенному количеству потоков, чаще одному.

Код, в котором происходит доступ к общему ресурсу называется «Критической секцией»

Еще одна опасность в многопоточном программировании, связанная с примитивами синхронизации, по большей части, и называется она дедлок.

**Дедлок** – это ситуация, когда 2 или несколько потоков пытаются получить доступ к ресурсам и в процессе этого блокируют работу друг друга.

Один из примеров как получить дедлок показан в таблице на слайде.

Когда два потока пытаются захватить по два общих ресурса и при этом поток один захватывает ресурс А, а потом поток два захватывает ресурс Б, а после они пытаются захватить противоположный ресурс, но не могут этого сделать, т.к. он занят другим потоком и так они повиснут навсегда.

Теперь давайте представим ситуацию, когда в приложении создается больше потоков чем есть ядер в системе. Что тогда будет происходить? Что бы это понять нам нужно понять такое понятие как конкурентность.

Конкурентность говорит о том, что система, в нашем случае операционная, обязуется выполнить наши задачи одновременно, но не говорит как это будет достигаться.

Классический пример. На одноядерном компьютере может быть запущенно несколько программ параллельно друг другу. И они будут работать нормально благодаря переключению контекстов.

Переключение контекстов – это один из способов достижения конкурентности.

Как показано на слайде, несколько потоков конкурируют за процессорное время, пропуски между ними – это и есть время на переключение контекста.

Поэтому, когда в приложении создается больше потоков чем ядер у системы, эти потоки вынуждены конкурировать за процессорное время.

В случае когда, у нас в приложении меньше потоков чем ядер, мы получаем параллельное исполнение.

Параллельное исполнение – это строгое подмножество конкурентного исполнения. Но потоки при таком исполнении выполняются параллельно благодаря наличию нескольких ядер в системе.

Это значит, что на компьютере с одним процессором и ядром параллельное программирование – невозможно.

На слайде показано, как потоки могут быть распределены при параллельном исполнении.

Хорошо, я думаю мы разобрались с многопоточностью, теперь давайте разберемся с синхронностью и асинхронностью.

И начнем мы с синхронности. Что значит синхронное выполнение? Это значит, что наш код будет выполнять все операции последовательно друг за другом. Даже если какая-то операция блокирует поток.

Но что значит заблокировать поток? Как узнать, что поток будет заблокирован?

Блокирующая операция – это операция, которая заставляет процессор в холостую тратить свою мощность. Т.е. процессор мог бы делать какую то полезную работу, но вместо этого он ожидает завершения какой либо операции.

И тут нужно понимать, что у нас есть 2 вида операций:

**CPU зависимые операции и Операции ввода/вывода или I/O операция**

CPU операции – «Съедают» такты процессора и при этом никак не взаимодействуют с внешней средой. Это такие операции как Условия, циклы, арифметические или битовые операции и т.д. Они тоже блокируют поток, но это единственно возможный путь их исполнения и помимо этого в этот момент процессор делает полезную работу.

**Операции ввода/вывода –** Это операции с помощью которых приложение взаимодействует с внешней средой. Такие как вывод на консоль, вывод на печать, обращение к удаленному серверу, чтение файла, обращение к серверу БД и т.д. Эти операции не нагружают процессор, ну если только для инициализации своей деятельности, а потом в дело вступает контроллер конечного устройства.

И при синхронном подходе мы вынуждены ждать операции ввода/вывода в режиме ожидания, т.е. поток из которого вызываются I/O операции будет крутиться впустую.

Например. Если мы в синхронном подходе попробуем прочитать большой файл в однопоточной программе, то вся программа зависнет до тех пор, пока жесткий диск не прочитает весь файл. В этот момент программа не будет реагировать ни на какие действия пользователя. В Windows вы могли сталкивались с проблемой, что «Программа не отвечает» - это все из-за блокирующих операций, ну или дедлока.

И вот именно от таких проблем нас и спасает асинхронность.

Асинхронность – это антоним синхронности, т.е. наши операции будут выполнены когда то в другой момент времени. Мы не знаем когда точно, потому что не знаем сколько ждать, но при этом, мы не будем блокировать поток. Т.е. асинхронные операции – это не блокирующие операции.

Как это происходит. Когда вы встречаете I/O операцию, вы можете запустить её асинхронно. Теперь эту операцию можно не дожидаться, можно указать, так называемый, callback, который будет выполнен по завершении операции. Вы могли такое видеть в каком-нибудь JavaScript, если работали с Ajax. Ну и в принципе в JS calback’и в почете  
При вызове асинхронной операции метод как бы завершается, что позволяет потоку не крутиться в ожидании завершения операции, а приступить к выполнению других задач.

Так же асинхронность позволяет создавать отзывчивые приложения.

Вернемся к примеру с чтением большого файла. Если бы мы сделали чтение файла в асинхронном виде, то все что нам требуется сделать – это послать жесткому диску запрос на чтение файла. А дальше мы завершаем работу нашего кода, но когда жесткий диск закончит читать файл, он сообщит об этом нашей программе и она сможет работать с данными из файла. При этом, мы освободили поток и программа не зависла, продолжая отвечать пользователю.

Т.е. асинхронный код, помогает нам использовать систему эффективно. Отпуская потоки тогда, когда нет смысла их держать.

При этом у асинхронности появляются свои недостатки. Как правило асинхронный код сложнее читать и поддерживать, т.к. выглядит он как портянка из callback’ов. В C# асинхронность сделали элегантно, но при этом там есть дополнительная нагрузка на CPU, как плата за удобство. А вот в каком-нибудь C++ можно легко потеряться в асинхронности.

Когда я готовился я посмотрел, как в Python обеспечивается асинхронность и по синтаксису она схожа с C#, питон должен быть выше версии 3.5. Здесь так же используется 2 ключевых слова async и await. Чтов каком-то смысле является хорошей новостью, потому что модель асинхронности в C# выглядит как синхронная, с некоторыми оговорками.

Последний важный момент использование многопоточности не исключает использование асинхронности. Вместе они создают очень мощный тандем.  
  
Ваше приложение может быть однопоточным и асинхронными, а может многопоточным, но синхронным или многопоточным и асинхронным.

Т.е. Многопоточность и асинхронность повышают эффективность использования системы.

Итоги:

* Использование более одного потока в приложении делает его многопоточным.
* Многопоточность позволяет использовать всю мощь многоядерных вычислительных машин.
* Многопоточность увеличивает сложность понимания программ. Нужно четко представлять, как ваши потоки будут использовать общие ресурсы, определяя критические секции и избегая при этом дедлоков.
* Создание большого количества потоков в приложении, практически не имеет смысла, если такое количество потоков не подкреплено схожим количеством ядер, в таком случае стоит помнить о конкурентности и параллельности.
* Асинхронность позволяет освободить поток приложения в случаях, когда операция, которую нам хотелось бы дождаться является операцией ввода/вывода
* В некоторых языках, асинхронный код становится сложным для поддержки и понимания

Некоторые определения:

**Параллельное (parallel) исполнение** - это один из способов конкурентногоисполнения.

Параллельное исполнение - подразумевает наличие более одного вычислительного устройства, например, несколько процессоров или несколько ядер, которые будут одновременно выполнять несколько задач.

**Конкурентное (concurrency) исполнение** - это наиболее общий термин, который говорит, что одновременно выполняется более одной задачи. Данный термин не говорит о том, каким образом эта конкурентностьбудет получена.

Т.е. **Критическая секция** – это участок исполняемого кода программы, в котором производится доступ к общему ресурсу (данным или устройству), который не должен быть одновременно использован более чем одним потоком выполнения.

**Атомарная** **операция** — операция, которая либо выполняется целиком, либо не выполняется вовсе; операция, которая не может быть частично выполнена и частично не выполнена.

**CPU зависимые операции** – это операции которые используют время центрального процессора. Такие как: Условия, циклы, арифметические операции, битовые операции и т.д.

**Операции ввода/вывода или I/O операция** – это операции в которых происходит обращение к внешним ресурсам. Такие как вывод в консоль, вывод на печать, обращение к удаленному серверу, чтение файла и т.д.

**Синхронные операции** — операции, при которых мы получаем результат в результате блокирования потока выполнения. Для CPU операций – это единственный способ исполнения, а вот для операций ввода/вывода один из.