

Re: forget cooperative multitasking in .NET ?

Source:

<http://www.tech-archive.net/Archive/DotNet/microsoft.public.dotnet.languages.csharp/2005-01/0339.html>

From: Frans Bouma [C# MVP] (*perseus.usenetNOSPAM_at_xs4all.nl*)

Date: 01/02/05

Date: Sun, 02 Jan 2005 16:51:26 +0100

Cor Ligthert wrote:

> Willy,

>

> *This makes me curious, I know the term multitasking as an IBM term, what was
> a substitute of multiprogramming a term used by Burroughs. Multiprogramming
> did mean that you did not have to keep track on the programs that where
> executed, they where managed by the OS side by side. Although you could (as
> far as I remember me) set with multiprogramming priorities about scheduling,
> executing and virtual memory, while there where a lot of possibilities of
> queuing and to synchronize the programs that used those what you can call
> using programs as threads.*

>

> *Multitasking was something (it was in the seventies) you had to set by hand
> (JCL) in those days. However there was an advantage from multitasking, with
> multitasking you had to tell what part of memory would be used and therefore
> the memory was protected. What was impossible with a Burroughs system
> (although I have never seen an error with that and was coding in BPL (a PLI
> substitute) and Cobol for that computer). IBM has forever had a good
> marketing team, so you know probably how important that memory protection
> became.*

>

> *For those days a Burroughs system was great, however probably much too
> advanced for those days.*

>

> *In a Burroughs system was everytime the program pushed down, when there was
> a new one loaded.*

>

> *Therefore I compare that Burroughs system often with W9x while as we know
> NTx comes from that IBM OS although it is not anymore done by hand (JCL).*

>

> *Can you maybe give me some light if my comparing is wrong in this, because I
> was a long time not interested in computer OS architecture however want to
> know where I am wrong in my ideas?*

I don't know what IBM nor JCL (?) has to do with it, but the thing is this:
on 16 bit windows (win3.xx), the scheduler worked with 16bit processes
only and basicly scheduled the processes/threads once the thread who had

the processor gave execution back to the scheduler. This is called 'co-operative' as multi-tasking was depending on the cooperation between applications. I.e.: if an application didn't give execution back to the scheduler, the system 'hanged' (no other threads got the CPU) unless a non-maskable interrupt was triggered.

on 32bit windows based on the win95 kernel (win9x, winME) the scheduler worked pre-emptively. This means that the whole system was divided in 32bit processes and the scheduler scheduled the threads of these processes using a priority stack and some other protocols (mostly round-robin). Each thread in the system was given a fixed amount of time, after that, the scheduler simply took the CPU away from the thread and the thread was physically halted as another thread was given the CPU. It depended on the priority/place in the thread queue which thread was given the CPU. This is totally different from the 16bit OS, as the scheduler decided which thread was given the CPU, not the thread currently running.

16bit applications were all ran in a single 32bit process. 16bit processes on 16bit windows shared the system memory so they all had access to each other's memory and as they were all scheduled as 1 process, their multitasking was still depending on co-operation, as the whole 16bit process space was given the CPU, inside it it was then decided which process got the CPU, i.e.: the last process which had the CPU.

On 32bit windows based on the NT kernel, this is not that different, except from the fact that 16bit processes are run inside a virtual machine.

Co-operative multi-tasking is stupid. It easily hangs the machine. (win9x/ME also had problems with 16bit processes which stepped on its own 16bit OS libraries (as they all shared the same memory), for example parts of the shell were 16bit code, so a bad 16bit program could hang the complete system, as it could overwrite parts of the OS without having the OS stopping it because of a GPF)

Co-operative multi-tasking is also just a trick to do things 'in parallel' without having to work with threads. The reason why some people still want it is that they want to do things in parallel in a synchronous way, as in pre-emptive multi-tasking it is undefined which thread gets the CPU and when, so it requires extra code to make asynchronous code work as planned and debugging that kind of code can be cumbersome.

We all do have synchronous multitasking code today: simply write a routine and call from that routine your other routines. That's co-operative multitasking. As I wrote earlier, a state-machine can help you with this. You can chop up a long running routine into several states and hop between them using the state machine, which then allows you to add states in between these 'substates' to get things done in parallel, but in a synchronous way. Using this I wrote 13 years ago a parallel computing system for a 120-node Sparc cluster (in C) which

microsoft.public.dotnet.languages.csharp: Re: forget cooperative multitasking in .NET ?

could schedule and dispatch jobs on all the nodes, synchronously and in parallel without having to have a lot of threads and syncing code in place.

Frans.

--

Get LLBLGen Pro, productive O/R mapping for .NET: <http://www.llblgen.com>
My .NET blog: <http://weblogs.asp.net/fbouma>
Microsoft MVP (C#)
