Politechnika Łódzka
Wydział Elektrotechniki, Elektroniki, Informatyki i Automatyki

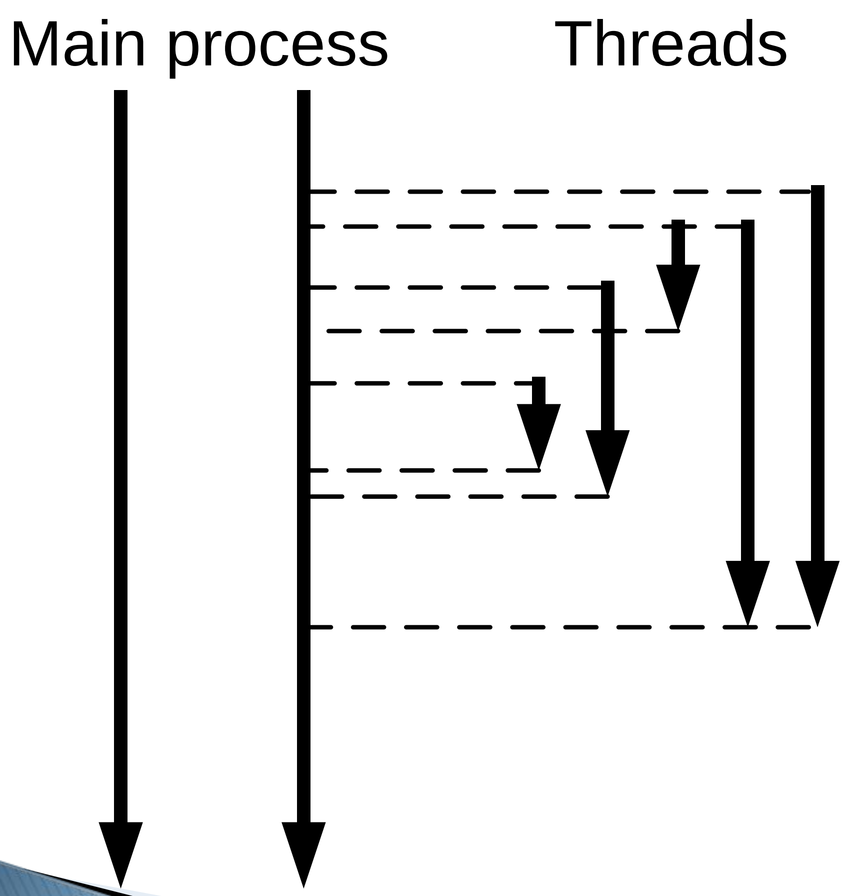
Network programming

Lecture 5
Multithreading and asynchronous calls

Instytut Informatyki Stosowanej
Politechniki Łódzkiej

dr inż. Radosław Wajman

Multitask concept



A multitasking operating system:

- divides the available processor time among the processes or threads that need it,
- it allocates a processor time slice to each thread it executes,
- suspends the currently executing thread when its time slice elapses, allowing another thread to run,
- switches from one thread to another saving the context of the preempted thread and restoring the saved context of the next thread in the queue,

Multitasking styles

Preemptive multitasking

- involves the use of an interrupt mechanism which suspends the currently executing process and invokes a scheduler to determine which process should execute next,
- therefore, all processes will get some amount of CPU time at any given time,
- the operating system kernel can also initiate a context switch to satisfy the scheduling policy's priority constraint.

Cooperative multitasking

- is a style of computer multitasking in which the operating system never initiates a context switch from a running process to another process.
- instead, processes voluntarily stop their operation periodically in order to enable multiple applications to be run simultaneously.

Main thread

When program starts one of its thread becomes alive.

It is a main thread.
- is able to initialize other threads,
- mainly it is a last thread finishing the program.

Besides, the main thread is like the normal thread.

In order to control it there is a need to have a handle

C – creating the thread

```
HANDLE WINAPI CreateThread(
    SECURITY_ATTRIBUTES lpThreadAttributes,
    SIZE_T dwStackSize,
    LPTHREAD_START_ROUTINE lpStartAddress,
    LPVOID lpParameter,
    DWORD dwCreationFlags,
    LPDWORD lpThreadId );
```

parameters:

- *lpThreadAttributes* – A pointer to a structure that determines whether the returned handle can be inherited by child processes. If lpThreadAttributes is NULL, the handle cannot be inherited.
- *dwStackSize* – The initial size of the stack, in bytes. Default 0.
- *lpStartAddress* – A pointer to the application-defined function to be executed by the thread. This pointer represents the starting address of the thread.
- *lpParameter* – A pointer to a variable to be passed to the thread
- *dwCreationFlags* – The flags that control the creation of the thread i.e. CREATE_SUSPENDED
- *lpThreadId* – A pointer to a variable that receives the thread identifier. If this parameter is NULL, the thread identifier is not returned.

Returned value is a handle to the new thread if succedeed. Otherwise returns NULL

Example

Code of this function is run in seperate thread is described in next slide

```
HANDLE th1, th2, th3, th4;
th1 = CreateThread(NULL, 0, my_thread, (void*)1, CREATE_SUSPENDED, NULL);
th2 = CreateThread(NULL, 0, my_thread, (void*)2, CREATE_SUSPENDED, NULL);
th3 = CreateThread(NULL, 0, my_thread, (void*)3, CREATE_SUSPENDED, NULL);
th4 = CreateThread(NULL, 0, my_thread, (void*)4, CREATE_SUSPENDED, NULL);
```

In the example the 4 treads are created and suspended.

All of them proceeds the same function but with different parameter value.

Calling **ResumeThread** function it is possible to resume on of them.

Thread procedure

Example:

```
DWORD WINAPI my_thread(LPVOID arg)
{
   int i = (int)arg;
   for (int j = 1; j < 10; j++)
   {
       Sleep(0);
       printf("%d", i);
   }

   return 0;
}
```

```
VOID WINAPI Sleep(
     DWORD dwMilliseconds);
```

The **`Sleep`** function suspends the current thread for *`dwMilliseconds`* milisekonds.

When the parameter value is **0** than the thread is switched to the next in scheduler queue.

`LPVOID` *`arg`* – is a pointer to a variable passed from **CreateThread**. function.

In this example it is an `int` value.

Terminating the thread

```
BOOL WINAPI TerminateThread(HANDLE hObject,
                            DWORD  dwExitCode);
```

Parameters:

- *hObject* – A handle to the thread to be terminated,
- *dwExitCode* – The exit code for the thread.

- If the target thread owns a critical section, the critical section will not be released,
- If the target thread is allocating memory from the heap, the heap lock will not be released,
- If the target thread is executing certain kernel32 calls when it is terminated, the kernel32 state for the thread's process could be inconsistent.
- If the target thread is manipulating the global state of a shared DLL, the state of the DLL could be destroyed, affecting other users of the DLL.

Suspending/resuming threads

```
BOOL WINAPI ResumeThread(HANDLE hThread);
```

- If a thread is created in a suspended state (with the CREATE_SUSPENDED flag), it does not begin to execute until another thread calls the ResumeThread function with a handle to the suspended thread.
- This can be useful for initializing the thread's state before it begins to execute.
- Suspending a thread at creation can be useful for one-time synchronization, because this ensures that the suspended thread will execute the starting point of its code when you call ResumeThread.

```
BOOL WINAPI SuspendThread(HANDLE hThread);
```

- The SuspendThread function is not intended to be used for thread synchronization because it does not control the point in the code at which the thread's execution is suspended.

```
th1 = CreateThread(NULL, 0, my_thread, (void*)1, CREATE_SUSPENDED,
   NULL);
ResumeThread(th1);
SuspendThread(th1);
```

Waiting for the thread

The basic method for thread synchronization is a function:

```
DWORD WINAPI WaitForSingleObject(
         HANDLE hHandle,
         DWORD dwMilliseconds);
```

Waits for `dwMilliseconds` for the thread terminating. If the *INFINITE* value is set, the function will return only when the object is signaled.

Result:

WAIT_OBJECT_0 – The state of the specified object is signaled (i.e. **`ExitThread`**)

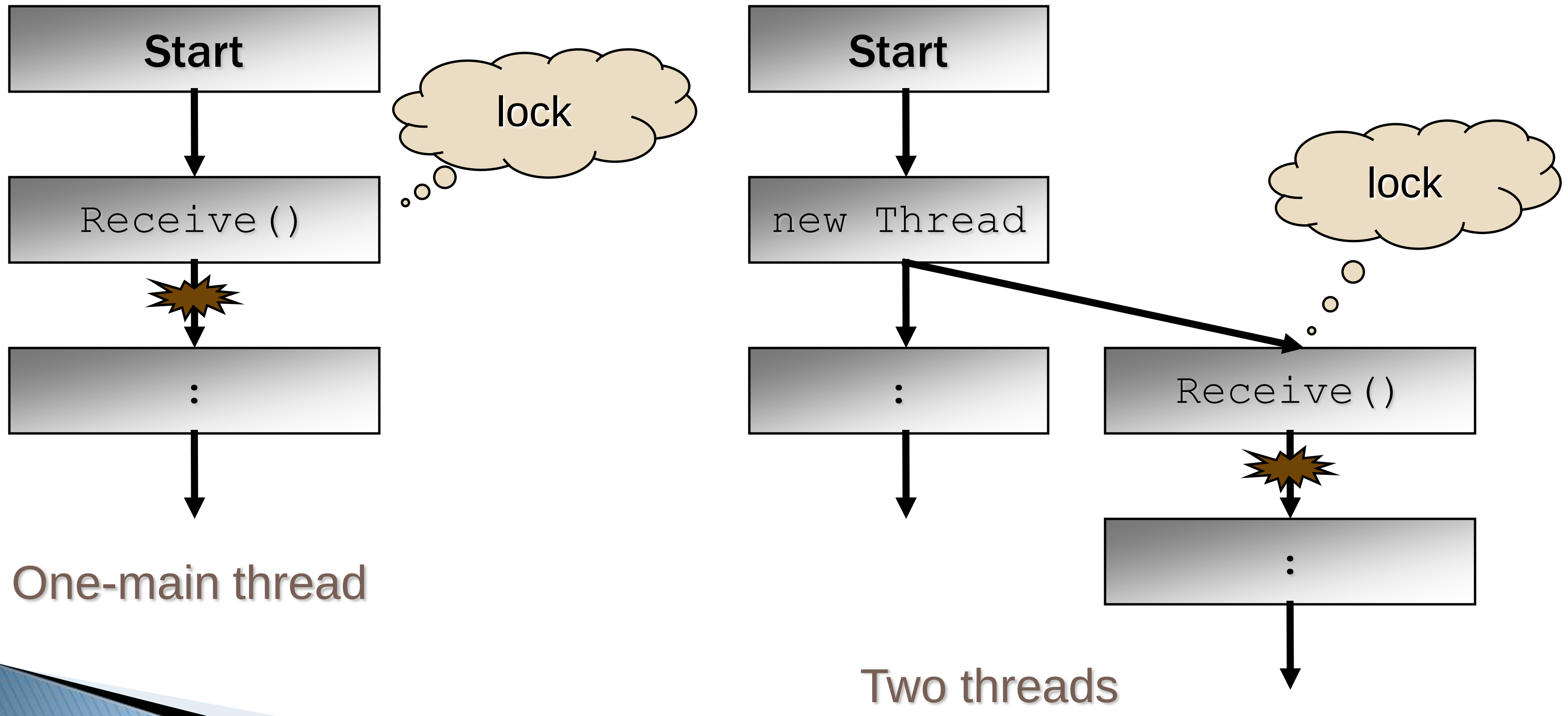
WAIT_TIMEOUT – The time-out interval elapsed, and the object's state is nonsignaled. (`dwMilliseconds`).

Example:

```
th1 = CreateThread(NULL, 0, my_thread, (void*)1, 0/*CREATE_SUSPENDED*/, NULL);
WaitForSingleObject(th1, INFINITE);
```

Multithreaded server

- Receive() blocks
  - waits for data



C# – .NET Framework
Creating threads

- System.Threading namespace

```
using System.Threading;
```

```
void ThreadProc()
{
    :
    while (true)
    {
        s.ReceiveFrom(...);
    }
}
void MainProc()
{
    :
    Thread t = new Thread(new ThreadStart(ThreadProc));
    t.Start();
    :
}
```

C# – running the main server thread

```
private TcpListener tcpListener;
private Thread listenThread;

public int Server()
{
 this.tcpListener = new TcpListener(IPAddress.Any, 7);
 this.listenThread = new Thread(new ThreadStart(ListenForClients));
 this.listenThread.Start();
 this.listenThread.Join();
 return 0;
}
```

The listening thread function

```
private void ListenForClients()
{
  try
       {
            this.tcpListener.Start();   //run listening
       }
       catch (Exception e)
       {
            this.SetTextOnListBox1("Server could not start. Error of binding");
            return;
       }
       this.SetTextOnListBox1("Server run on port 7");
       while (true)
       {
       //blocks until a client has connected to the server
            TcpClient client = this.tcpListener.AcceptTcpClient();
       this.SetTextOnListBox1("New client connected from: " + client.Client.RemoteEndPoint);

            //create a thread to handle communication with connected client
       Thread clientThread = new Thread(new ParameterizedThreadStart(HandleClientComm));
            clientThread.Start(client);
       }
}
```

The client service thread

```
private void HandleClientComm(object client)
{
  TcpClient tcpClient = (TcpClient)client;
  String clientIP = "" + tcpClient.Client.RemoteEndPoint.AddressFamily;
  NetworkStream clientStream = tcpClient.GetStream();

  int size = 1024;
  byte[] message = new byte[size];
  int bytesRead;
  ASCIIEncoding encoder = new ASCIIEncoding();

  while (true)    {
        bytesRead = 0;
        try {   //blocks until a client sends a message
             bytesRead = clientStream.Read(message, 0, size);
             clientStream.Write(message, 0, bytesRead);
        } catch  { break;   //a socket error has occured}

        if (bytesRead == 0) break;  //the client has disconnected from the server

        //message has successfully been received
        this.SetTextOnListBox1("  Message from " + tcpClient.Client.RemoteEndPoint + ": " +
                                          encoder.GetString(message, 0, bytesRead));
               //System.Diagnostics.Debug.WriteLine(encoder.GetString(message, 0, bytesRead));
        }
        tcpClient.Close();
  }
}
```

Thread-Safe Calls to Windows Forms Controls

```
// This event handler creates a thread that calls a
// Windows Forms control in an unsafe way.
delegate void SetTextCallback(string text);

private void SetTextOnListBox1(string text)
{
    // InvokeRequired required compares the thread ID of the
    // calling thread to the thread ID of the creating thread.
    // If these threads are different, it returns true.
    if (this.listBox1.InvokeRequired)
    {
        SetTextCallback d = new SetTextCallback(SetTextOnListBox1);
        this.Invoke(d, new object[] { text });
        //Executes the given delegation in a thread
        //which owns the control
    }
    else
    {
        this.listBox1.Items.Insert(0, text);
    }

}
```

More about delegations:
https://arvangen.wordpress.com/2011/08/27/c-var-delegacje-metody-anonimowe-i-wyrazenia-lambda/

Thread in JAVA

In order to start the new task in a thread there is a need to create a **Thread** object in two ways:

- Implement the **Runnable** interface,
- Extend the **Thread** class.

<u>Runnable</u>

The **Runnable** interface is an abstract way.

Implementing this it is possible to create the theead in basis of any object.

It is required only to implement the method:

```
public void run()
```

The body of this methos becomes the body of the thread.

The thread lives as long as the **run()** method.

<u>Creating the **Thread** object by constructor:</u>

```
Thread(Runnable Thread, String name);
```

The **start()** method executes the **run()** method of the thread object.

isAlive() and join() methodes

How one thread can get know if another thread signaled?

The answer is in two methoed of the **Thread** class:

```
final boolean isAlive()
```

returns **true**, when the thread, the method was called for, is still alive. Otherwise returns **false.**

```
final void join() throws InterruptedException
```

waits until the thread, the method was called for, is finished.

Overloads allow to control the time of waiting.

Threads synchronization

- **Critical section**
- **Semaphor**
- **Mutex** (*Mutual Exclusion*) – similar to the critical sections supporting the IPC (*Interprocess Communication*)

Critical section

The code area closed in a **critical section** can be executed by only one thread in the same time.

C# – Critical section

example 1st – without the CS

```
class Program
   {
       private static int a = 62;
       private static int b = 20;

       static void Main(string[] args)
       {
           for (int i = 0; i < 100; i++)
           {
               new Thread(Divide).Start();
           }
       }

       private static void Divide()
       {
           b = 23;
           if (b != 0)
           {
               Console.WriteLine(a / b);
           }
           b = 0;
       }
   }
```

C# – Critical section

```
class Program
{
    private static int a = 62;
    private static int b = 20;
    private static object o = new object();


    static void Main(string[] args) {
        for (int i = 0; i < 100; i++)
        {
            new Thread(Divide).Start();
        }
    }


    private static void Divide()  {
        lock (o)  {
            b = 23;
            if (b != 0)
            {
                Console.WriteLine(a / b);
            }
            b = 0;
        }
    }
}
```

API – Critical Section

```
CRITICAL_SECTION cs;

// .......

DWORD WINAPI my_thread(LPVOID arg)
{
    EnterCriticalSection(&cs);
    example code;
    LeaveCriticalSection(&cs);
}

int main(void)
{
    InitializeCriticalSection(&cs);
    HANDLE th1, th2, th3;
    th1 = CreateThread(NULL, 0, my_thread, (void*)1, 0, NULL);
    th2 = CreateThread(NULL, 0, my_thread, (void*)1, 0, NULL);
    th3 = CreateThread(NULL, 0, my_thread, (void*)1, 0, NULL);
    WaitForSingleObject(th1, INFINITE);
    WaitForSingleObject(th2, INFINITE);
    WaitForSingleObject(th3, INFINITE);
    DeleteCriticalSection(&cs);
}
```

Semaphor

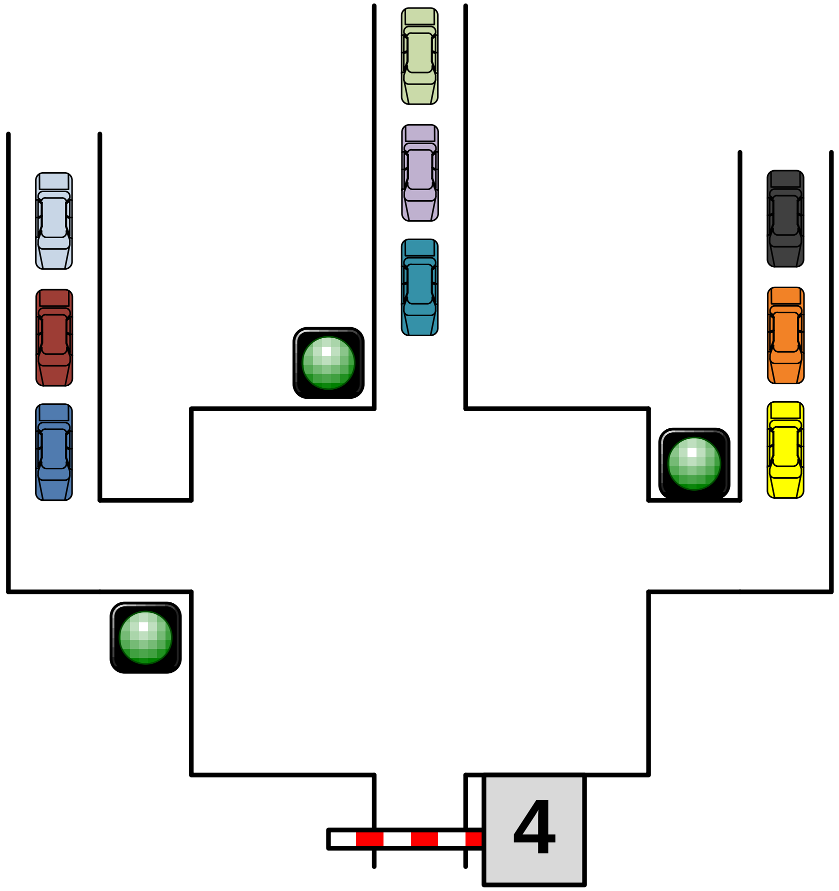
Semaphor is more sophisticated critical section.
Allows to determine the maximal thread number which can simultanously execute the given part of the code.

Each semaphore has assigned a counter.

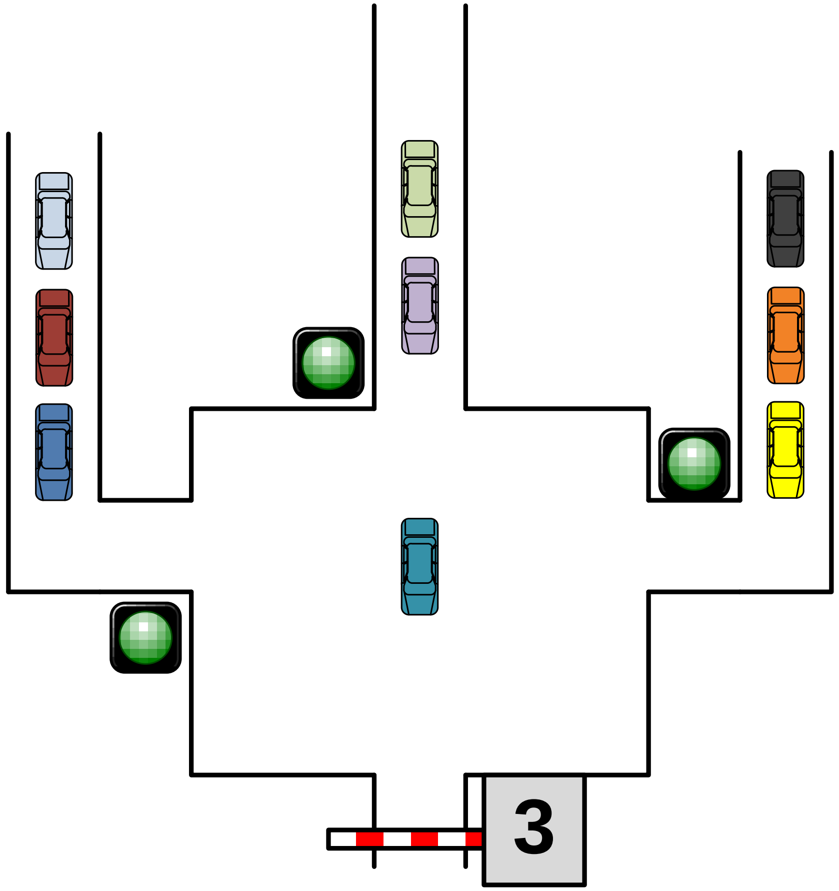
If the <u>counter value > 0</u>, then the semaphor **gives the access to execute**.

If the <u>counter value = 0</u>, then the semaphor **denies the access to execute.**

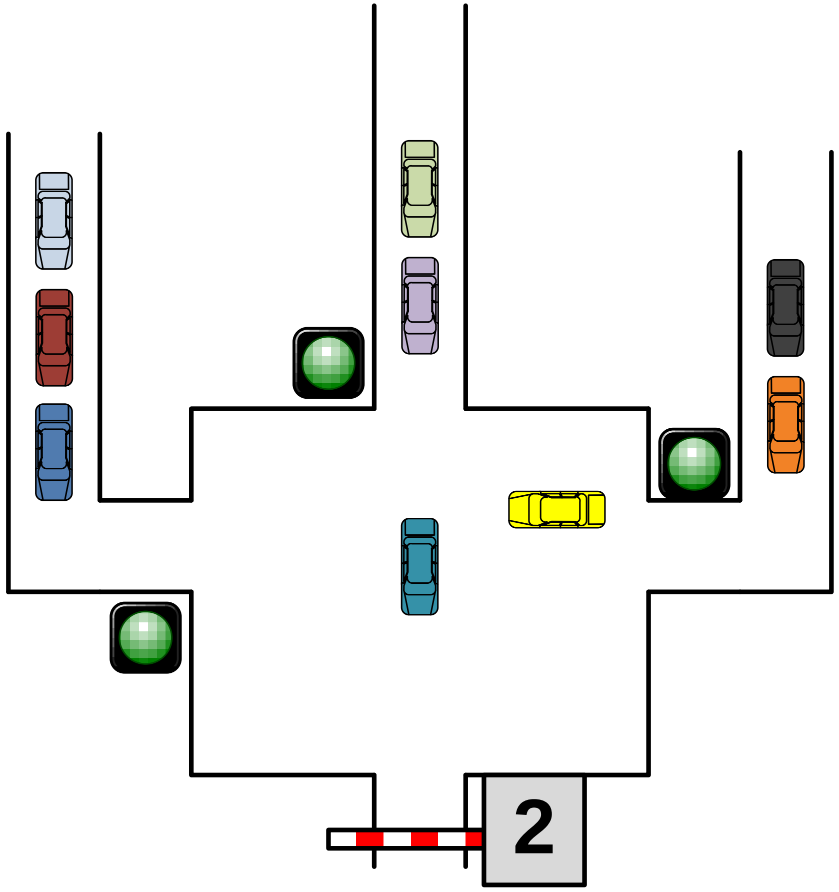
Semaphor – example



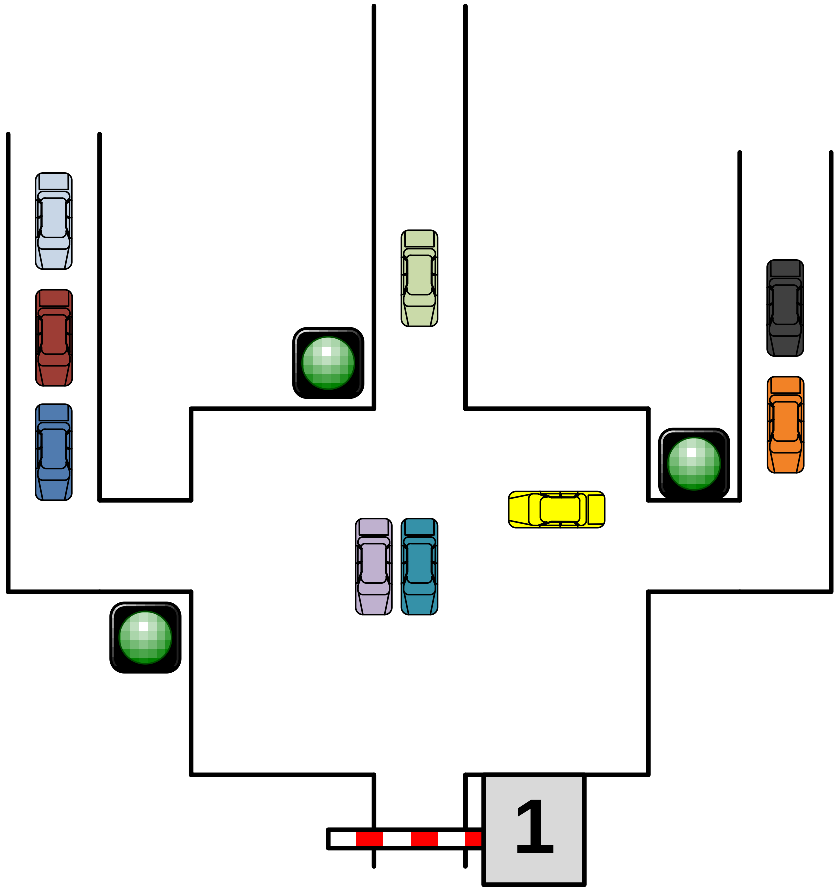
Semaphor – example



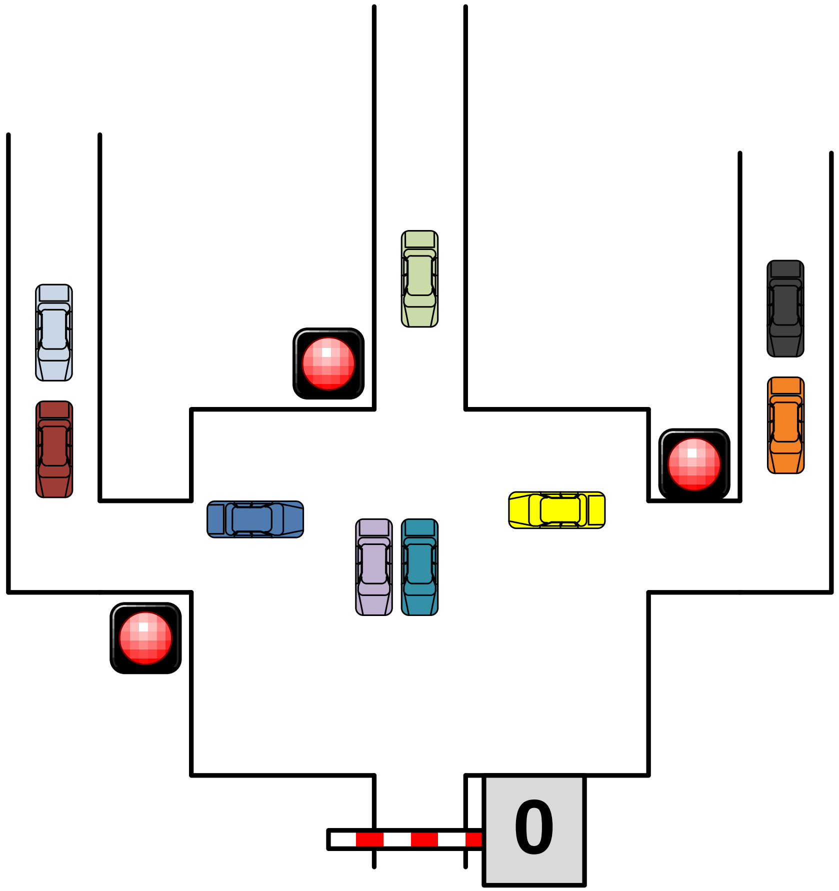
Semaphor – example



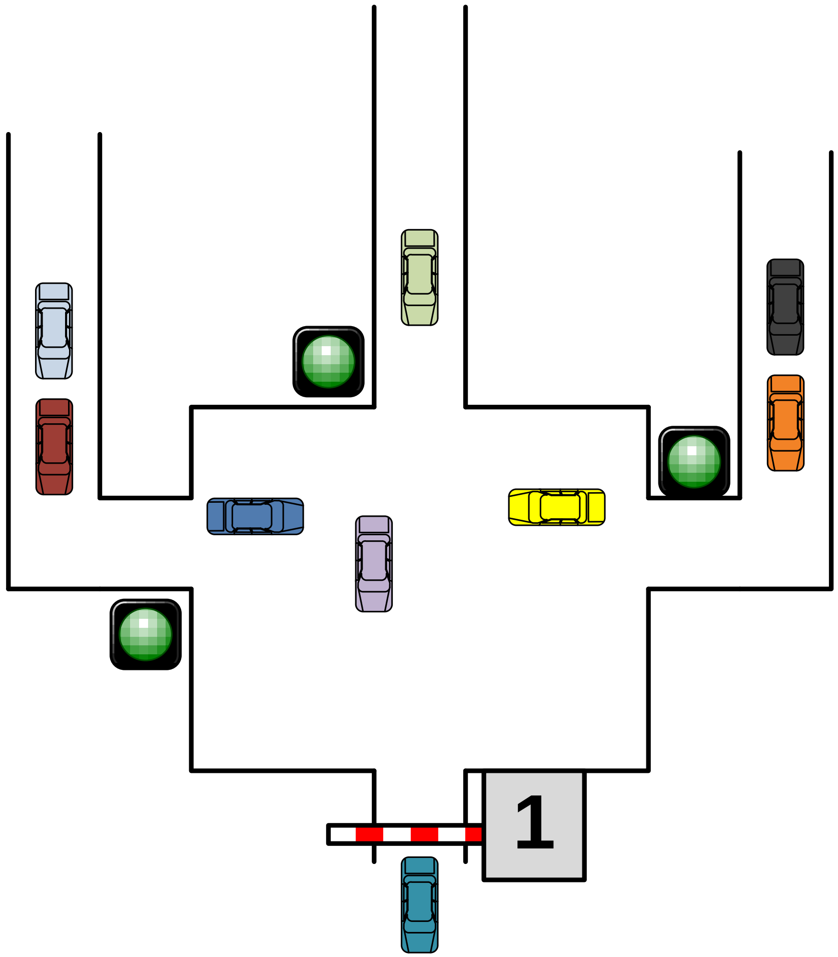
Semaphor – example



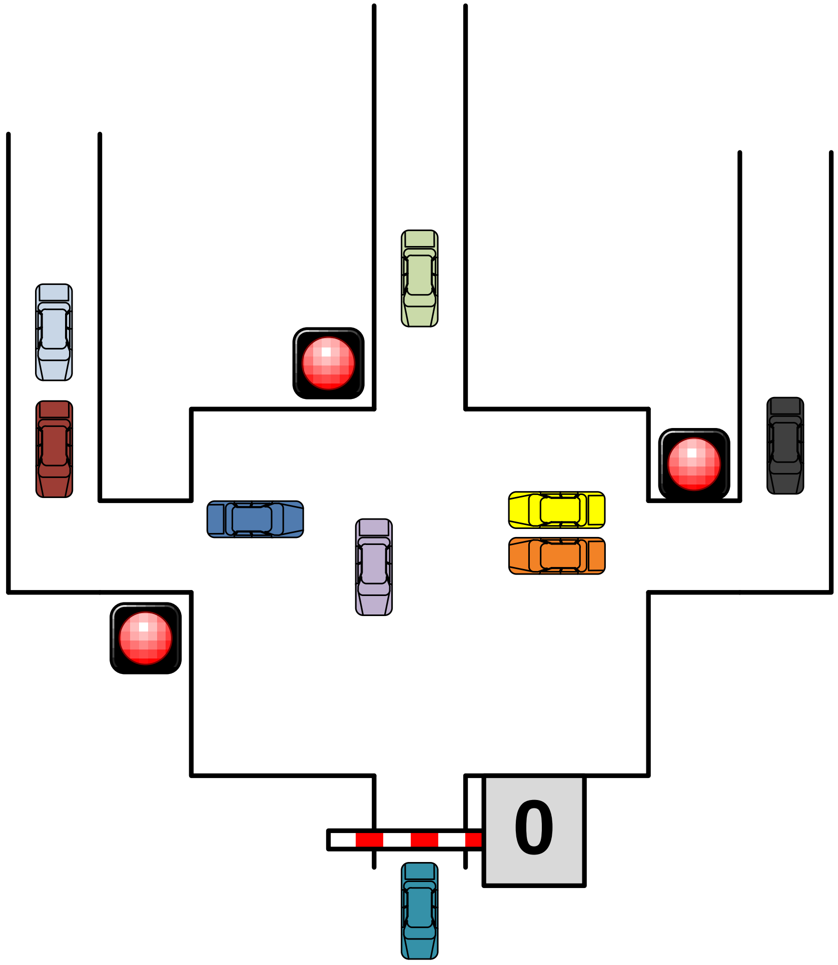
Semaphor – example



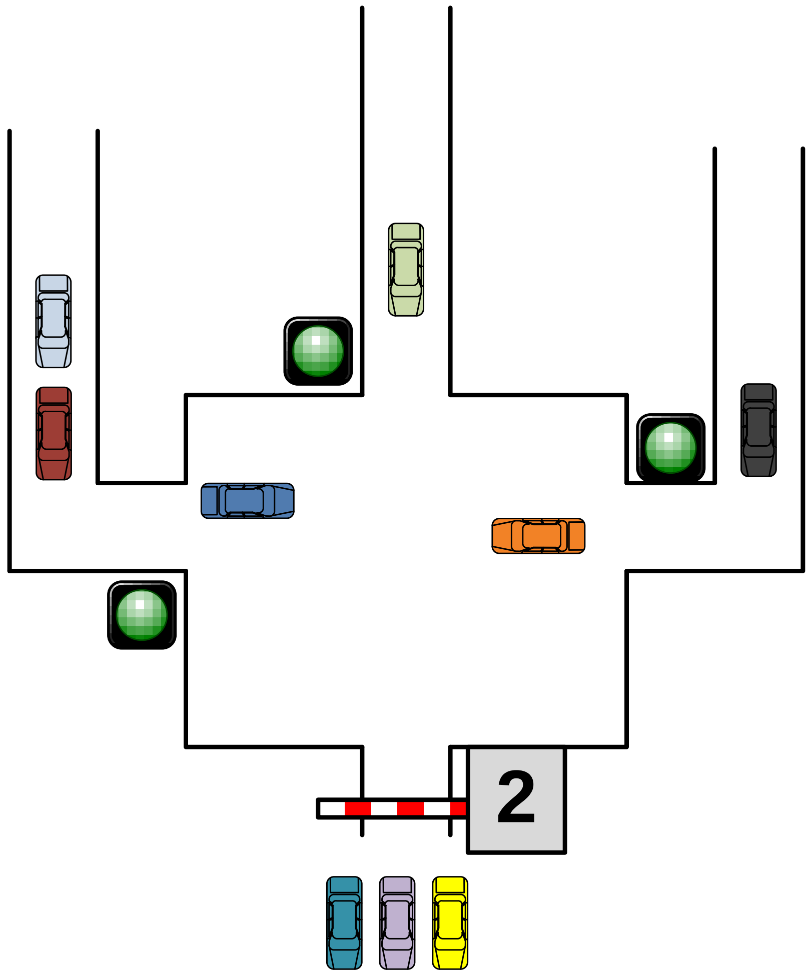
Semaphor – example



Semaphor – example



Semaphor – example



Semaphors API

Call the **CreateSemaphore** function to create semaphore.

```
HANDLE WINAPI CreateSemaphore(
    LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
    LONG lInitialCount,
    LONG lMaximumCount,
    LPCTSTR lpName);
```

Parametry:

- *lpSemaphoreAttributes* – A pointer to the structure with semaphore attributes. If this parameter is NULL, the handle cannot be inherited by child processes.
- *lInitialCount* – The initial count for the semaphore object. For the cars example it is **4.**
- *lMaximumCount* – The maximum count for the semaphore object. This value must be greater than zero.
- *lpName* – The name of the semaphore object limited to MAX_PATH characters. Name comparison is case sensitive. Default **NULL.**

To release the semaphore:

```
BOOL WINAPI CloseHandle(HANDLE hObject);
```

Semaphors –API

```
DWORD WINAPI WaitForSingleObject(
    HANDLE hSemaphore,
    DWORD dwMilliseconds);



BOOL WINAPI ReleaseSemaphore(
    HANDLE hSemaphore,
    LONG lReleaseCount,
    LPLONG lpPreviousCount);
```

Parameters:

- *hSemaphore* –semafhor handle,
- *dwMilliseconds* – time of waiting; default **INFINITE,**
- *lReleaseCount* – the value for counter increasing; default 1,
- *lpPreviousCount* – A pointer to a variable to receive the previous count for the semaphore.; default NULL.

Semafory – C#

```
class Program {
        private static Semaphore s = new Semaphore(2, 2);//init val.. 2, max count 2
        static void Main(string[] args)
        {
            for (int i = 0; i < 10; i++)
            {
                Thread t = new Thread(Run);
                t.Name = i.ToString();
                t.Start();
            }
        }
        public static void Run()
        {
            s.WaitOne();
            Console.WriteLine(Thread.CurrentThread.Name);
            Thread.Sleep(1000);
            s.Release();
        }
    }
```

The concept of asynchronous programming

- Why to use Asynchronous Programming?
  - You can avoid performance bottlenecks.
  - You can enhance the overall responsiveness of your application.

- Asynchrony is essential for activities that are potentially blocking (i.e. web access)
- If such an activity is blocked within a synchronous process, the entire application must wait.
- In an asynchronous process, the application can continue with other work that doesn't depend on the web resource until the potentially blocking task finishes.

Typical areas where asynchronous programming improves responsiveness

Application area	Supporting APIs that contain async methods
Web access	HttpClient, SyndicationClient
Working with files	StorageFile, StreamWriter, StreamReader, XmlReader
Working with images	MediaCapture, BitmapEncoder, BitmapDecoder
WCF programming	Synchronous and Asynchronous Operations

The concept of asynchronous programming

- Asynchrony proves especially valuable for applications that access the UI thread because all UI-related activity usually shares one thread.
- If any process is blocked in a synchronous application, all are blocked.
- Your application stops responding, and you might conclude that it has failed when instead it's just waiting.
- When you use asynchronous methods, the application continues to respond to the UI.
- You can resize or minimize a window, for example, or you can close the application if you don't want to wait for it to finish.

How to use it?

- The `async` and `await` keywords in C# are the heart of async programming.

- By using those two keywords, you can use resources in the .NET Framework or the Windows Runtime to create your own asynchronous method.

- In order to implement this you need to:
  - add an async modifier to the method,
  - set the return type to Task or Task<T>.
  - Finish the method name ends with "Async.„

```
async Task<int> AccessTheWebAsync()
```

How to use it?

```
async Task<int> AccessTheWebAsync()
{
    // You need to add a reference to System.Net.Http to declare client.
    HttpClient client = new HttpClient();

    // GetStringAsync returns a Task<string>. That means that when you await the
    // task you'll get a string (urlContents).
    Task<string> getStringTask = client.GetStringAsync("http://msdn.microsoft.com");

    // You can do work here that doesn't rely on the string from GetStringAsync.
    DoIndependentWork();

    // The await operator suspends AccessTheWebAsync.
    //  - AccessTheWebAsync can't continue until getStringTask is complete.
    //  - Meanwhile, control returns to the caller of AccessTheWebAsync.
    //  - Control resumes here when getStringTask is complete.
    //  - The await operator then retrieves the string result from getStringTask.
    string urlContents = await getStringTask;

    // The return statement specifies an integer result.
    // Any methods that are awaiting AccessTheWebAsync retrieve the length value.
    return urlContents.Length;
}
```

How to use it?

- If `AccessTheWebAsync` doesn't have any work that it can do between calling `GetStringAsync` and awaiting its completion, you can simplify your code by calling and awaiting in the following single statement.

```
string urlContents = await client.GetStringAsync();
```

Main features of the given example

- The method signature includes an ***Async*** or ***async*** modifier.

- The name of an async method, by convention, ends with an "Async" suffix.

- The return type is one of the following types:
  - `Task` if your method has a return statement in which the operand has type TResult.
  - `Task` if your method has no return statement or has a return statement with no operand.
  - `Void` if you're writing an async event handler.

- The method usually includes at least one await expression, which marks a point where the method can't continue until the awaited asynchronous operation is complete.

- In the meantime, the method is suspended, and control returns to the method's caller.

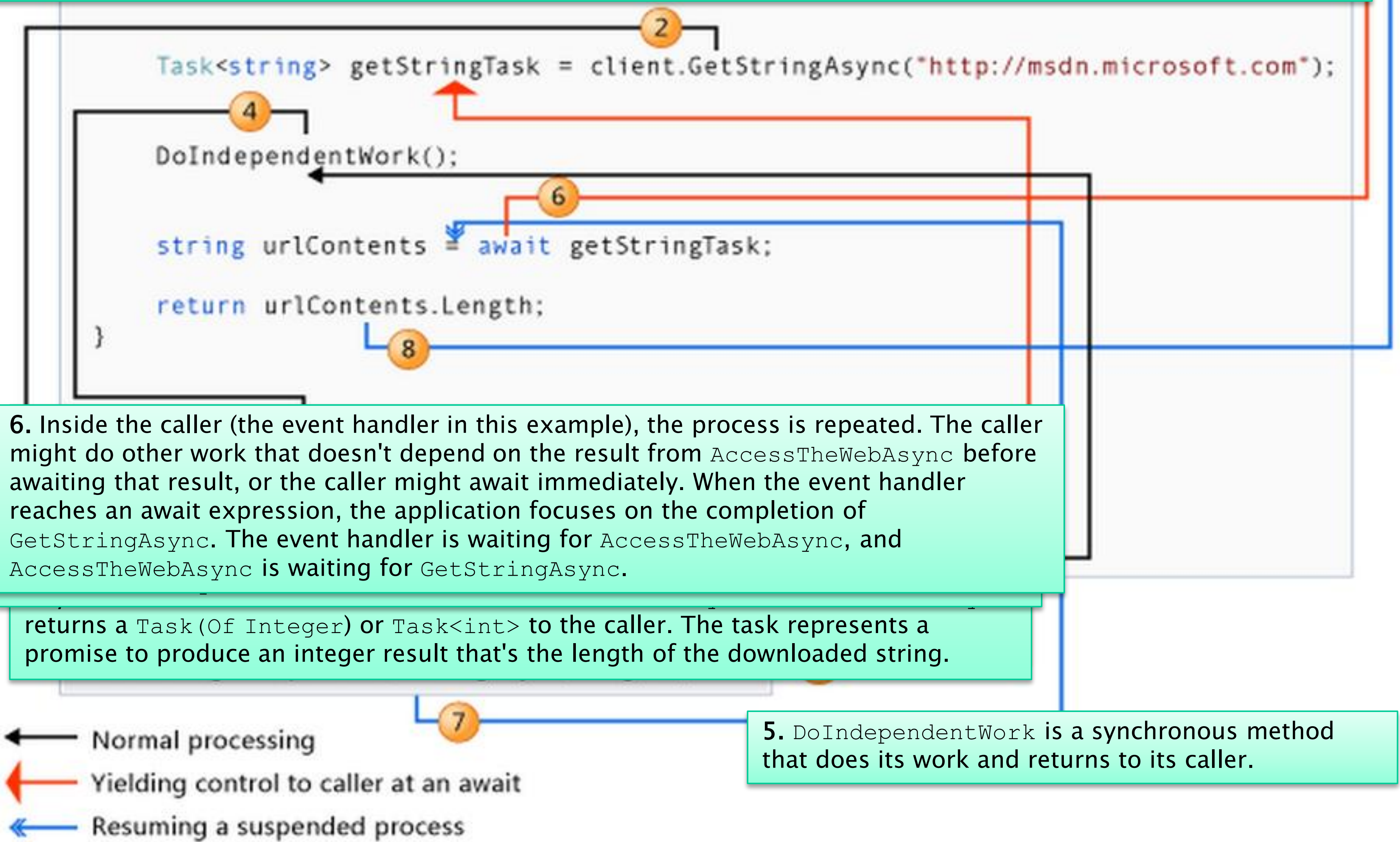
3. Something
download or
AccessTheWe

GetStringAs
getStringTa
commitment to produce an actual string value when the work is complete.

7. GetStringAsync completes and produces a string result. The string result isn't
re
(R
re
ge
assignment statement assigns the retrieved result to urlContents.

8. When AccessTheWebAsync has the string result, the method can calculate the length of the string. Then the work of AccessTheWebAsync is also complete, and the waiting event handler can resume. In the full example at the end of the topic, you can confirm that the event handler retrieves and prints the value of the length result.

hasn't been
n
't depend
sync.
the
ependentWork.



```
Task<string> getStringTask = client.GetStringAsync("http://msdn.microsoft.com");

DoIndependentWork();

string urlContents = await getStringTask;

return urlContents.Length;
}
```

2
4
6
8
7
Normal processing
Yielding control to caller at an await
Resuming a suspended process

6. Inside the caller (the event handler in this example), the process is repeated. The caller might do other work that doesn't depend on the result from AccessTheWebAsync before awaiting that result, or the caller might await immediately. When the event handler reaches an await expression, the application focuses on the completion of GetStringAsync. The event handler is waiting for AccessTheWebAsync, and AccessTheWebAsync is waiting for GetStringAsync.

returns a Task(Of Integer) or Task<int> to the caller. The task represents a promise to produce an integer result that's the length of the downloaded string.

5. DoIndependentWork is a synchronous method that does its work and returns to its caller.

Synchronous and asynchronous behavior

- A synchronous method returns when its work is complete (step 5),
- But an async method returns a task value when its work is suspended (steps 3 and 6).
- When the async method eventually completes its work, the task is marked as completed and the result, if any, is stored in the task.

API Async Methods

- The .NET Framework 4.5 contains many members that work with async and await. Thay may recognize by the "Async" suffix that's attached to the member name and a return type of Task or Task. For example, the `System.IO.Stream` class contains methods such as `CopyToAsync`, `ReadAsync`, and `WriteAsync` alongside the synchronous methods `CopyTo`, `Read`, and `Write`.

- The Windows Runtime also contains many methods that may be used with `async` and `await` in Windows Store apps.

Threads in async calls

- ***Async*** methods are intended to be non-blocking operations.
- An ***await*** expression in an ***async*** method doesn’t block the current thread while the awaited task is running.
- Instead, the expression signs up the rest of the method as a continuation and returns control to the caller of the ***async*** method.
- The ***async*** and ***await*** keywords don't cause additional threads to be created.
- Async methods don't require multithreading because an ***async*** method doesn't run on its own thread.
- The method runs on the current synchronization context and uses time on the thread only when the method is active.
- You can use `Task.Run` to move CPU-bound work to a background thread, but a background thread doesn't help with a process that's just waiting for results to become available.

Async and Await

- If you specify that a method is an ***async*** method by using an `async` modifier, you enable the following two capabilities.
  - The marked async method can use await to designate suspension points.
    The await operator tells the compiler that the async method can't continue past that point until the awaited asynchronous process is complete.
    In the meantime, control returns to the caller of the async method.
    The suspension of an async method at an await expression doesn't constitute an exit from the method, and finally blocks don't run.
  - The marked async method can itself be awaited by methods that call it.

- An async method typically contains one or more occurrences of an await operator, but the absence of await expressions doesn't cause a compiler error.

- If an async method doesn't use an await operator to mark a suspension point, the method executes as a synchronous method does, despite the async modifier. The compiler issues a warning for such methods.

Return types and parameters

- In .NET Framework programming, an `async` method typically returns a `Task`.
- Inside an `async` method, an await operator is applied to a task that's returned from a call to another `async` method.
- You specify Task as the return type if the method contains `return` (C#) statement that specifies an operand of type TResult.
- You use `Task` as the return type if the method has no return statement or has a return statement that doesn't return an operand.

How you declare and call a method that returns a Task

```
// Signature specifies Task<TResult>
async Task<int> TaskOfTResult_MethodAsync()
{
    int hours;
    // . . .
    // The body of the method should contain one or more await expressions.

    // Return statement specifies an integer result.
    return hours;
}

    // Calls to TaskOfTResult_MethodAsync from another async method.
private async void CallTaskTButton_Click(object sender, RoutedEventArgs e)
{
    Task<int> returnedTaskTResult = TaskOfTResult_MethodAsync();
    int intResult = await returnedTaskTResult;
    // or, in a single statement
    //int intResult = await TaskOfTResult_MethodAsync();
}
```

How you declare and call a method that returns a Task

```
// Signature specifies Task
async Task Task_MethodAsync()
{
    // . . .
    // The body of the method should contain one or more await expressions.

    // The method has no return statement.
}

    // Calls to Task_MethodAsync from another async method.
private async void CallTaskButton_Click(object sender, RoutedEventArgs e)
{
    Task returnedTask = Task_MethodAsync();
    await returnedTask;
    // or, in a single statement
    //await Task_MethodAsync();
}
```

Remarks to the examples

- Each returned task represents ongoing work. A task encapsulates information about the state of the asynchronous process and, eventually, either the final result from the process or the exception that the process raises if it doesn't succeed.
- An async method can also have a void return type (C#). This return type is used primarily to define event handlers, where a void return type is required. Async event handlers often serve as the starting point for async programs.
- An async method that has a void return type can’t be awaited, and the caller of a void-returning method can't catch any exceptions that the method throws.
- An async method can't declare ref or out parameters in C#, but the method can call methods that have such parameters.

Remarks to the examples

- Asynchronous APIs in Windows Runtime programming have one of the following return types, which are similar to tasks:
  - IAsyncOperation<TResult>, which corresponds to Task<TResult>
  - IAsyncAction, which corresponds to Task
  - IAsyncActionWithProgress
  - IAsyncOperationWithProgress

Using the thread pool in Windows Runtime

- To do parallel work in your app you can also use the thread pool.
- Your app can use the thread pool to accomplish work asynchronously in parallel threads.
- The thread pool manages a set of threads and uses a queue to assign work items to threads as they become available.
- The thread pool is similar to the asynchronous programming patterns available in the Windows Runtime because it can be used to accomplish extended work without blocking the UI, but the thread pool offers more control than the asynchronous programming patterns and you can use it to complete multiple work items in parallel.

Using the thread pool in Windows Runtime

- The ThreadPool may be used to:
  - Submit work items, control their priority, and cancel work items.
  - Schedule work items using timers and periodic timers.
  - Set aside resources for critical work items.
  - Run work items in response to named events and semaphores.

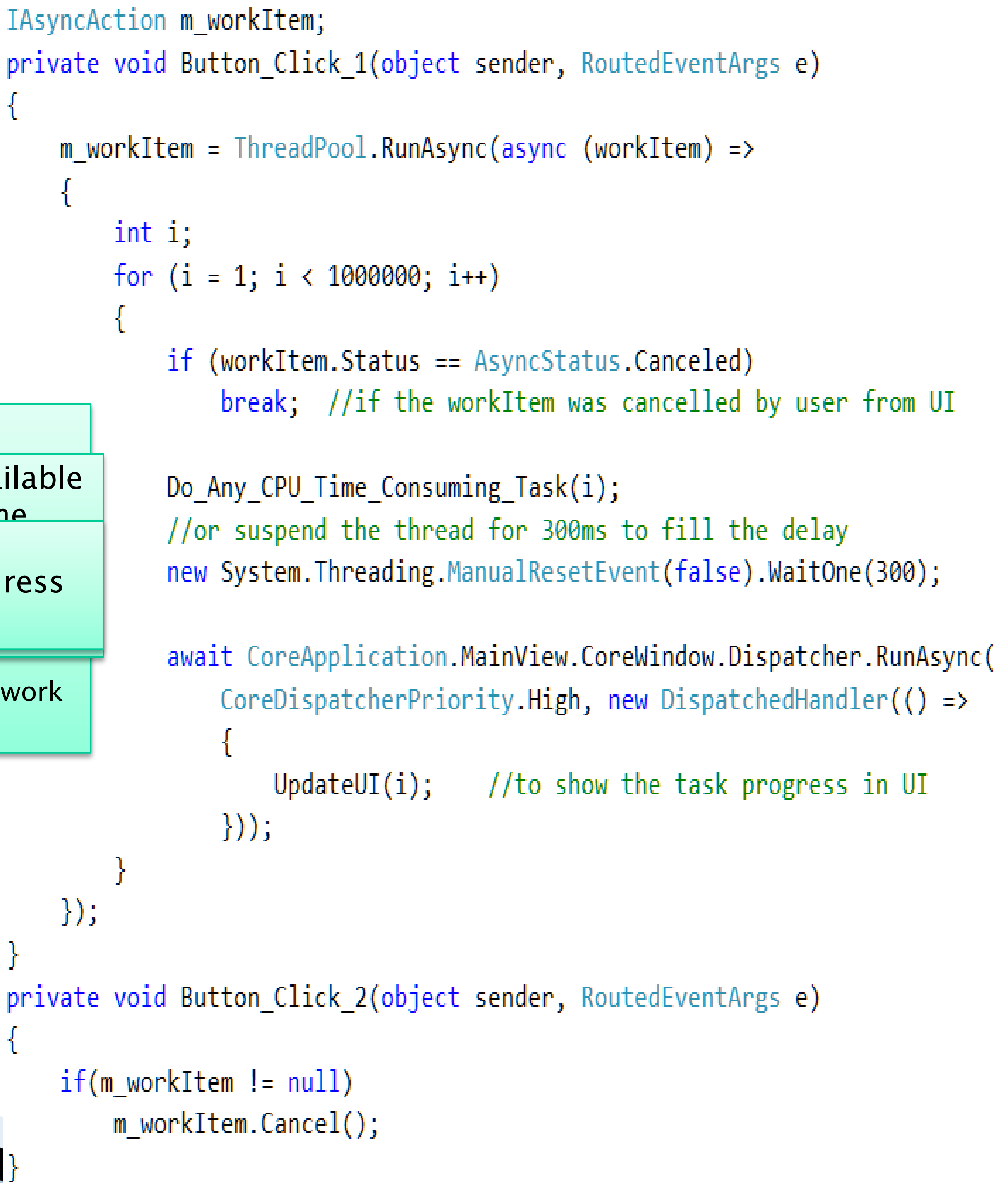
Submitting a work item to the thread pool

Create a work item by calling

Three versions of `RunAsync` are available so that you can optionally specify the

Use `CoreDispatcher.RunAsync` to access the UI thread and show progress from the work item.

- Store this object to cancel the work item.



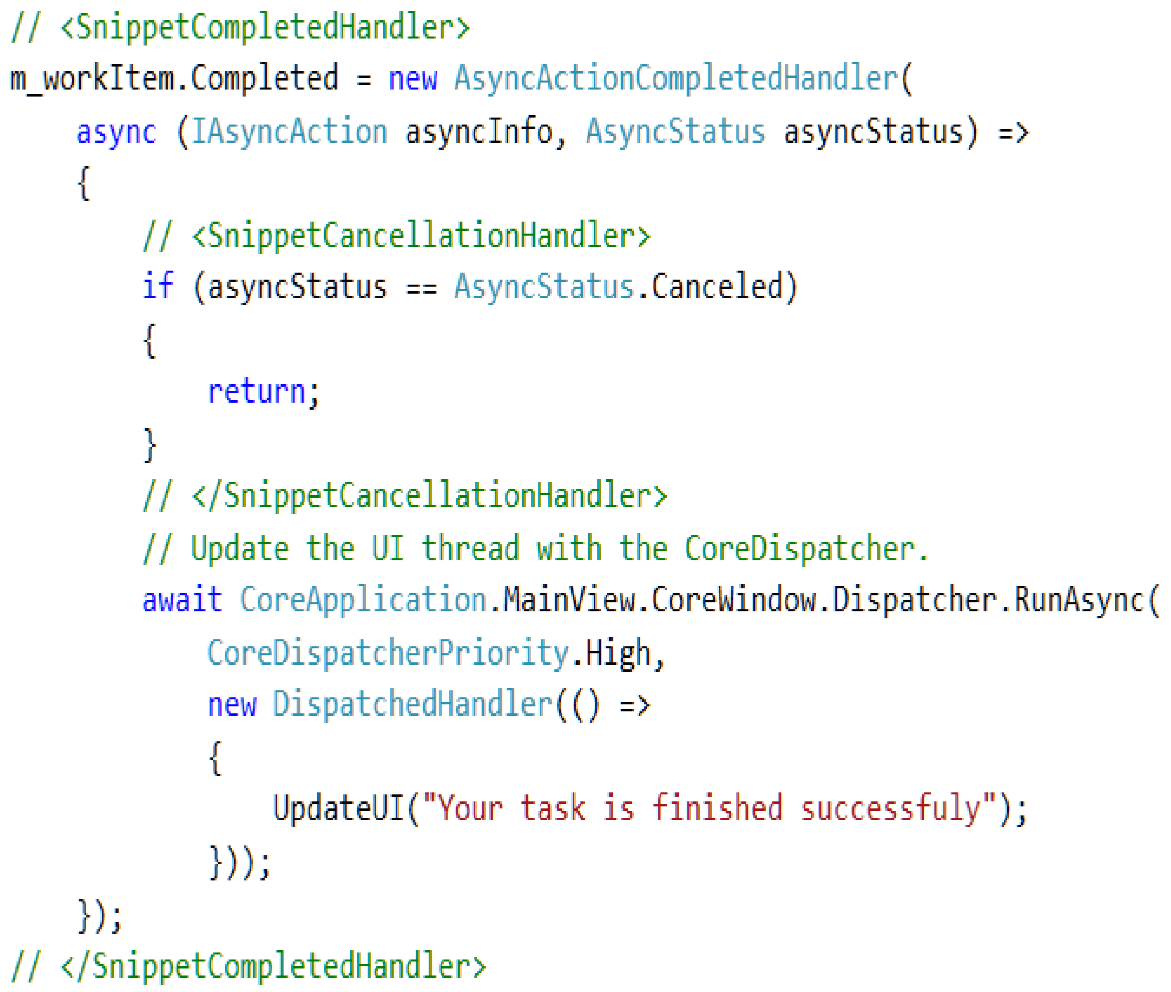
```
IAsyncAction m_workItem;
private void Button_Click_1(object sender, RoutedEventArgs e)
{
    m_workItem = ThreadPool.RunAsync(async (workItem) =>
    {
        int i;
        for (i = 1; i < 1000000; i++)
        {
            if (workItem.Status == AsyncStatus.Canceled)
                break;  //if the workItem was cancelled by user from UI

            Do_Any_CPU_Time_Consuming_Task(i);
            //or suspend the thread for 300ms to fill the delay
            new System.Threading.ManualResetEvent(false).WaitOne(300);

            await CoreApplication.MainView.CoreWindow.Dispatcher.RunAsync(
                CoreDispatcherPriority.High, new DispatchedHandler(() =>
                {
                    UpdateUI(i);    //to show the task progress in UI
                }));
        }
    });
}
private void Button_Click_2(object sender, RoutedEventArgs e)
{
    if(m_workItem != null)
        m_workItem.Cancel();
}
```

Handle work item completion

- This code finalized the Button_Clicked_1 method from previous example



```
// <SnippetCompletedHandler>
m_workItem.Completed = new AsyncActionCompletedHandler(
    async (IAsyncAction asyncInfo, AsyncStatus asyncStatus) =>
    {
        // <SnippetCancellationHandler>
        if (asyncStatus == AsyncStatus.Canceled)
        {
            return;
        }
        // </SnippetCancellationHandler>
        // Update the UI thread with the CoreDispatcher.
        await CoreApplication.MainView.CoreWindow.Dispatcher.RunAsync(
            CoreDispatcherPriority.High,
            new DispatchedHandler(() =>
            {
                UpdateUI("Your task is finished successfuly");
            }));
    });
// </SnippetCompletedHandler>
```

Use `CoreDispatcher.RunAsync` to access the UI thread and show the result.

Content was based on:

- Asynchronous Programming with Async and Await
  https://msdn.microsoft.com/en-us/library/hh191443(VS.110).aspx
- Using the thread pool in Windows Runtime apps
  https://msdn.microsoft.com/en-us/library/windows/apps/xaml/hh465290.aspx
- Submitting a work item to the thread pool
  https://msdn.microsoft.com/en-us/library/windows/apps/xaml/jj248677.aspx

The content is based on the Mic