**1 Процессы и потоки в ОС**

**1.1 Понятие процесса и потока**

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

*Представленная теоретическая часть для ОС Windows и не актуальная.*

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

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.

**1.2 Многопоточное программирование**

С помощью языка .NET можно создавать приложения, которые выполняют несколько операций одновременно. Операции, которые потенциально могут задержать выполнение других операций, выполняются в отдельных потоках; такой способ организации работы приложения называется многопоточностью или свободным созданием потоков.

Приложения, использующие многопоточность, более оперативно реагируют на действия пользователя, поскольку пользовательский интерфейс остается активным, в то время как задачи, требующие интенсивной работы процессора, выполняются в других потоках. Многопоточность также эффективна при создании масштабируемых приложений, поскольку пользователь может добавлять потоки при увеличении рабочей нагрузки.

*Если требуется больший контроль над поведением потоков приложения, можно управлять потоками самостоятельно. Но начиная с .NET Framework 4 многопоточное программирование значительно упростилось благодаря классам System.Threading.Tasks.Parallel и System.Threading.Tasks.Task, Parallel LINQ (PLINQ), новым классам параллельных коллекций из пространства имен System.Collections.Concurrent и новой модели программирования, которая вместо потоков использует концепцию задач.*

**1.3 Работа с потоками**

1.3.1 Создание и запуск нового потока

Чтобы создать поток, создайте новый экземпляр класса System.Threading.Thread и укажите в конструкторе имя метода, который должен выполняться в новом потоке. Чтобы запустить созданный поток, вызовите метод Thread.Start.

1.3.2 Остановка потока

Чтобы прервать выполнение потока, используйте метод System.Threading.CancellationToken. Это единый способ совместной отмены потоков.

Иногда выполнить совместную отмену потока невозможно, так как он выполняет код сторонних производителей, не поддерживающий такую отмену. В этом случае вы можете выполнить принудительное завершение. Чтобы принудительно завершить выполнение потока, воспользуйтесь методом Thread.Abort в .NET Framework. Этот метод вызывает ThreadAbortException в потоке, для которого был вызван. Метод Thread.Abort не поддерживается в .NET Core. Если необходимо принудительно завершить выполнение кода сторонних производителей в .NET Core, запустите его в отдельном процессе и воспользуйтесь Process.Kill.

System.Threading.CancellationToken можно использовать только начиная с .NET Framework 4. Чтобы завершить поток в более ранних версиях .NET Framework, следует вручную реализовать совместную отмену с помощью методов синхронизации потоков. Например, вы можете создать изменяемое логическое поле shouldStop и использовать его для запроса остановки кода, выполняемого потоком.

Используйте метод Thread.Join, чтобы вызывающий поток ждал завершения останавливаемого потока.

1.3.3 Приостановка или прерывание потока

Используйте метод Thread.Sleep, чтобы приостановить текущий поток на определенное время. Заблокированный поток можно прервать путем вызова метода Thread.Interrupt.

**1.4 Класс Thread и свойства потоков**

Класс Thread является самым элементарным из всех типов пространства имен System.Threading. Этот класс представляет объектно-ориентированную оболочку вокруг заданного пути выполнения внутри определенного AppDomain. Этот тип также определяет набор методов (как статических, так и уровня экземпляра), которые позволяют создавать новые потоки внутри текущего AppDomain, а также приостанавливать, останавливать и уничтожать определенный поток.

Основной функционал для использования потоков в приложении сосредоточен в пространстве имен System.Threading. В нем определен класс, представляющий отдельный поток - класс Thread (табл. 1.1).

Таблица 1.1

**Свойства Thread**

| **Свойство** | **Описание** |
| --- | --- |
| IsAlive | Возвращает true, если поток был запущен и не был завершен нормально либо был прерван. |
| IsBackground | Возвращает или задает логическое значение, которое указывает, является ли поток фоновым потоком. Фоновые потоки отличаются от основных потоков лишь тем, что они не влияют на завершение процесса. Когда обработка всех основных потоков закончена, общеязыковая среда выполнения завершает процесс, применяя метод Abort к тем фоновым потокам, которые еще продолжают существовать. |
| Name | Возвращает или задает имя потока. Наиболее часто используется для обнаружения отдельных потоков при отладке. |
| Priority | Возвращает или задает значение ThreadPriority, которое используется операционной системой для установки приоритетов потоков. Дополнительные сведения см. в разделе Планирование потоков и в справочнике по ThreadPriority. |
| ThreadState | Возвращает значение ThreadState, содержащее текущие состояния потока. |

**1.5 Создание потоков и передача данных во время запуска**

При создании процесса в операционной системе она добавляет поток для выполнения кода этого процесса, включая все исходные домены приложений. С этого момента домены приложений могут создаваться и уничтожаться, что не обязательно сопровождается созданием или уничтожением потоков операционной системы. Если выполняемый код является управляемым, то вы можете получить объект Thread для потока, выполняющегося в текущем домене приложения. Для этого получите статическое свойство CurrentThread типа Thread.

1.5.1 Создание потока

Создание нового объекта Thread приводит к созданию нового управляемого потока. Класс Thread имеет конструкторы, которые принимают делегат ThreadStart или ParameterizedThreadStart. Этот делегат инкапсулирует метод, который вызывается новым потоком при вызове метода Start. Повторный вызов Start приводит к созданию исключения ThreadStateException.

Метод Start завершается немедленно, часто даже до запуска нового потока. Вы можете использовать свойства ThreadState и IsAlive, чтобы определить состояние потока в настоящий момент, но эти свойства ни в коем случае нельзя использовать для синхронизации действий потоков.

*После запуска потока не нужно сохранять ссылку на объект Thread. Поток продолжит выполняться, пока не завершится запущенная в нем процедура.*

В следующем примере кода создается два новых потока для вызова метода экземпляра и статического метода из другого объекта.

using System;

using System.Threading;

public class ServerClass

{

// The method that will be called when the thread is started.

public void InstanceMethod()

{

Console.WriteLine(

"ServerClass.InstanceMethod is running on another thread.");

// Pause for a moment to provide a delay to make

// threads more apparent.

Thread.Sleep(3000);

Console.WriteLine(

"The instance method called by the worker thread has ended.");

}

public static void StaticMethod()

{

Console.WriteLine(

"ServerClass.StaticMethod is running on another thread.");

// Pause for a moment to provide a delay to make

// threads more apparent.

Thread.Sleep(5000);

Console.WriteLine(

"The static method called by the worker thread has ended.");

}

}

public class Simple

{

public static void Main()

{

ServerClass serverObject = new ServerClass();

// Create the thread object, passing in the

// serverObject.InstanceMethod method using a

// ThreadStart delegate.

Thread InstanceCaller = new Thread(

new ThreadStart(serverObject.InstanceMethod));

// Start the thread.

InstanceCaller.Start();

Console.WriteLine("The Main() thread calls this after "

+ "starting the new InstanceCaller thread.");

// Create the thread object, passing in the

// serverObject.StaticMethod method using a

// ThreadStart delegate.

Thread StaticCaller = new Thread(

new ThreadStart(ServerClass.StaticMethod));

// Start the thread.

StaticCaller.Start();

Console.WriteLine("The Main() thread calls this after "

+ "starting the new StaticCaller thread.");

}

}

// The example displays the output like the following:

// The Main() thread calls this after starting the new InstanceCaller thread.

// The Main() thread calls this after starting the new StaticCaller thread.

// ServerClass.StaticMethod is running on another thread.

// ServerClass.InstanceMethod is running on another thread.

// The instance method called by the worker thread has ended.

// The static method called by the worker thread has ended.

1.5.2 Передача данных в потоки

На платформе .NET Framework версии 2.0 делегат ParameterizedThreadStart предоставляет простой способ передать в поток объект с данными при вызове перегрузки метода Thread.Start.

Делегат ParameterizedThreadStart не является типобезопасным способом передачи данных, так как перегруженный метод Thread.Start принимает любой объект. Вместо этого можно инкапсулировать процедуру потока и данные во вспомогательный класс и выполнять процедуру потока при помощи делегата ThreadStart. В следующем примере показана эта техника.

using System;

using System.Threading;

// The ThreadWithState class contains the information needed for

// a task, and the method that executes the task.

//

public class ThreadWithState

{

// State information used in the task.

private string boilerplate;

private int numberValue;

// The constructor obtains the state information.

public ThreadWithState(string text, int number)

{

boilerplate = text;

numberValue = number;

}

// The thread procedure performs the task, such as formatting

// and printing a document.

public void ThreadProc()

{

Console.WriteLine(boilerplate, numberValue);

}

}

// Entry point for the example.

//

public class Example

{

public static void Main()

{

// Supply the state information required by the task.

ThreadWithState tws = new ThreadWithState(

"This report displays the number {0}.", 42);

// Create a thread to execute the task, and then

// start the thread.

Thread t = new Thread(new ThreadStart(tws.ThreadProc));

t.Start();

Console.WriteLine("Main thread does some work, then waits.");

t.Join();

Console.WriteLine(

"Independent task has completed; main thread ends.");

}

}

// The example displays the following output:

// Main thread does some work, then waits.

// This report displays the number 42.

// Independent task has completed; main thread ends.

Ни один из делегатов ThreadStart или ParameterizedThreadStart не имеет возвращаемого значения, так как им некуда возвращать данные после асинхронного вызова. Чтобы получить результаты из метода потока, вы можете использовать метод обратного вызова, как показано в следующем разделе.

**1.6 Извлечение данных из потоков с помощью методов обратного вызова**

Приведенный ниже пример демонстрирует метод обратного вызова, который получает данные из потока. Конструктор класса, содержащего данные и метод потока, также принимает делегат, который представляет метод обратного вызова. Метод потока перед завершением своей работы вызывает этот делегат обратного вызова.

using System;

using System.Threading;

// The ThreadWithState class contains the information needed for

// a task, the method that executes the task, and a delegate

// to call when the task is complete.

//

public class ThreadWithState

{

// State information used in the task.

private string boilerplate;

private int numberValue;

// Delegate used to execute the callback method when the

// task is complete.

private ExampleCallback callback;

// The constructor obtains the state information and the

// callback delegate.

public ThreadWithState(string text, int number,

ExampleCallback callbackDelegate)

{

boilerplate = text;

numberValue = number;

callback = callbackDelegate;

}

// The thread procedure performs the task, such as

// formatting and printing a document, and then invokes

// the callback delegate with the number of lines printed.

public void ThreadProc()

{

Console.WriteLine(boilerplate, numberValue);

if (callback != null)

callback(1);

}

}

// Delegate that defines the signature for the callback method.

//

public delegate void ExampleCallback(int lineCount);

// Entry point for the example.

//

public class Example

{

public static void Main()

{

// Supply the state information required by the task.

ThreadWithState tws = new ThreadWithState(

"This report displays the number {0}.",

42,

new ExampleCallback(ResultCallback)

);

Thread t = new Thread(new ThreadStart(tws.ThreadProc));

t.Start();

Console.WriteLine("Main thread does some work, then waits.");

t.Join();

Console.WriteLine(

"Independent task has completed; main thread ends.");

}

// The callback method must match the signature of the

// callback delegate.

//

public static void ResultCallback(int lineCount)

{

Console.WriteLine(

"Independent task printed {0} lines.", lineCount);

}

}

// The example displays the following output:

// Main thread does some work, then waits.

// This report displays the number 42.

// Independent task printed 1 lines.

// Independent task has completed; main thread ends.

**1.7 Приостановка и прерывание потоков**

Наиболее распространенными способами синхронизации действий потоков являются блокировка и освобождение потоков или блокировка объектов или областей кода.

Также можно организовать перевод потоков в спящий режим. Если потоки заблокированы или находятся в спящем режиме, можно использовать ThreadInterruptedException для вывода потоков из состояния ожидания.

1.7.1 Метод Thread.Sleep

Thread.Sleep() — статический метод класса Thread, который приостанавливает выполнение потока, в котором он был вызван. Во время выполнения метода Sleep() система перестает выделять потоку процессорное время, распределяя его между другими потоками. Метод Sleep() может выполняться либо заданное кол-во времени (миллисекунды или наносекунды) либо до тех пор пока он не будет остановлен прерыванием (в этом случае он сгенерирует исключение InterruptedException).

Вызов метода Thread.Sleep приводит к немедленной блокировке текущего потока на определенное количество миллисекунд, переданное этому методу, вследствие чего остаток среза времени передается другому потоку. По истечении этого интервала времени спящий поток возобновляет выполнение.

Поток не может вызвать метод Thread.Sleep для другого потока. Статический метод Thread.Sleep всегда переводит текущий поток в спящий режим.

Вызов метода Thread.Sleep с аргументом Timeout.Infinite переводит поток в спящий режим до того момента, пока он не будет прерван другим потоком путем вызова метода Thread.Interrupt или завершен путем вызова метода Thread.Abort.

using System;

using System.Threading;

public class Example

{

public static void Main()

{

// Interrupt a sleeping thread.

var sleepingThread = new Thread(Example.SleepIndefinitely);

sleepingThread.Name = "Sleeping";

sleepingThread.Start();

Thread.Sleep(2000);

sleepingThread.Interrupt();

Thread.Sleep(1000);

sleepingThread = new Thread(Example.SleepIndefinitely);

sleepingThread.Name = "Sleeping2";

sleepingThread.Start();

Thread.Sleep(2000);

sleepingThread.Abort();

}

private static void SleepIndefinitely()

{

Console.WriteLine("Thread '{0}' about to sleep indefinitely.",

Thread.CurrentThread.Name);

try {

Thread.Sleep(Timeout.Infinite);

}

catch (ThreadInterruptedException) {

Console.WriteLine("Thread '{0}' awoken.",

Thread.CurrentThread.Name);

}

catch (ThreadAbortException) {

Console.WriteLine("Thread '{0}' aborted.",

Thread.CurrentThread.Name);

}

finally

{

Console.WriteLine("Thread '{0}' executing finally block.",

Thread.CurrentThread.Name);

}

Console.WriteLine("Thread '{0} finishing normal execution.",

Thread.CurrentThread.Name);

Console.WriteLine();

}

}

// The example displays the following output:

// Thread 'Sleeping' about to sleep indefinitely.

// Thread 'Sleeping' awoken.

// Thread 'Sleeping' executing finally block.

// Thread 'Sleeping finishing normal execution.

//

// Thread 'Sleeping2' about to sleep indefinitely.

// Thread 'Sleeping2' aborted.

// Thread 'Sleeping2' executing finally block.

1.7.2 Прерывание потоков

Ожидающий поток можно прервать, вызвав метод Thread.Interrupt для заблокированного потока. Это действие создает исключение ThreadInterruptedException, которое выводит поток из вызова блокировки. Поток должен перехватить исключение ThreadInterruptedException и выполнить соответствующие действия для продолжения работы. Если поток пропускает исключение, среда выполнения перехватывает его и останавливает поток.

*Если целевой поток не заблокирован при вызове метода Thread.Interrupt, поток не прерывается до блокировки. Если поток никогда не блокируется, он может завершиться, не будучи прерванным.*

Если ожидание является управляемым, методы Thread.Interrupt и Thread.Abort незамедлительно выводят поток из спящего режима. Если ожидание является неуправляемым (как, например, вызов неуправляемого кода функции Win32 WaitForSingleObject), то методы Thread.Interrupt и Thread.Abort не могут управлять потоком, пока он не вернется в управляемый код или не вызовет управляемый код.

В управляемом коде это поведение выглядит следующим образом:

1. Thread.Interrupt выводит поток из состояния ожидания, в котором он может находиться, и приводит к созданию исключения ThreadInterruptedException в целевом потоке.
2. Thread.Abort выводит поток из состояния ожидания, в котором он может находиться, и приводит к созданию исключения ThreadAbortException в этом потоке.

**1.8 Планирование потоков**

Каждому потоку назначается определенный приоритет. Потокам, созданным в общеязыковой среде выполнения, изначально назначается приоритет ThreadPriority.Normal. Потоки, созданные за пределами среды выполнения, сохраняют приоритет, который они имели до входа в управляемую среду. Вы можете получить или задать приоритет для любого потока, используя свойство Thread.Priority.

Свойство Priority определяет, сколько времени на исполнение будет выделено потоку относительно других потоков того же процесса. Существует 5 градаций приоритета потока: Lowest, BelowNormal, Normal, AboveNormal, Highest.

static void Main(string[] args)

{

Thread thread1 = new Thread(mythread1);

Thread thread2 = new Thread(mythread2);

Thread thread3 = new Thread(mythread3);

thread1.Priority = ThreadPriority.Highest;

thread2.Priority = ThreadPriority.AboveNormal;

thread3.Priority = ThreadPriority.Lowest;

thread1.Start();

thread2.Start();

thread3.Start();

Console.ReadLine();

}

Потоки назначаются на выполнение с учетом их приоритетов. Несмотря на то, что потоки выполняются в среде выполнения, процессорное время им выделяет операционная система. Конкретный алгоритм планирования, который определяет порядок выполнения потоков, будет разным у каждой операционной системы. В некоторых операционных системах первым всегда выполняется поток с наивысшим приоритетом (из числа тех потоков, которые готовы к выполнению). Если доступны несколько потоков с одинаковым приоритетом, планировщик поочередно запускает все потоки с этим приоритетом, выделяя каждому фиксированное время для выполнения. Пока есть хоть один доступный для выполнения поток с более высоким приоритетом, ни один из потоков с более низким приоритетом не запускается. Когда не останется ни одного готового потока с этим приоритетом, планировщик назначает для выполнения потоки со следующим, более низким, приоритетом. Если в любой момент будет готов поток с более высоким приоритетом, поток с низким приоритетом сразу же прерывается и вместо него снова запускается поток с более высоким приоритетом. Кроме того, операционная система может динамически изменять приоритеты потоков при перемещении пользовательского интерфейса приложения на передний план или в фоновый режим. Другие операционные системы могут использовать другой алгоритм планирования.

**1.9 Удаление потоков**

1.9.1 Метод Thread.Abort

Модель совместной отмены используется для остановки потока. Иногда выполнить совместную отмену потока невозможно, так как он выполняет код сторонних производителей, не поддерживающий такую отмену. Метод Thread.Abort в .NET Framework можно использовать для принудительного завершения управляемого потока. При вызове Abort общеязыковая среда выполнения создает в целевом потоке исключение ThreadAbortException, которое целевой поток может перехватить. Метод Thread.Abort не поддерживается в .NET Core. Если необходимо принудительно завершить выполнение кода сторонних производителей в .NET Core, запустите его в отдельном процессе и воспользуйтесь Process.Kill.

*Если в потоке выполняется неуправляемый код при вызове его метода Abort, в среде выполнения он отмечается как ThreadState.AbortRequested. В этом случае исключение вызывается после возврата потока в управляемый код.*

После прерывания поток не перезапускается.

Метод Abort не приводит к немедленному прерыванию потока, так как целевой поток может перехватить ThreadAbortException и выполнить любой объем кода в блоке finally. Если необходимо дождаться завершения потока, вы можете вызвать Thread.Join. Блокирующий вызов Thread.Join не завершится, пока поток полностью не закончит выполнение или не истечет указанный (необязательный) интервал времени ожидания. Прерванный поток может вызвать метод ResetAbort или выполнить любой объем операций в блоке finally, поэтому завершение ожидания не гарантируется, если вы не укажете время ожидания.

Потоки, ожидающие завершения метода Thread.Join, могут быть прерваны другими потоками, которые вызывают Thread.Interrupt.

1.9.2 Обработка ThreadAbortException

Если вы ожидаете, что поток будет прерван вызовом Abort из вашего кода или в результате выгрузки домена приложения, в котором выполняется поток (AppDomain.Unload использует Thread.Abort для прерывания потоков), в таком потоке необходимо обработать ThreadAbortException, включив в предложение finally все операции последней обработки, как показано в следующем примере.

try

{

// Code that is executing when the thread is aborted.

}

catch (ThreadAbortException ex)

{

// Clean-up code can go here.

// If there is no Finally clause, ThreadAbortException is

// re-thrown by the system at the end of the Catch clause.

}

// Do not put clean-up code here, because the exception

// is rethrown at the end of the Finally clause.

Код завершения работы следует поместить в предложении catch или finally, так как система повторно создаст исключение ThreadAbortException в конце предложения finally или catch, если отсутствует предложение finally.

Чтобы предотвратить повторное создание исключения, вы можете вызвать метод Thread.ResetAbort. Но этот вариант следует использовать только в том случае, если ThreadAbortException вызывается в пользовательском коде.

**Лабораторная работа №1 – Процессы и потоки в ОС Linux**

Перед защитой лабораторной работы необходимо подготовить архив, который содержит: проект, исполняемый файл, отчет по работе и скриншот программы (\*.jpg). Отчет по работе должен включать: титульный лист, разработку, пример работы программы и исходный код. Файл архива назвать <№ЛР\_Вариант\_ Фамилия\_группа>.

При защите лабораторной работы необходимо ответить на вопрос по процессу выполнения работы и/или на контрольный вопрос.

Цель лабораторной работы **–** освоение навыков работы с процессами и потоками ОС Linux.

Краткие теоретические сведения

При разработке для компьютеров как с одним, так и с несколькими процессорами, необходимо, чтобы приложение обеспечивало наиболее эффективное взаимодействие с пользователем, даже если в приложении выполняются другие задачи. Использование нескольких потоков выполнения — это один из способов обеспечить в приложении возможность реагирования на действия пользователя при одновременном использовании процессора для выполнения задач между появлением или даже во время появления событий пользователя.

Задание на лабораторную работу

Все варианты должны иметь графический интерфейс. Для каждого из вариантов необходимо реализовать возможность задавать приоритет каждого процесса и/или порожденных потоков, вручную пользователем. Процессы и потоки должны быть созданы явно.

Дополнительно позволить запуск из приложения собственного командного (аналог терминала Linux). Добавить по 10 команд для работы с процессами и потоками ОС, дать им пояснение. Например, команда терминала Linux – *find,* которая осуществляет поиск в файловой системе, файлах и папках. Cоздать аналог команды с именем *find.* Реализовать все команды на языке программирования.

**Вариант 1.** Посчитать количество символов. Взять два любых художественных произведений и посчитать в нем количество: букв, точек и запятых в ведение. Поиск по каждому условию осуществляется отдельным потоком. Книги обрабатываются в разных процессах.

**Вариант 2.** Умножение матрицы (10х10) на вектор (шестизначный). Обработка одной строки матрицы - в порожденном потоке. Транспонировать данную матрицу в новом процессе.

**Вариант 3.** Для 5-ти файлов (разного размера) требуется вычислить контрольную сумму (сумму кодов всех символов файла). Обработка каждого файла выполняется в отдельном потоке.

**Вариант 4.** Найти пересечение прямой с функции: параболой, кубической параболой, гиперболой, экспонентой и логарифмической функцией. Где, *а* – предпоследняя цифра зачетки, *b* – последняя цифра и *с* – первая цифра (если они присутствуют). Каждое решение выполнить в отдельном потоке.

**Вариант 5.** Заполнить три «сосуда» емкостью 1000мл. Каждый раз произвольно заполняются: 1 сосуд на 5мл, 2 сосуд на 7мл и 3 сосуд на 10мл. Каждый сосуд обрабатывает отдельный поток.

**Вариант 6.** Ввести трехзначное число и найти: его факториал и делитель (натуральное число, которое делит данное число без остатка). Вывести результаты на экран, и затем новыми процессами сохранить в отдельные файлы. Каждое действие выполнить отдельными потоками.

**Вариант 7.** Изменение заряда батареи телефона. Телефон имеет уровень заряда 100%. Один процесс содержит два потока, которые случайным образом уменьшают уровень заряда на 0,1%. Второй процесс содержит один поток исполнения и случайным образом увеличивает уровень заряда на 0,1%.

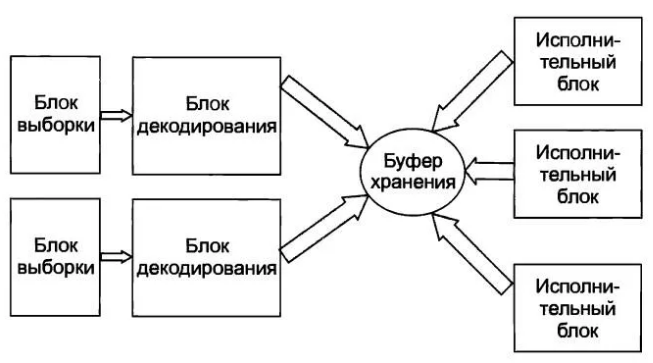
**Вариант 8.** Перевести случайное число (шестизначное) в двоичную, восьмеричную и шестнадцатеричную систему исчислений в каждом порождаемом потоке отдельно. С помощью нового процесса записать полученные значения в отдельные файлы.

**Вариант 9.** Расставить знаки операций и скобки так, чтобы равенства были верными: 4 4 4 4 = 8; 2 2 2 2 = 4; 5 5 5 5 = 26; 9 9 9 9 = 63. Каждое равенство обрабатывается отдельными процессами.

**Вариант 10.** Четыре команды спортсменов (4 отдельных потока) участвует в эстафете. Эстафета состоит из 4-х этапов забега по 1км. Каждый из спортсменов преодолевает по 10м, 9м, 8м и 7м за секунду соответственно. На каждом из этапов команды имеют бонус плюс 10*x* метров. Где, *x* – случайный номер приоритета потока, который не может повторяться в следующих этапах. Как только один из спортсменов доходит до конца этапа, забег останавливается и фиксируется проеденное расстояние. По нажатию кнопки «Старт» спортсмены бегут следующий этап с нулевой отметки. В конце забега сложить расстояния по каждому этапу и отобразить результаты команд. Каждую команду обрабатывает порождаемый поток.

**Вариант 11.** Имеется три рулетки, с числовым диапазоном от 0 до 100. Каждая рулетка выбрасывает одно случайное число в единицу времени. Интервалы времени задаются произвольными. У каждой рулетки должен быть свой уникальный генератор случайных чисел. Каждая рулетка представляет собой отдельный процесс.

**Вариант 12.** Решить задачу производитель – потребитель с точки зрения синхронизации по общему буферу. Производитель и потребитель представлены порожденными потоками. Размер буфера устанавливается, как количество минут от начала суток перемноженное на 2, расчет производится в новом процессе.

**В****ариант 13.** Смоделировать работу суперскалярного процессора.

Размер буфера хранения задается пользователем. Блоки выборки - 2 порожденных потока. Исполнительные блоки — 3 порожденных потока.

**Вариант 14.** Из любой системной папки OС Linux выбрать три любые подпапки. Подсчитать в них количество файлов и их суммарный объем. Вывести результаты в отдельные окна (формы). Произвести подсчет в каждой папке отдельным потоком.

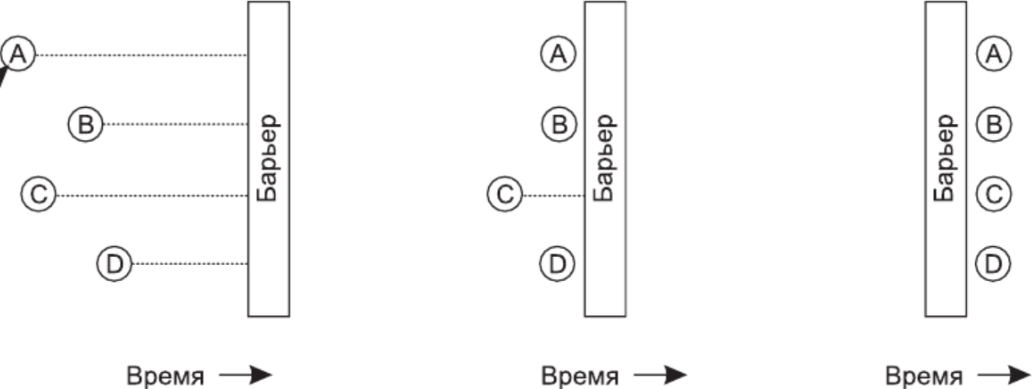
**Вариант 15.** Три путешественника идут по маршруту, протяженностью 5км. Каждый из них преодолевает 10*x,* 8*x* и 6*x* метров в секунду соответственно. Где, *x* – случайный номер приоритета отдельного потока, который меняется раз в 1км. Как только, один из путешественников доходит отметки кратной 500м, он останавливается и ждет остальных.

**Вариант 16.** Записать в текстовые поля: ФИО, специальность и название университета. Каждое слово записывать отдельным потоком. Текстовые поля заполняются параллельно, а слова в них последовательно.

**Вариант 17.** Найти заданное четырехбуквенное слово из набора символов русского и английского алфавита. Методом подбора и сравнения символов. Каждый символ определяется отдельным порожденным потоком.

**Вариант 18.** Зная IP адрес хоста и маску подсети определить: адрес сети, широковещательный адрес и количество действительных хостов. Для 5-ти IP адресов реализовать вычисления в отдельных потоках, каждый результат записать в свое новое окно (форму).

**Вариант 19.** Смоделировать работу барьера. Выполнение разбито на фазы и следуют правилу согласно которому ни один из потоков не может перейти к следующей фазе, пока все потоки не будут готовы перейти к следующей фазе. Добиться выполнения этого правила можно с помощью барьеров, поставленных в конце каждой фазы. Когда поток достигает барьера, он блокируется (ожидает) до тех пор, пока этого барьера не достигнут все остальные потоки. Имеется 4 порожденных потока. Расстояние от старта и между барьерами установить по 100 у.е.

**Вариант 20.** Подбор кодового слова. Создать 5 отдельных файлов, которые содержат по 500 слов на буквы А, Б, В, Г и Д соответственно. Необходимо определить кодовое слово из файлов со словами. Методом подбора и сравнения слов. Каждая база слов обрабатывается отдельным процессом и отдельными потоками.

Дополнительное задание

Добавить командный интерпретатор **Windows PowerShell/CMD** с соответствующим функционалом и аналогичными командами.

Контрольные вопросы

1. Что такое процесс?
2. Что такое поток?
3. Преимущества многопоточных приложений?
4. Создание и запуск нового потока?
5. Остановка потока?
6. Приостановка или прерывание потока?
7. Свойства потока?
8. Класс Thread?
9. Свойства для определения состояния потока?
10. Передача данных в потоки?
11. Извлечение данных из потоков с помощью методов обратного вызова?
12. Метод Thread.Sleep?
13. Прерывание потоков?
14. Планирование потоков?
15. Удаление потоков?
16. Отличие между Abort и Join?
17. Обработка ThreadAbortException?
18. Командные интерпретаторы для работы с процессами и потоками?