1. **Что такое процесс, домен, поток? Как они связаны между собой?**

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.

При запуске приложения, написанного на C#, операционная система создает процесс, а среда CLR создает внутри этого процесса логический контейнер, который называется **доменом приложения** и внутри которого работает запущенное приложение.

1. **Как получить информацию о процессах?**

С помощью класса Process, определённого в простанстве имён System.Diagnostics

1. **Как создать и настроить домен?**

    AppDomain secondaryDomain =  AppDomain.CreateDomain("Secondary domain");

secondaryDomain.Load(new AssemblyName("System.Data"));

1. **Как создать и настроить поток?**
2. **В каких состояниях может быть поток?**
3. **Какие методы управления потоками вы знаете, для чего и как их использовать?**

Поток создается за счет вызова метода Start() объекта Thread. Однако после вызова метода Start() новый поток все еще пребывает не в состоянии Running, а в состоянии *Unstarted*. В состояние Running поток переходит сразу после того, как планировщик потоков операционной системы выберет его для выполнения. Информация о текущем состоянии потока доступна через свойство **Thread.ThreadState**.

С помощью метода Thread.Sleep() поток можно перевести в состояние WaitSleepJoin и при этом указать, через какой промежуток времени поток должен возобновить работу.

Чтобы остановить поток, необходимо вызвать метод Thread.Abort(). При вызове этого метода в соответствующем потоке генерируется исключение типа ThreadAbortException. В случае если для этого исключения предусмотрен обработчик, перед завершением поток сможет выполнить необходимые операции по очистке. Чтобы продолжить выполнение потока после выдачи исключения ThreadAbortException, следует вызвать метод **Thread.ResetAbort()**. Состояние потока, получающего запрос на немедленное прекращение, изменяется с AbortRequested на Aborted, если поток не производит сброс.

Если необходимо дожидаться завершения работы потока, можно вызвать метод Thread.Join(). Этот метод блокирует текущий поток и переводит его в состояние WaitSleepJoin до тех пор, пока не будет завершен присоединенный к нему поток.

1. **Какие приоритеты потока вы знаете?**

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

1. **Что такое пул потоков и для чего он используется?**

Создание потоков требует времени. Если есть различные короткие задачи, подлежащие выполнению, можно создать набор потоков заранее и затем просто отправлять соответствующие запросы, когда наступает очередь для их выполнения. Было бы неплохо, если бы количество этих потоков автоматически увеличивалось с ростом потребности в потоках и уменьшалось при возникновении потребности в освобождении ресурсов.

Создавать подобный список потоков самостоятельно не понадобится. Для управления таким списком предусмотрен **класс ThreadPool**, который по мере необходимости уменьшает и увеличивает количество потоков в пуле до максимально допустимого. Значение максимально допустимого количества потоков в пуле может изменяться. В случае двуядерного ЦП оно по умолчанию составляет 1023 рабочих потоков и 1000 потоков ввода-вывода.

Можно указывать минимальное количество потоков, которые должны запускаться сразу после создания пула, и максимальное количество потоков, доступных в пуле. Если остались какие-то подлежащие обработке задания, а максимальное количество потоков в пуле уже достигнуто, то более новые задания будут помещаться в очередь и там ожидать, пока какой-то из потоков завершит свою работу.

Чтобы запросить поток из пула для обработки вызова метода, можно использовать метод **QueueUserWorkItem()**. Этот метод перегружен, чтобы в дополнение к экземпляру делегата WaitCallback позволить указывать необязательный параметр System.Object для специальных данных состояния.

Ниже приведен пример приложения, в котором сначала читается и выводится на консоль информация о максимальном количестве рабочих потоков и потоков ввода-вывода. Затем в цикле for метод JobForAThread() назначается потоку из пула потоков за счет вызова метода ThreadPool.QueueUserWorkltem() и передачи делегата типа WaitCallback. Пул потоков получает этот запрос и выбирает из пула один из потоков для вызова метода. Если пул еще не существует, он создается и запускается первый поток. Если же пул существует и в нем имеется один свободный поток, задание переадресовывается этому потоку

1. **Что такое критическая секция? Поясните использование.**

Оператор lock определяет блок кода, внутри которого весь код блокируется и становится недоступным для других потоков до завершения работы текущего потока.

Класс Monitor обладает одним важным преимуществом по сравнению с оператором lock в C#: он позволяет добавлять значение тайм-аута для ожидания получения блокировки. Таким образом, вместо того, чтобы ожидать блокировку до бесконечности, можно вызвать метод TryEnter и передать в нем значение тайм-аута, указывающее, сколько максимум времени должно ожидаться получение блокировки.

Когда блокировка obj получена, метод **TryEnter()** устанавливает булевский параметр ref в true и производит синхронизированный доступ к состоянию, охраняемому объектом obj. Если obj блокируется другим потоком на протяжении более 500 миллисекунд, то TryEnter() устанавливает переменную lockTaken в false и поток больше не ожидает, а используется для выполнения другой работы. Возможно, позже поток попытается получить блокировку еще раз.

**10.Что такое мьютекс? Поясните использование**

*Класс Mutex (mutual exclusion — взаимное исключение или мьютекс)* является одним из классов в .NET Framework, позволяющих обеспечить синхронизацию среди множества процессов. Он очень похож на класс Monitor тем, что тоже допускает наличие только одного владельца. Только один поток может получить блокировку и иметь доступ к защищаемым мьютексом синхронизированным областям кода.

В конструкторе класса Mutex указывается, должен ли мьютексом изначально владеть вызывающий поток, и его имя. Кроме того, конструктор позволяет получить информацию о том, существует ли уже такой класс.

***Мьютекс*** представляет собой взаимно исключающий синхронизирующий объект. Это означает, что он может быть получен потоком только по очереди. Мьютекс предназначен для тех ситуаций, в которых общий ресурс может быть одновременно использован только в одном потоке. Допустим, что системный журнал совместно используется в нескольких процессах, но только в одном из них данные могут записываться в файл этого журнала в любой момент времени. Для синхронизации процессов в данной ситуации идеально подходит мьютекс.

**11.Что такое семафор? Поясните использование**

***Семафор*** подобен мьютексу, за исключением того, что он предоставляет одновременный доступ к общему ресурсу не одному, а нескольким потокам. Поэтому семафор пригоден для синхронизации целого ряда ресурсов. Семафор управляет доступом к общему ресурсу, используя для этой цели счетчик. Если значение счетчика больше нуля, то доступ к ресурсу разрешен. А если это значение равно нулю, то доступ к ресурсу запрещен. С помощью счетчика ведется подсчет количества разрешений. Следовательно, для доступа к ресурсу поток должен получить разрешение от семафора.

Обычно поток, которому требуется доступ к общему ресурсу, пытается получить разрешение от семафора. Если значение счетчика семафора больше нуля, то поток получает разрешение, а счетчик семафора декрементируется. В противном случае поток блокируется до тех пор, пока не получит разрешение. Когда же потоку больше не требуется доступ к общему ресурсу, он высвобождает разрешение, а счетчик семафора инкрементируется. Если разрешения ожидает другой поток, то он получает его в этот момент. Количество одновременно разрешаемых доступов указывается при создании семафора. Так, если создать семафор, одновременно разрешающий только один доступ, то такой семафор будет действовать как мьютекс.

Семафоры особенно полезны в тех случаях, когда общий ресурс состоит из группы или пула ресурсов. Например, пул ресурсов может состоять из целого ряда сетевых соединений, каждое из которых служит для передачи данных. Поэтому потоку, которому требуется сетевое соединение, все равно, какое именно соединение он получит. В данном случае семафор обеспечивает удобный механизм управления доступом к сетевым соединениям.

**12.Что такое неблокирующие средства синхронизации?**

**13.Для чего можно использовать класс Timer?**

Данный класс позволяет запускать определенные действия по истечению некоторого периода времени.