ASYNCHRONOUS PROGRAMMING IN C# 5 WITHOUT USE OF MULTIPLE THREADS

Aleš Keprt

Department of Informatics, Moravian College Olomouc

ales.keprt@mvso.cz

ABSTRACT:

Asynchrony is a situation when multiple things occur at once, in parallel. In programming we encounter it quite often, typically for example when working with computer network or user interface, where program needs to wait for externally started events and want do its own work at once. These situations are usually treated by using multiple threads. Drawback of this approach is that programming with threads and debugging those programs is often quite hard and complicated, and threads can also lower computational power of a computer when there are too many of them used at once. Fifth version of the C# language brings a new approach to these situations, which is a revolutionary one in common praxis: Now we can create asynchronous programs even without use of multiple threads. We can create the same applications as before with all their goods, but with just a single thread. The paper introduces this new technology and shows which situations it is most suitable for.

KEYWORDS:

Asynchrony, C#, .NET, thread, continuation, await, apartment threading, exception

1 STATUS QUO: THREADS

In recent years it has been quite common to use multiple threads in programs. They provide at least two great benefits:

1. We can do more things at once. This is widely used on all kinds of servers, where they accept requests from their clients and each request can be executed by another thread. This way the server can utilize more than one physical CPU to speed up the whole system a lot. The idea of multiple CPUs in a single computer is now common on all kind of computers, so it is not limited to servers. Many kinds of programs now try to use more than one thread to speed up computation, including the most popular kinds of software like web browsers, audio and video players, office applications or computer games.
2. We can wait for something to happen and do other things until. This is usually related to external events, in praxis it happens most often when working with user interface and/or computer network. (Human users and other computers in network are those externals.) These situations are usually treated by using a dedicated thread to wait for all incoming events, and then it hand over the events to other threads to do the actual job.

From a computer user's perspective, the use of threads makes programs better. From a programmer's perspective, the use of threads brings complications to source code. Common

programming languages are primarily suited for single threaded programming, and we need to use a lot of inter-thread synchronizations due to shared memory. Some software, like latest web browsers, split their threads into multiple processes, so they actually don't need so much synchronization, but in turn it needs some level of inter-process synchronization which is also quite complex and nontrivial.

Other drawback of using many threads is that they need a lot of memory. Technically, each thread eats a few megabytes of RAM, as it needs its own program stack and task state segment with system variables. This isn't a big problem on smaller scale but limits the scaling of server applications a lot. For example making a multiuser server to host 5000 users at once is not possible with threads, because a computer can hardly create and support 5000 threads and (metaphorically) stay alive. Some operating systems like Windows NT family offer other solutions to these kinds of servers, which allow to do multiple disk and network access without threads, but these libraries are even more complicated to use than threads, so vast majority of real world applications stick with common thread based implementation.

One quite new field of problems where threads are used came into play when modern rich web applications started to spread. The key benefit from user's perspective is that these applications can do partial page updates, instead of full page reloads, which speeds up web applications and makes better users' experience. These applications can be implemented either with HTML and JavaScript, or with some other non-HTML framework like Adobe Flash or Microsoft Silverlight. These modern non-HTML web frameworks offer the possibility of creating better looking graphical user interface than classical HTML with CSS styles. While Adobe Flash is today de facto standard for web pages based on animations and video players, Microsoft's Silverlight starts to spread in the field of information systems and other enterprise applications where it can offer much better user experience than solutions based on ASP.NET, and still run in a standard web browser. All these web applications share one clear common quality: They often download additional data from server on background. Network can lag or go down sometimes, so these applications need to use threads in order to communicate over network and "stay alive" at the level of user interface.

2 IDEA OF SINGLE THREADED ASYNCHRONY

The previous section described why threads are used so often in nowadays software applications. Now let's introduce a new idea to it. If we look precisely on the multithreaded scenarios presented above, we can see that a significant portion of them is related to "waiting to something", not actually "computing something". Whether it's a user interface or a network communication, it actually doesn't compute things in parallel, it just waits for something to happen and do nothing until it happens. We say something is "parallel" when it actually does some computation in multiple threads, and "asynchronous" when it just waits for some activity to happen outside (and it happens parallel, but not here). Normally these two terms are melted together because it's how computer works – both parallel and asynchronous scenarios are often treated by threads. Now let's split it, and talk about asynchrony.

In general, asynchronous programming model can be used in any kind of situations. Speaking in programming terms, synchronous model is based on "call and wait for result, then use the result", while asynchronous is based on "call and return immediately, call back later with result". When asynchrony is well implemented and used, we don't waste time by waiting.

If we would be able to wait for something without multiple threads, we could make these kinds of applications much simpler, because we could omit all inter-thread synchronization (or at least most of it, if we kept some other threads of other reasons). Obviously, this cannot be made so easily. Namely, it needs a new programming paradigm, i.e. a new programming

language or a substantial change in an existing programming language. Fortunately, Microsoft is planning to make this substantial change in the forthcoming fifth version of their C# language. It means that we will be able to exploit benefits of the single threaded asynchrony in main stream programming language and mainstream platform, which is quite more interesting for many programmers than to have it implemented in a separate third party library.

This paper talks about CTP (community tech preview) version called "Microsoft Visual Studio Async" [Async, Mad10] which can be installed to Visual Studio 2010. It is a preliminary version of a library which is meant to be included in the next version of Visual Studio and C# language compiler.

3 EXAMPLE OF AWAIT-ASYNC PATTERN

Let's show the idea on a simple example. Let's imagine we want to somehow archive a list of documents which must be downloaded from web. We can do it in C# this way:

```
void ArchiveDocuments(List<Url> urls) {
  foreach(Url url in urls) Archive(Fetch(url));
}
```

And now let's do the same in parallel in C# 5:

```
async void ArchiveDocuments(List<Url> urls) {
  Task archive = null;
  foreach(Url url in urls) {
    var document = await FetchAsync(url);
    if(archive != null) await archive;
    archive = ArchiveAsync(document);
  }
}
```

This can seem to be confusing at the first sight, but it's not pseudocode – it's the real piece of code. There are new language keywords: **async** and **await**.

Keyword **async** is a marker for asynchronous methods. It says: "Hey, this method is asynchronous i.e. it returns immediately and then call back with result." Technically, these methods return Task or Task<T> instead of void or T, but they are declared as usual only with added async at the front.

Keyword **await** makes the program wait for async result to be computed and ready. The method returns immediately when this command is encountered and wanted result is not ready. Later when the awaited result is ready, the interrupted method is resumed.

In our example we call await twice: First time we wait for next document to be fetched, second time we wait for the previous document to be archived. This way we fetch documents one after another, and archive one after another. Fetching and archiving goes in parallel, but one document after another. The magic beyond the scenes adds one important feature to it: The program is single threaded, no synchronization is required by the programmer, and our ArchiveDocuments() method returns immediately. Indeed, it really returns back to the caller on the first await command. The flow gets back to ArchiveDocuments() later when the first document is fetched, its archiving is started, and the second document is started to be fetched. Then the flow switches back to main program. It switches back to ArchiveDocuments() later

when the second document is fetched. If the first document is not yet archived, the flow switches back to main program... etc.

The whole archiving operation is asynchronous, because it is marked async. We can either use it synchronously:

await ArchiveDocuments(urls);

or we can let it run truly asynchronously:

```
var archiving = ArchiveDocuments(urls);
...do anything...
await archiving; //here we wait for archiving to complete
```

So the use of await-async is very easy. The code behind added by C# compiler is still large, but we don't need to bother, it is hidden from us. Also, the main huge benefit of this approach is that the compiler makes it work without additional threads. It "somehow" makes all things happen in a switched context of a single thread. And yet again for us: We don't need to care about "how's".

4 CURRENT CONTINUATION

As said above, in praxis we don't need to care about "how's" when using await-async constructs. But we do care here. The whole concept is not based on threads, it's based on Continuation Passing Style (CPS) known from functional programming. (Also note that F# programming language (a new functional language in Visual Studio 2010) already does support a kind of single threaded asynchrony in its current version.)

Before continuing let's briefly recall the concept of CPS. CPS is an alternative to classical imperative programming languages' style of controlling flow. While imperative languages do one command after another, and allow to call a subroutine which can result some kind of a value, here we explicitly pass "what to do next" (wtdn) as an argument to a called function. Called function never returns, it uses this wtdn argument to know what to do after its own computation is finished. This programming style makes programs longer and less easily readable, but allows programmers to design their own flow control commands. For example if we imagine there is no commands like if, ?: operator, switch-case, while or do-while, we can program all these constructs from scratch if our programming language supports CPS. The important note here is that programming directly with CPS is not easy, but programming with those if, for or while is very easy and all the complexity is hidden behind in language's compiler. Similarly, await-async is easily usable, still very complicated behind and all that complicated stuff is hidden from us. (Another nice example where continuation is involved, is yield command in C# iterators.)

In the multithreaded world we would assume **await** command to mean: "Block the current thread until the asynchronous operation returns", and **async** command to mean: "Automatically schedule this method to run on a worker thread". [Lip10-A2] Interestingly, the opposite is truth.

Await never blocks the current thread. If the awaited thing is already computed, the flow advances normally. If the awaited thing isn't computed yet, the current continuation is packed into object and it is immediately returned to the calling method. In fact, await says: "This method returns immediately, either returning the computed result, or returning the continuation if the result is not ready yet." Or as defined by Lippert [Lip10-A2]: "If the task we are awaiting has not yet completed then sign up the rest of this method as the continuation

of that task, and then return to your caller immediately; the task will invoke the continuation when it completes."

Async declares the method as returning some kind of Task<T>, a standard .NET task (which, technically, can be run anytime later). Or as defined by Lippert [Lip10-A2]: "This method contains control flow that involves awaiting asynchronous operations and will therefore be rewritten by the compiler into continuation passing style to ensure that the asynchronous operations can resume this method at the right spot." The important point here is that async keyword doesn't say a word about how the real result of the method is computed. It can be run on a background worker thread, it can be done using operating system's threadless IO routines, etc. We don't care about how the async method computes the result; we just declare what to do when the method's result becomes ready. Also note that by using Task<> objects here we don't become multithreaded. Task class is a standard way of declaring a job to be done in .NET. It is used by Task Parallel Library for multithreaded computation, but here we can use it without threads. (We actually don't compute anything by using it. We really just declare a job to be computed. Somewhere, sometime, by someone…)

5 AWAITING MANY TASKS

We can also await more tasks than one at once. Let's get back to our prior example method ArchiveDocuments(), and now imagine we have a list of url lists which we want to archive and wait for it.

```
List<List<Url>> groupsOfUrls;
Task<long[]> allResults
 = Task.WhenAll(from urls in groupsOfUrls select ArchiveDocumentsAsync(urls));
long[] results = await allResults;
```

Task.WhenAll() method takes a list of tasks, asynchronously awaits each of them and return a task which represents this composition. Literally, it says: "This task gets finished when all those tasks are finished." Similarly, there is other method called **Task.WhenAny()** which awaits any single task from the list to complete.

6 SINGLE THREADED ASYNCHRONY UNCOVERED

In the previous sections we omitted details on how the C# compiler implements await and async keywords, because proper description would take too much space and can't fit into this short paper. Also, it is more important to clearly understand the end user's point of view. (By end user we mean a programmer who wants to use the await-async approach.)

So is it really possible to have all the work done in a single thread? Yes, it is. The situation is similar to the one well know from multithreaded programming on a single CPU computers. How can many threads run on a single CPU? They are switched. Indeed, once in a while operating system stores the continuation of the current thread and switches to the continuation of another one. Using the principle of continuation, operating system is able to schedule many threads on a single physical CPU. C# uses a similar approach to implement single threaded asynchrony.

There are a few different scheduling schemes in .NET for asynchronous tasks, based on the kind of application. Applications (or better said threads) with windows have got a message queue, so it is used as a primary means for scheduling asynchronous tasks. The message queue makes the application's user interface thread run in an apartment mode (i.e. single threaded apartment), which is great for our purposes. Await command is implemented so that it packs the current continuation to a task object and attaches this task via message queue to

the event of finishing the awaited task. Better said whenever the awaited task is finished, an event signals it, and the task with packed continuation is put to message queue so it can be resumed when the message loop gets the message. Other scheduling mechanisms are used in scenarios where message queue is not available (they are other, but technically similar), this applies e.g. to ASP.NET and other server-based scenarios [Lip10-A4].

7 EXCEPTIONS

Our final stand will be at exceptions. Normally we can use try-catch-finally blocks to deal with exceptional situations. In the case of single threaded asynchrony, the situation is a bit more complex because we often lost original program context and are unable to easily find the correct or intended catch and finally block. Let's show it on a simple example:

```
async void MyMethod() {
  try { await DoSomethingAsync(); }
  catch { ... }
  finally { ... }
}
```

If DoSomethingAsync() throws an exception before its first return, there will be no problem and our catch and finally blocks will normally run. If DoSomethingAsync() finishes its first run normally, it returns an asynchronous task. Then, later, when this task is scheduled, an exception may arise. In that case we cannot run our original catch and finally block, because MyMethod is long gone from call stack, and its try-catch-finally doesn't exist anymore. In that case the exception is caught and stored in the task object, and the task is marked as unsuccessful. When awaiting task (i.e. the interrupted MyMethod) is resumed using its continuation, it takes the exception from the (DoSomethingAsync) task object and serves the exception using intended catch and finally blocks. If our method (MyMethod) is unable to catch the exception (probably because the exception's type doesn't match) it does the same thing: Packs the exception into its task object. (Remember: Every method which uses await must be declared as async and return an asynchronous task object.)

8 CONCLUSION

In this paper we took a look at Microsoft's new asynchronous programming library which is to be shipped with the next version of C# language and .NET Framework version 5.0. We briefly showed where and why it is beneficial, and looked at its basic programming constructs from the user's perspective.

Note: All code examples presented in this paper are based on or taken directly from Eric Lippert's blog (it's on MSDN website, see the list of references).

REFERENCES

[Async] Microsoft Visual Studio Asynchronous Programming. *http://msdn.microsoft.com/en-us/vstudio/gg316360*

[Hej10] Anders Hejlsberg. *The Future of C# and Visual Basic.* PDC 2010, Microsoft Corporation. http://channel9.msdn.com/Events/PDC/PDC10/FT09

[Lip10-C1] Eric Lippert. *Continuation Passing Style Revisited.* MSDN Blogs, Microsoft Corporation, 2010. http://blogs.msdn.com/b/ericlippert/archive/2010/10/21/continuation-passing-style-revisited-part-one.aspx

[Lip10-A1] Eric Lippert. *Asynchrony in C# 5, Part One.* MSDN Blogs, Microsoft Corporation, 2010. http://blogs.msdn.com/b/ericlippert/archive/2010/10/28/asynchrony-in-c-5-part-one.aspx

[Lip10-A2] Eric Lippert. *Asynchronous Programming in C# 5.0 part two: Whence await?* MSDN Blogs, Microsoft Corporation, 2010. http://blogs.msdn.com/b/ericlippert/archive/2010/10/29/asynchronous-programming-in-c-5-0-part-two-whence-await.aspx

[Lip10-A4] Eric Lippert. *Asynchronous Programming in C# 5.0 part four: It's not magic.* MSDN Blogs, Microsoft Corporation, 2010. http://blogs.msdn.com/b/ericlippert/archive/2010/11/04/asynchrony-in-c-5-0-part-four-it-s-not-magic.aspx

[Tor10] Mads Torgersen. *Asynchronous Programming in C# and Visual Basic.* Microsoft Corporation, 2010. http://www.microsoft.com/downloads/en/details.aspx?FamilyID=d7ccfefa-123a-40e5-8ed5-8d2edd68acf4