



MULTITHREADING IN C#/C++

Arctech Info

Multithreading in C#/C++



THREADS



THREADPOOL



TASKS

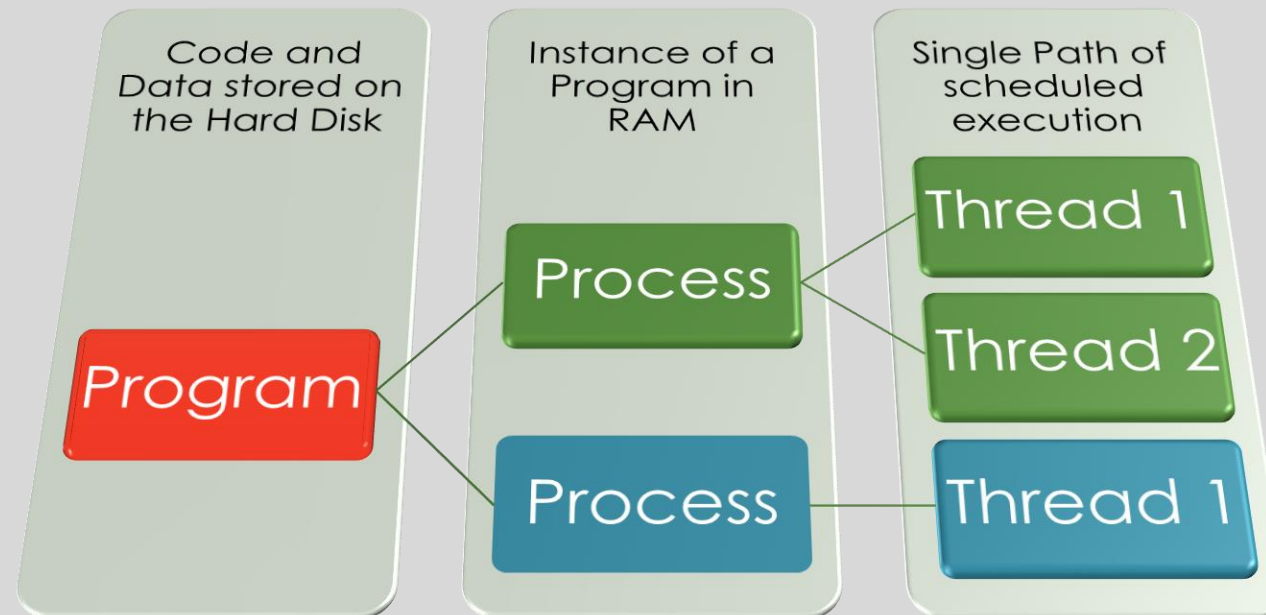
Multi-Tasking

- Multitasking is the simultaneous execution of multiple tasks or processes over a certain time interval.
- All modern operating systems (like Windows) implement multi-tasking.
- An Operating Systems
 - has its own programs files like boot program, the application manger, thread scheduler etc.
 - other Applications like Word, Excel, Notepad etc. (on Windows OS)
- An application consists of one or more processes
 - A process (in simple terms) is an executing program. See Task Manager in Windows
 - Every program that executes on your system is a process (or a collection of processes)
- Every single process can have one or more threads running in the context of the process.
- Note: Processes are heavier than threads
 - i.e. Processes have higher memory consumption and uses more battery power on a laptop

Some application examples

- Chrome supports multiple tabs via a multi-process architecture
 - 1 process per tab, so 10 tabs => 10 processes
 - Each process(tab) has 3 important threads and a few more threads
 - Main Thread
 - Allows users to interact with the tab, i.e. minimize, close, type a new url etc.
 - IO thread
 - Handles network communications.
 - Connects to the server and downloads the website
 - Renderer Thread
 - The downloaded website is a file in a specific format (HTML)
 - Parses the website file and displays it on the inside area of the Chrome Tab.
- Firefox supports multi-tabs via a multi-thread architecture
- Excel has 1 process per open instance/file (Most common architecture)
 - Each process has multiple threads to manage various functionality
 - E.g. Main Thread, formula calculation thread, etc.

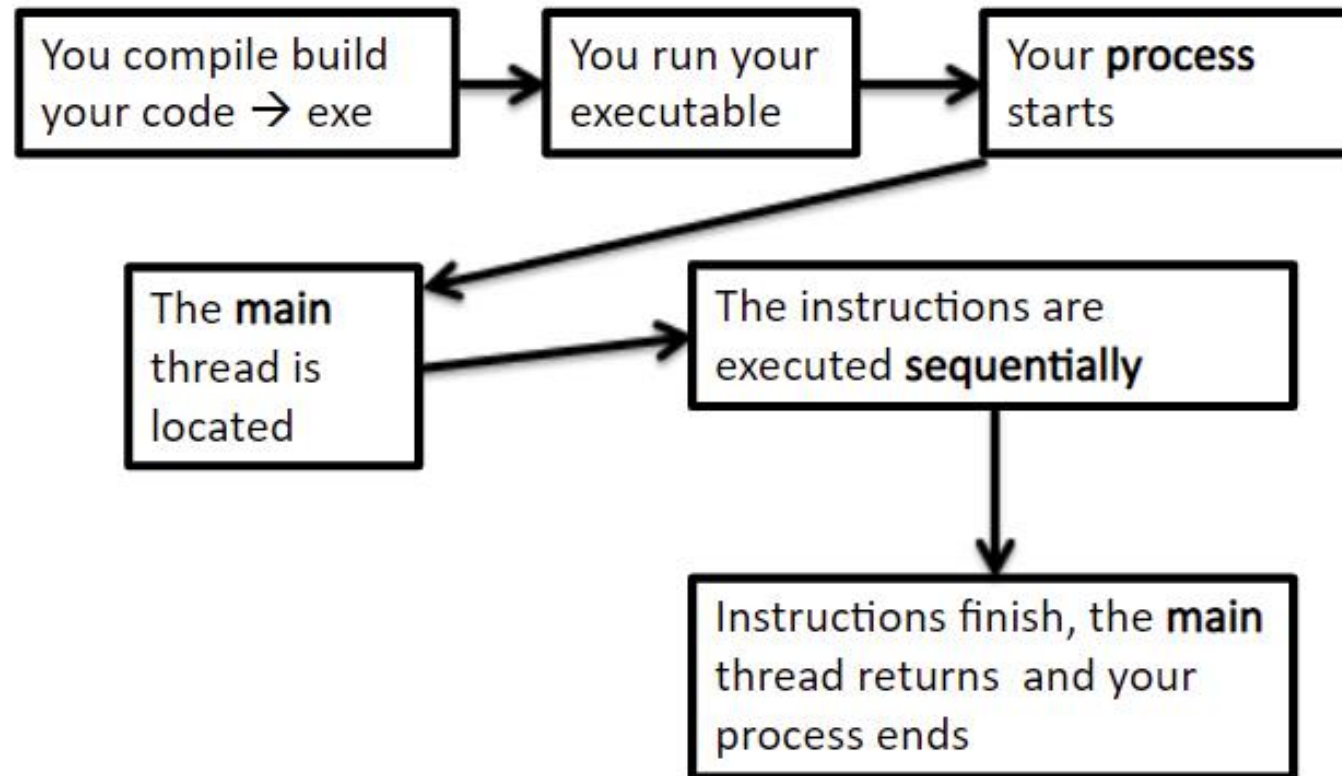
Processes & Threads



What is a Thread

- Basic unit of execution (A lightweight process)
- Managed by the Thread Scheduler which is part of a Operating System.
 - Is allocated processor time by the Operating System
- Every program has some logic, and a thread is responsible for executing this logic.
- Every program by default carries one thread to executes the logic of the program
 - the thread is known as the Main Thread,
- So, every program or application is by default single-threaded model.

A typical windows exe



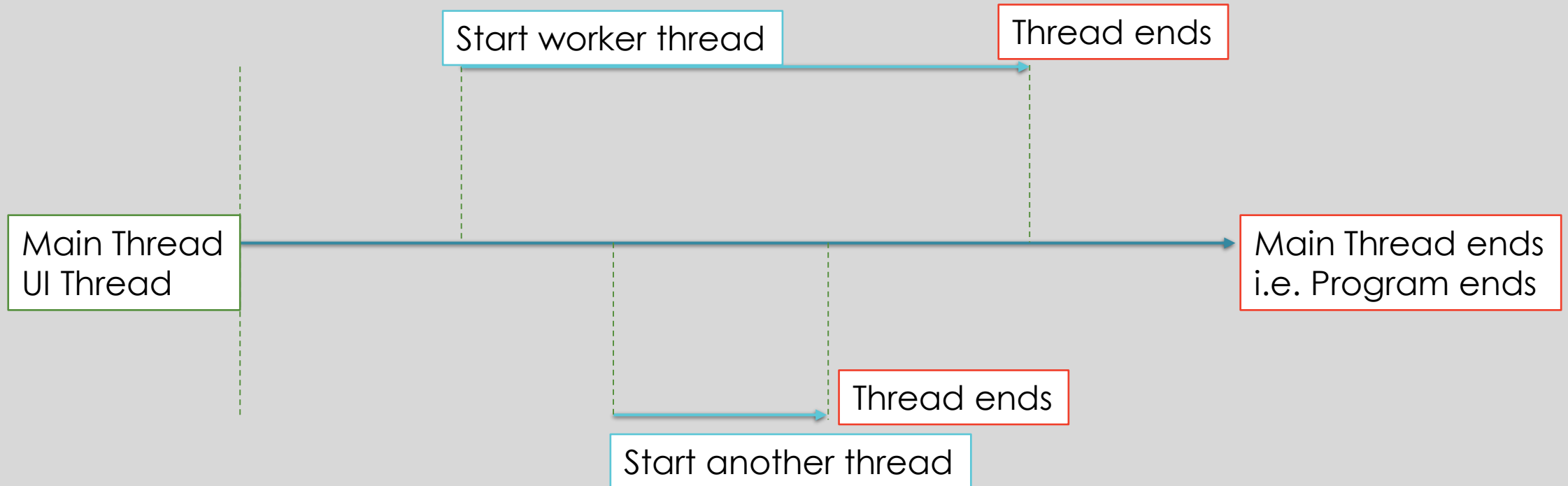
Drawbacks of Single Threaded Model

- A single thread runs all the instruction set (code blocks, methods, statements) present in the program in synchronized manner
 - one after another.
- The programs takes longer to run.
- For example, we have a class named as Animal and this class contains two different methods
 - Method1, Method2.
- Now the main thread is responsible for executing all these methods, so the main thread executes all these methods one by one.
 - The Main thread executes Method1 fully before executing Method2.

Multi-threading

- Multi-threading contains multiple threads within a single process.
- Here each thread performs different activities.
- For example, in the previous class, we can use multithreading, so each method is executed by a separate thread.
- The major advantage of multithreading is it works simultaneously
 - i.e. Multiple tasks can execute at the same time.
- This maximizes the utilization of the CPU because multithreading works on time-sharing concept.
- Each thread takes its own time for execution and does not affect the execution of another thread, this time interval is given by the operating system.

Main Thread and Worker Threads



Working with Threads in .NET

- Create and start a new thread
 - Create a new instance of the **System.Threading.Thread** class.
 - Provide the name of the method that you want to execute on a new thread to the constructor.
 - To start a created thread, call the **Thread.Start** method.

- Create and initialize two threads

```
Thread thread1 = new Thread(Show1);  
Thread thread2 = new Thread(Show2);
```

- Show1 & Show2 are methods

```
void Show1() {} & void Show2() {}
```

- Now start the execution of both the threads.

```
thread1.Start();  
thread2.Start();
```

Working with Threads in C++

- Create and start a new thread
 - Create a new instance of the `std::thread` class.
 - Provide the name of the method that you want to execute on a new thread to the constructor.

- Create and start two threads

```
#include <thread>
using namespace std;
thread thread1(Show1);
thread thread1(Show2);
```

- Show1 & Show2 are methods

```
void Show1() {} & void Show2() {}
```

Threads in C++

- Till C++ 11, there was no standard way to work with threads
- 3rd party libraries/framework were used.
- Every library or framework implemented threads in different ways
- Now we use the standard thread implementation in the STL.

```
#include <iostream>
#include <thread>
using namespace std;

void hello()
{
    cout << "Hello thread\n";
}

int main()
{
    thread aThread(&hello);
    aThread.join();
    cout << "Bye main\n";
    return 0;
}
```

standard C++ header to use threads

Later, this function will be the entry point (starting point) of aThread

The **main** thread starts here

aThread starts (is spawned) here.
Parent thread: main
Child thread: aThread

Every thread has to have an initial function where the new thread of execution begins. The new thread is started by constructing **aThread** object that specifies the task **hello()** to run on that thread.

```
Hello thread
Bye main
```

Review Example

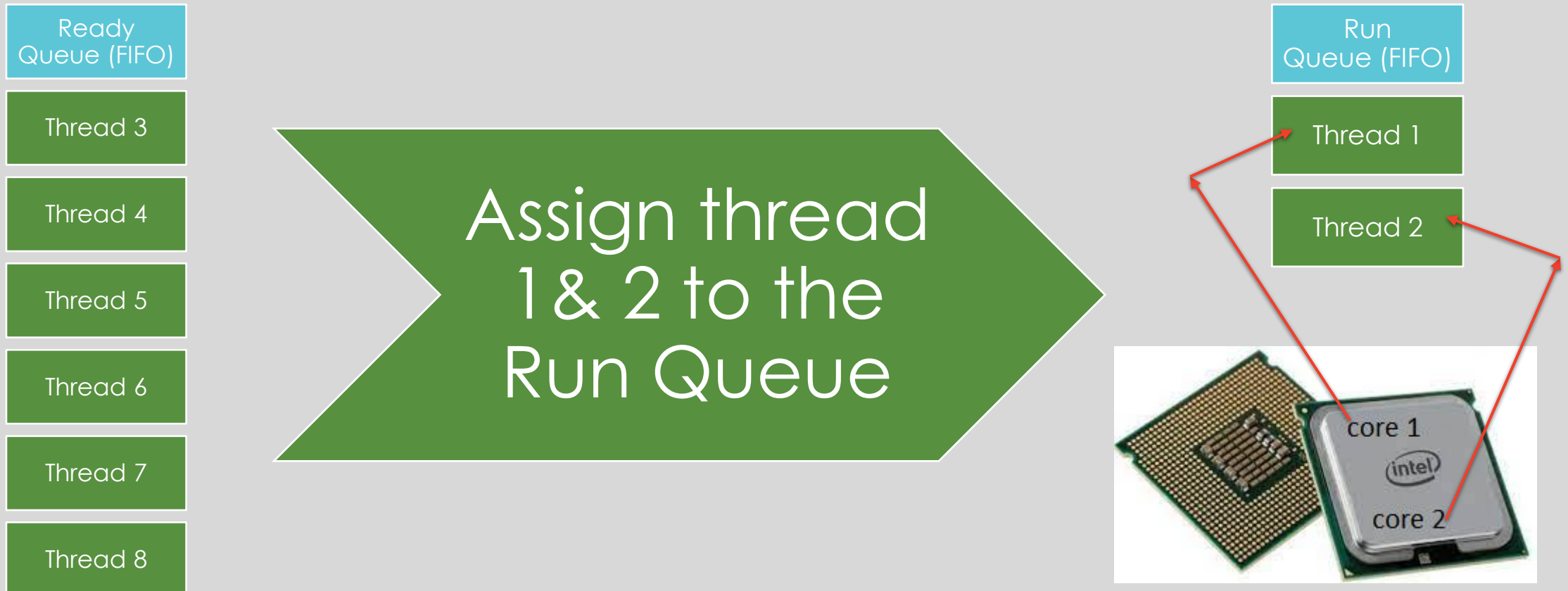
- Both thread runs simultaneously and the processing of thread2 does not depend upon the processing of thread1 like in the single threaded model.
- **Note:** Output may vary due to context switching.
- Advantages of Multithreading:
 - It executes multiple code sequences simultaneously.
 - Maximize the utilization of CPU resources.
 - Time sharing between multiple process.

OS Scheduler - Allocate CPU time

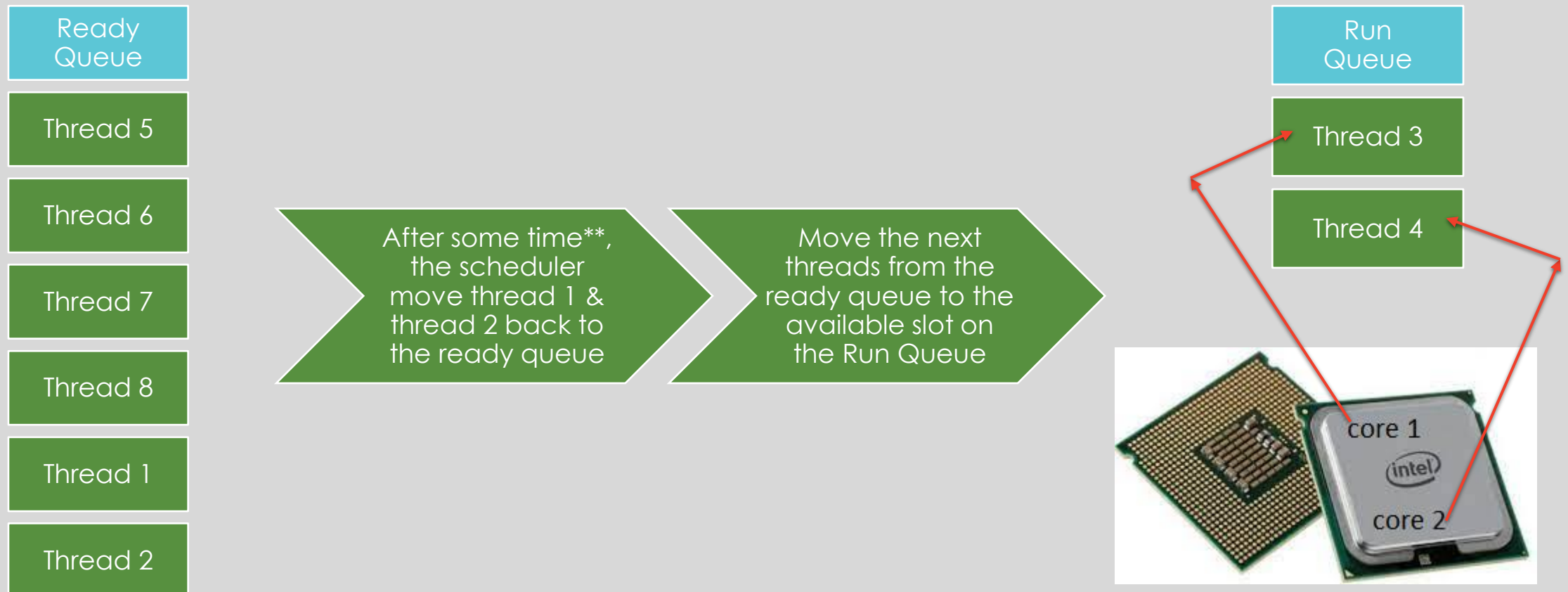
- Single Processor vs Multi-Processor
 - Single processor Core
 - Only 1 thread can run at one time
 - Multiple cores e.g. 4 cores
 - 4 threads can run simultaneously
- OS Scheduler & multi-tasking
 - Allows unlimited threads (theoretically) to run simultaneously**
 - TimeSlicing
 - Rapidly switching execution between all active threads.
 - In Windows, timeslicing for each thread typically 10-20 milliseconds.
 - When a thread is interrupted by timeslicing, it is said to be Pre-empted.
 - The CPU is no longer executing a pre-empted thread
 - The OS scheduler will keep running and pre-empting threads, over and over again
 - A Thread itself has no control over when it is pre-empted.



FIFO Thread Scheduler – 2 core CPU



FIFO Thread Scheduler – 2 core CPU



FIFO Thread Scheduler – 2 core CPU

Ready Queue

Thread 7

Thread 8

Thread 1

Thread 2

Thread 3

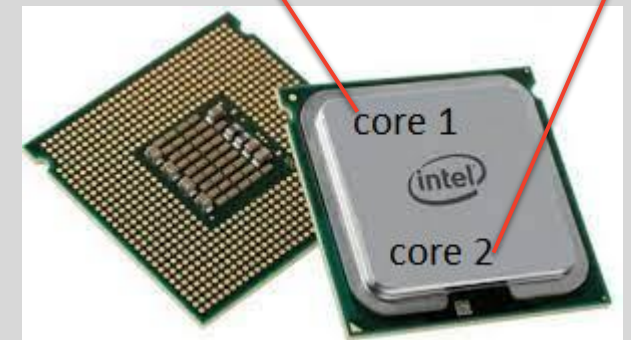
Thread 4

This continues in a FIFO, round-robin fashion. Some Threads may be killed from the ready queue and new threads may be created.

Run Queue

Thread 5

Thread 6



Thread States

- Ready (**Ready Queue**)
 - Ready to run
- Executing (**Run Queue**)
 - Running
- Blocked
 - Waiting for an event
- Ended
 - The thread is no longer running

Priority Scheduler vs FIFO Scheduler

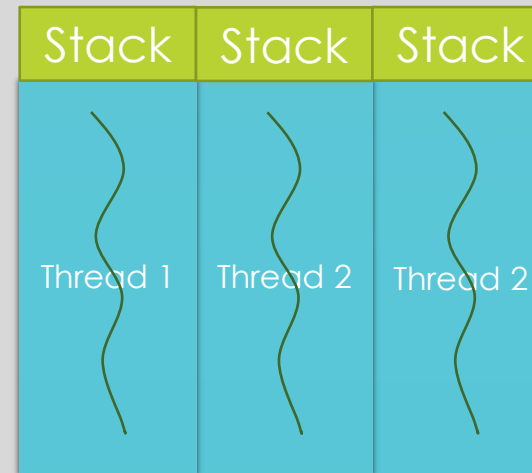
- Priority Scheduler is more common than FIFO
- Similar concept as the FIFO scheduling
- However, each thread has a priority associated with it
- The Scheduler places the high priority threads higher up in the **Ready Queue**
- It also leaves the high priority threads on the **Run Queue** for longer time.
- This ensures, the CPU starts working on the **high priority threads** 1st and also executes their instructions longer, than the **lower priority threads**

Local variables in Multiple Threading

- You can have multiple threads executing the same method.
- CLR assigns each thread its own local memory stack, to keep local variables separate.
- A separate copy of the local variables are created on that threads memory stack. See Example
- **Note:** All threads within a process share the same heap memory.



Single Threaded Process



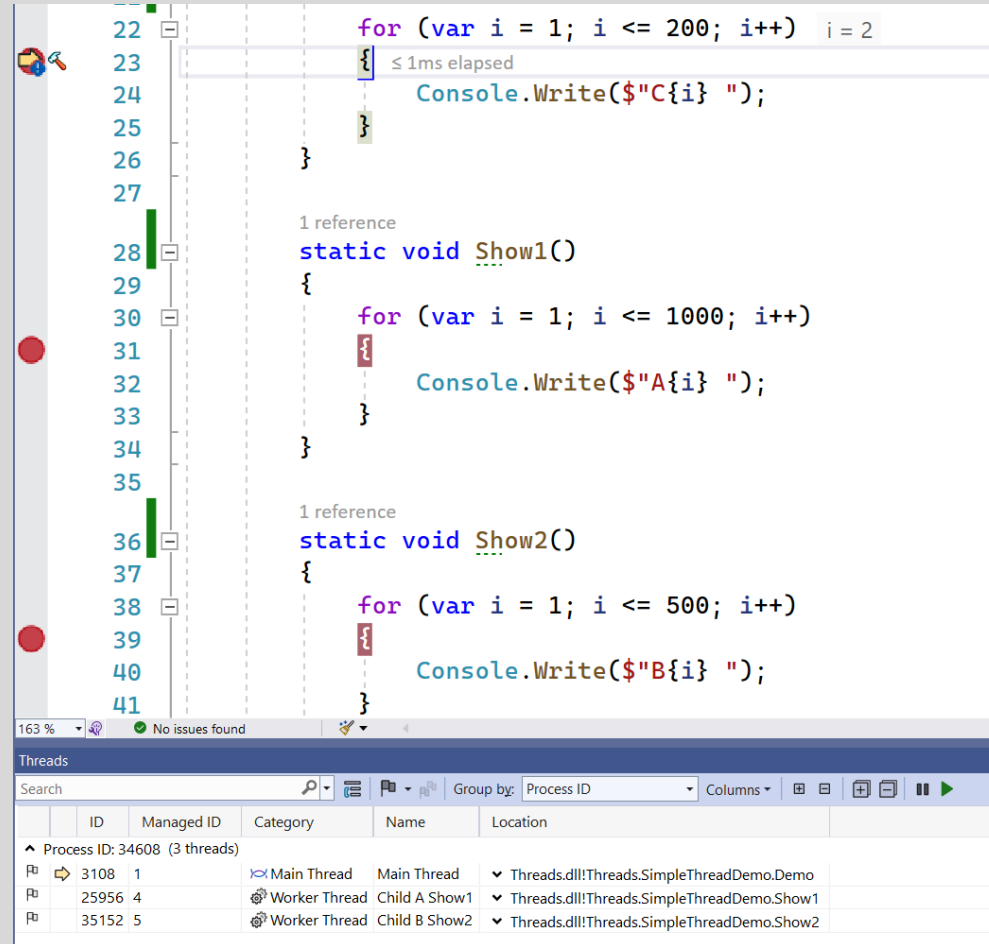
Multi Threaded Process

Foreground vs Background Threads

- Background threads are identical to foreground threads with one difference
- A background thread does not keep the application running.
- Once all foreground threads have stopped, the system abruptly terminates all background threads and shuts down the application.
- No Exceptions are thrown, and the application simply ends without waiting for the background thread to finish.
- In C++, by default `std::thread` is a background thread.
- In C#, by default `System.Threading.Thread` is a foreground thread. To create a background thread.
 - `Thread t1 = new Thread(fn);`
 - `t1.IsBackground = true;`

Debugging Threads in Visual Studio

- Add Breakpoints in thread method
- Debug -> Window -> Threads
- Set Thread Name



```
22 for (var i = 1; i <= 200; i++) i = 2
23 {
24     Console.WriteLine($"C{i} ");
25 }
26
27
28 1 reference
29 static void Show1()
30 {
31     for (var i = 1; i <= 1000; i++)
32     {
33         Console.WriteLine($"A{i} ");
34     }
35
36 1 reference
37 static void Show2()
38 {
39     for (var i = 1; i <= 500; i++)
40     {
41         Console.WriteLine($"B{i} ");
42     }
43 }
```

Threads

ID	Managed ID	Category	Name	Location
Process ID: 34608 (3 threads)				
3108	1	Main Thread	Main Thread	Threads.dll!Threads.SimpleThreadDemo.Demo
25956	4	Worker Thread	Child A Show1	Threads.dll!Threads.SimpleThreadDemo.Show1
35152	5	Worker Thread	Child B Show2	Threads.dll!Threads.SimpleThreadDemo.Show2

Access UI from Worker Thread

- You can only access UI controls from the UI thread - the main thread
- If you try to access the UI controls from any other thread, you will get a runtime error.
 - Sometimes the screen may hang or show unexpected results.
- To access UI controls from any thread, use `Control.Invoke` method
- This method moves the action back to the main thread.
 - i.e. the worker thread requests to UI thread to access the control and set its property
- E.g. if you want to change the text of a textbox from worker thread

- // Run time Error, or unexpected output

```
TextBoxName.Text = "Hello";
```

- // Correct way to access UI control from worker thread

```
{  
    TextBoxName.Invoke(new MethodInvoker(MyMethod1));  
}  
private void MyMethod1()  
{  
    TextBoxName.Text = "Hello";  
}
```

- As a good practice, use `if (Control.InvokeRequired)` before calling `Invoke`.

- // C# concise Syntax

```
TextBoxName.Invoke((MethodInvoker)delegate {  
    TextBoxName.Text = "Hello";  
});
```


Stopping or Pausing a Thread in .NET

- Stop a Thread
 - Co-operatively terminate the execution of a thread using the `System.Threading.CancellationTokenSource` class.
 - `Cancel()`
 - main thread sends a request to worker thread to gracefully terminate.
 - `IsCancellationRequested`
 - worker thread checks this property to see if someone has requested for thread termination.
 - Forcibly terminate the execution of a thread using `Thread.Abort`.
 - Causes a `ThreadAbortException` to be thrown on the thread
 - This does not work in .NET 5.
- Pause a Thread
 - The `Thread.Sleep` method is used to pause the current thread for a specified amount of time.
 - `Thread.Sleep(1000); // Pause current thread for 1 second.`
- Wait for all child Threads to complete
 - By default, the main thread does not exit till all foreground threads have completed.
 - However, if we want to wait anywhere else in code, use `Thread.Join()` ;

Shared Resources

- Threading seems to be simple in the beginning.
- However when you want to share resources between threads, we see issues.
- See example.
- Synchronizing threads to ensure thread safe access of shared resources.
- lock keyword

Thread Pool

- Creating a thread has some penalty in terms of time taken and memory used
- This overhead could be about few hundred milliseconds.
- This is the time taken in
 - created a new copy of the local stack and
 - starting the new thread
- Every thread also consumes about 1 MB RAM
- You can reduce this performance penalty by using Thread Pools

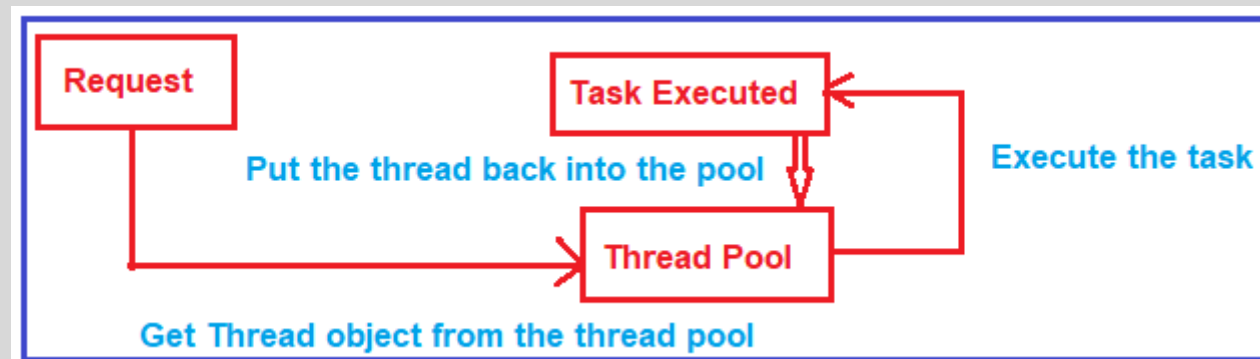
Thread Pool

- Too many active threads can throttle an operating system, with everything running very slowly
 - It can also freeze your system for sometime or even crash the CPU
- Thread Pools limits the number of threads that can run simultaneously.
- For e.g. If the Thread Pool count is 5, more than 5 threads cannot be run in parallel
- If your application creates more threads, the ThreadPool will queue these new Thread requests.
- The new Thread requests can only begin when an existing job finishes, effectively releasing the thread.
- See ThreadPoolDemo

Thread Pool



- Thread pool in C# is nothing but a collection of threads that can be reused to perform no of tasks in the background.
- Now when a request comes, then it directly goes to the thread pool and checks whether there are any free threads available or not.
- If available, then it takes the thread object from the thread pool and executes the task as shown in the below image.
- Once the thread completes its task then it again sent back to the thread pool so that it can reuse. This reusability avoids an application to create the number of threads, and this enables less memory consumption.



Asynchronous Programming with Tasks

- .NET introduced Asynchronous programming with C# version ver. 5.0 in visual studio 2012
- This was done using Tasks & Tasks<T> and the async await keywords.
- Asynchronous programming is not the same as multi-threaded programming.
- They are different programming models

Asynchronous Programming with Tasks

- Analogy of a Mr. X starting a Chinese restaurant & gets order of Noodles & Manchurian Gravy
 - Synchronous or Single Threaded
 - goes to kitchen, prepares gravy in wok, takes gravy to customer,
 - goes back to kitchen prepares noodles, takes noodles to customer.
 - Advantages: Customer will always get the order without any orders being stuck or forgotten.
 - Disadvantages: Order might be slow, the gravy might get cold, by the time the noodles is served, etc. Mr. X's time is blocked & cannot do anything else while the 1st order is in progress.
 - Multi Threaded
 - Here Mr. X works with 2 more people, a gravy chef and a noodles chef.
 - informs gravy chef to prepare gravy
 - informs noodles chef to prepare noodles
 - goes back to counter waiting for more orders, cleaning the tables, etc.
 - When burger chef finishes burger he gives it to Mr. X,
 - Fries chef finishes the Fries and gives it to Mr. X who then gives both to customer.
 - Advantages: Customer gets the burger and fries hot and fast.
 - Disadvantages: If gravy chef takes salt container and is waiting to take pepper container, while noodles chef takes pepper bottle and is waiting for the salt container to become available. Now if both do not talk with each other, they just keep waiting. So Mr. X has manage/synchronize both chefs access to common resources. If Mr. X does not synchronize properly, the customer does not get the food at all. Restaurant can shut down. It increases Mr. X's work and complexity.

Asynchronous Programming with Tasks

- Analogy of a Mr. X starting a Chinese restaurant & gets order of Noodles & Manchurian Gravy continued
 - Asynchronous
 - Goes to kitchen, puts the gravy ingredients in the Wok and keeps alarm for 5 minutes
 - goes to other Wok, puts in the noodles ingredients and keeps another alarm for 3 minutes.
 - goes back to counter waiting for more orders, cleaning the tables, etc.
 - When the 1st alarm sounds pauses his work to go pickup the noodles and put it in plate.
 - goes back to counter
 - When the 2nd alarm sounds pauses his work to go pickup the gravy and put it in plate.
 - Gives the plate to customer
 - goes back to counter

Tasks vs Threads

- It is always advised to use tasks instead of thread as it is created on the thread pool which has already system created threads to improve the performance.
- The task can return a result. There is no direct mechanism to return the result from a thread.
- A task can have multiple processes happening at the same time. Threads can only have one task running at a time.
- While using thread if we get the exception in the long running method it is not possible to catch the exception in the parent function but the same can be easily caught if we are using tasks.

Asynchronous Programming

- Methods containing a call to an asynchronous code have to follow the following rules
 - Add the **async** keyword before the method signature
 - Return **Task** or **Task<T>**
 - Add the **await** keyword before the call to the asynchronous method
- Example

```
public async Task<string> GetTextFromFile()  
{  
    var filePath = @"c:\files\test.txt";  
  
    var text = await File.ReadAllTextAsync(filePath);  
  
    return text;  
}
```

Asynchronous Sample

- Most libraries have asynchronous version of methods along with the regular synchronous methods.
- Use Async version of methods wherever possible.
- Create a class with sync and async version of methods to
 - Find number of characters in a file.
 - Sort all lines in a file
 - Get all text from file
- The class would be used as below.

```
var fileManager = new MyFileManager(@"C:\\File.txt");  
var count = await fileManager.CountCharacters();
```

```
var fileTextPreSort = await fileManager.GetAllText();
```

```
await fileManager.Sort();
```

```
var fileTextPostSort = await fileManager.GetAllText();
```

```
TextBoxMessage.Text = $"Count = {count}\n\nBefore Sort = {fileTextPostSort}\n\nAfter Sor = {fileTextPostSort}";
```

Asynchronous Programming guidelines

Name	Description	Exceptions
Avoid async void	Prefer async Task methods over async void methods	Event handlers
Async all the way	If a method is async, ensure all methods in the calling stack are also async	Console main method