

Re: The revelation of St. f0dder the Divine

Source: <http://coding.derkeiler.com/Archive/Assembler/alt.lang.asm/2005-08/msg01720.html>

- *From:* "Beth" <BethStone21@xxxxxxxxxxxxxxxxxxxxxxxxxxxx>
 - *Date:* Tue, 23 Aug 2005 22:59:34 GMT
-

Frankie say:

> Beth wrote:

>> You've shown before that you don't entirely grasp how the "priority"
>> scheduling scheme in Windows works, hutch...and you're doing it
again,

>> it seems...

>

> It's my understanding that Hutch knows perfectly well that a polling
> loop is a very "inconsiderate" thing to do in a multi-tasking OS. But
> the blocking alternative crashes on certain hardware/version
> combinations. (I guess there's some question if this is true or not,
but
> that's why Hutch does it that way... I understand)

I'd require some pretty direct evidence of this to believe it...

Because, basically, a multi-tasking OS that doesn't do "blocking"
properly? That's not a "bug"...that's a "catastrophic error" of the
highest proportions...

Even for Microsoft, that would be an error of the highest
order...because while applications can happily carry on in their
sequential "I'm the only application running" fantasy that the OS create
for them...you CANNOT have such a "casual" attitude to concurrency on
"the other side of the fence"...

In short: The entire structure of the OS is FUNDAMENTALLY BASED on
"blocking"...if that doesn't work properly, then we're not just talking
about a "minor bug"...we're not even talking about a tendency to BSOD
every fifteen minutes...we're talking about the darn thing not even
being able to get off the ground in the first place...

Not merely a "bug"...but a "serious, fatal catastrophic error"...

You simply could not get Windows to operate at all, if "blocking" failed
to work reliably...and, now, as you know, I give Microsoft credit for
nigh-on nothing...but they MUST be getting this substantially correct
or, simply, Windows couldn't exist and operate the way it does...

Re: The revelation of St. f0dder the Divine

To explain: When an application "blocks", it's a "don't call us, we'll call you" arrangement...if the OS cannot be relied upon to properly "unblock" applications, then it would all "logjam" into a "frozen" state...within `_SECONDS_`...it would be a "crash" very much like a real-world "pile up" on the motorway / highway / autobahn...they'd smash into each other...

It's a `_FUNDAMENTAL_` operation we're talking about here...if, as hutch claims, "blocking" didn't work properly then Windows wouldn't simply be "prone to crashing", it wouldn't even start up properly...

Further, how can it only effect specific hardware? Most hardware operates by interrupt and has an IRQ...when the OS receives this IRQ, it logically – in software, so hardware has no effect on this happening – "unblocks" any tasks waiting on it...

Hence, the only "factor" that's hardware-based that could go wrong is attaching a piece of specific hardware that fails to correctly send an IRQ...except, such a piece of hardware, again, wouldn't be "prone to crashing" and this is a "small bug", as much as any hardware that fails to return an IRQ would be a useless pile of junk...which couldn't be seriously used for anything...

This "story" isn't sounding altogether believable that Windows somehow fails to "block / unblock" properly on these mysterious "certain hardware configurations"...the only hardware configuration where that sounds likely is the one where you try to install Windows on a Texas Instruments calculator...now, yeah, that surely wouldn't work properly...

Before hutch says something about "belief that the OS is always right is wrong-thinking", anyone who's seen me talk about Windows and its operations knows that I am far from believing it "all-perfect"...far, far, far, far from believing that...

But we're not talking about a "minor bug" here...if "blocking" didn't work reliably in a multi-tasking OS, then that would be a "serious, fatal and catastrophic error"...if you were deliberately trying to make it go wrong in the worst possible way imaginable, then this, in fact, would be the most obvious "target"...

I don't believe for a second that Windows is "all perfect"...heck, I don't believe Microsoft's coders even manage a "bare minimum" standard of coding, judging by some of the crap I've seen come from Microsoft...

But, you know: "observational evidence"...if "blocking" fails to function on a multi-tasking OS, then it would not even start up properly...it would instantly crash, for example, on the first "disk access" because, as you can see with your own eyes, when Windows accesses the hard disks, it does so without halting the entire system (the screen doesn't "freeze"...at least, not for hard drives...floppies

Re: The revelation of St. f0dder the Divine

on older Windows did do this, which also conveniently acts as an "example" of what to expect to see)...hence, the access is "asynchronous"...running on interrupts...

And the "blocking" stuff really is just a software "extension" of the hardware's own interrupt system...very much like that "structured exception handling" is a "software extension" of the CPU's own "exceptions" that it throws...the OS "catches" these things and responds to them appropriately...the OS just "extends" this to also allow for "software exceptions" and "software blocking / unblocking" (you know, like when Randy adds in his own "file not found" exception to the HLA standard library...that's not an actual hardware exception, of course...but Windows "genericises" its exception handling system that Randy can throw a software-generated "exception" into the same "mechanism" and it gets treated just the same as if the CPU itself "threw" it...

Regards "blocking", exact same deal: The hardware itself throws "IRQs" when operations complete...the "interrupt handler" which catches these IRQs, then "unblocks" those applications which "blocked" on the condition of waiting for the hardware to complete...say, starting an "asynchronous read" on the hard drive and then "blocking" on the data being loaded and coming back from the hard drive...the application sends the hardware a "command" to do that, then (how this works exactly is OS-dependent, as the API "differs" but a basic, simple example here) it "blocks" waiting for the "response"...when the hard drive finishes, it sends the IRQ, this is picked up by the "interrupt handler", which then automatically checks its "list" of applications "blocked"...then it "unblocks" them...the scheduler puts them on the "ready list"...they start executing next time the scheduler picks them...

The point with this system is that it's a "chain of events"...if any part of this "chain" did not function reliably, then the system **COULD NOT WORK AT ALL**...you know, if the hardware does not send back an IRQ, then the whole "chain of events" doesn't happen...and applications would **_NEVER_** "unblock"...

One by one, each application would "block" and then never come back out of that state...and – BANG! – a very strange version of "deadlock"...every process is "blocked"...none of them can become "unblocked" because the CPU is **_ONLY_** running "system idle process" (which never unblocks anything)...

The machine just sits there forever, waiting for a "condition" that cannot ever become true...ironically because the act of waiting itself means it'll never happen...

It's an unusual variation (and one, as I say, you could never see in practice from a functioning multi-tasking OS because this would make it seriously and fatally "non-functioning")...but, essentially, that's a "deadlock" condition...

Re: The revelation of St. f0dder the Divine

An OS like this where "blocking" fails to function properly, would nigh-on instantly "block" each process one by one and then "lock up" in a permanent and never-ending "deadlock" condition...it would become "frozen" forever...just as soon as each application does something that requires "asynchronous notification" (which, when almost all hardware works on an "interrupt / IRQ" system and the OS itself wires up its own internal stuff to also follow the same pattern: This would happen nigh-on instantly)...

As hutch reports that this depends on "hardware configuration" then this suggests, on other hardware, it functions perfectly...as we would expect...but this, therefore, suggests "software error" is RULED OUT from causing it...

Leaving the prospect of certain old hardware that fails to send IRQs properly?

Well, what do you think about that prospect? Sound likely? Hardware that just arbitrarily decides not to send IRQs...it just wouldn't work at all, would it? That's not a "bug", that's seriously and fatally broken hardware...

Even by Microsoft's policy of trying to support absolutely everything – even when it doesn't work properly – this is stretching things...

I would require some pretty convincing proof of the situation hutch speaks of...because, currently, it sounds like a "fairy tale"...as a system with such "bugs" in it wouldn't simply be "a bit temperamental" here and there...it just wouldn't boot it at all...

> The polling loop isn't getting 100% CPU time because there's a "sleep"
> in the loop. (again, some controversy whether it should be "sleep 0"
or
> "sleep 1", I guess...)

Exactly; And "sleep" is a BLOCKING API...the application "blocks" on the condition of a "timer" triggering it to "wake up" again...

This is the underlying point: The WHOLE SYSTEM is fundamentally based on "blocking"...the system couldn't function AT ALL, if this didn't work properly...even hutch's "solution" here is using "blocking" (whether he realises it or not), in order to not steal all of the CPU's time...

Hence, the contention that it "doesn't work properly" sounds rather insane and grossly unlikely...

Indeed, it's "not working properly" in a HIGHLY SELECTIVE way ("sleep" is okay, the scheduler's working fine, the hardware's working fine, etc.)...suspiciously, a way that's SELECTIVE of making hutch's example code not seem idiotic...what a "coincidence"...

Further, he's waiting on an "exit code", right?

That's entirely SOFTWARE based...a "thread" is a software entity...the scheduler's software (and, as everything is functioning okay in the system, by all reports, it can't be a fault there)... "termination" is just pulling the "thread" off the scheduler's lists (and a bit of "clean up")...an "exit code" is just an integer value that's "negotiated" between the terminating process and the system (to allow it to "communicate" error codes and such to other programs)...the whole thing is "software", with no reliance on any hardware (except that the "timer" interrupt works for the scheduler – and as that is functioning fine otherwise, then it can't be a problem there – and that the CPU correctly executes the code, which has to be a "given" or none of it would work at all: If the CPU was "broken", then you wouldn't even see the BIOS screen when you switch it on, let alone booting up the OS, let alone start running programs on the OS ;)...

Yet, we're told that it doesn't work on some older HARDWARE configurations?

Yeah, right...and I totally believe the "rain man" when he says that his "rain dance" brought the rains...because, obviously, there's a direct connection between some arbitrary snake oil salesman's feet and atmospheric conditions...sure, his dance "caused" the rain...whatever you say...

And, sure, "older hardware" triggers an ENTIRELY SOFTWARE BASED problem with SOFTWARE entities (that depends upon no hardware which isn't already proving itself to be functioning correctly in that the OS boots up and executes at all)...while, we're also told that on more modern hardware, the exact same SOFTWARE functions perfectly...so, there couldn't actually be a "bug" in the code...

Obviously, the software is just "objecting" to the choice of older hardware or something...making a "protest" to not work properly...yeah, right...this all adds up...totally logical...

> Being a fairly deviant antique myself, I'm running a k6-300 (no -2), and
> can (if I must) boot Win98 (no se, I don't think). If that combination
> would reliably demonstrate the problem, perhaps I could check it out
> (haven't tried it).

It would be interesting for you to conduct that "test"...

I wouldn't say completely categorically that this "bug" doesn't exist...because, well, when dealing with Microsoft, you learn never to say "never" about just how insanely idiotic they can be in screwing up a perfectly good piece of kit with appallingly bad software...knowing Microsoft, they might even have "put it in deliberately", just to try to

Re: The revelation of St. f0dder the Divine

"force an upgrade" or other nonsense like that...the more you learn of Microsoft, the less you find that a perposterous notion, the more it sounds "likely"...

But, doing the "Sherlock" on what information we've been given by hutch, this "story" really sounds like a "fairy tale"...

The system itself is fundamentally based on "blocking" throughout...from the OS side of things, you can't be "casual" about correctly implementing "concurrency"...if "unblocking" didn't work properly, then the entire system would go into a "deadlock" condition within a fraction of a second...and be permanently "frozen"...the "reset switch" (and using a different OS because this one would just consistently drive your machine to "deadlock" each and every time you booted it up) being your only escape...

The way this "concurrency" stuff and the scheduler works is actually a "delicately balanced" affair...it triggers a "chain of events" and if one of those "events" doesn't happen, then the system locks up...

Remembering, "blocking" is the scheme of "don't call us, we'll call you"...so, just like when you get a day off work in order to "wait for the man with the van to call around" (to deliver your new fridge ;)...you're sitting in the house all day long – unable to go anywhere, just in case that's the exact moment he decides to show up at your door – just waiting for him to turn up (that's the "condition" you're "blocked" on ;)...if he doesn't show up – at all – then kiss goodbye to the entire day, which has been completely wasted...and while, as human beings, we'd pick up the phone and complain and ask them why the man hasn't shown up...but, of course, the application that "blocks" literally gives up all CPU time...it's literally doing nothing and unable to do anything...it will sit there _FOREVER_, "sitting on a cornflake, waiting for the man to call"...that would be a "stupid bloody Tuesday", indeed...

This is why I'm having a hard time believing this "it doesn't work" stuff (especially while he calls "sleep" – a _BLOCKING API_ – in order to keep the CPU available...strange, ain't it? This "blocking" he is actually using works just fine ;)...)

If there was a serious "flaw" in the "blocking" system on Windows – of the kind he's suggesting – then this logically would lead to consistent "deadlock"...all the time...

If the "test" were to prove "positive" that there's a problem...then I'd immediately want to scan every line of _HUTCH'S PROGRAM_ to find the "flaw" that _HE_ (not the OS) is making...

As you know well, Frank, this has nothing to do with any belief that Windows is "flawless"...because I "Love" Windows so much...oh, far from it...but it's the logical conclusion all the information I've seen so

Re: The revelation of St. f0dder the Divine

Re: The revelation of St. f0dder the Divine

far leads me to...if Windows functions with other programs perfectly...if the same Windows version functions perfectly well on hardware that isn't "old"...these are all things that are effectively RULING OUT Windows as being the ultimate cause of the problem...

- > If some workaround for this bug could be found
- > *other* than a polling loop, that would be "better", IMHO, but perhaps
- > there isn't one...
- >
- > If the "blocking" alternative fails because of a bug, what would *you* do?

There are many ways to skin a cat...

Of course, it depends entirely on what the problem actually is – what's causing it – to know what exactly would be valid alternatives...

But, for example, one alternative that would probably work:

Create a system-wide "mutex"...the "child" grabs it upon creation (created in a "grabbed" state :)...the "parent" also tries to grab it using a "blocking condition" on the "mutex"...and – BANG! – the "parent" is blocked on this "mutex" which the "child" owns...when the child needs to terminate, it sets the "exit code" and then releases the "mutex"...and – BANG! – the "parent" then "unblocks" because it's now "owner" of the "mutex"...and it can "pick up" that "exit code"...

There, problem solved...

Note: This is GUARANTEED to work...the reason being that this is the way Microsoft "recommends" that a "may only run one instance" application (e.g. Media Player, as an example, refuses to run more than one instance simultaneously), ensures that only one instance runs under Win32 (with Win16, that second parameter of "WinMain" – "hinstPrevious" being the usual name in C coding – used to give you an "instance handle" to the "previous instance"...so, if "not NULL", then you knew that the application was already running somewhere...under Win32, Microsoft changed "memory model" and all processes are 100% SEPARATED...you're ALWAYS passed "NULL" for the previous instance's "handle"...the HINSTANCE value often isn't actually important (even in API that ask for it) because, truthfully, it's a "left over" from the Win16 model and is just retained for "source level compatibility"...you can test that out by sending "NULL" rather than the actual HINSTANCE to functions like "RegisterWindowClass" under Win32...and it still works just fine...as the whole "HINSTANCE" thing was a Win16 concept from when all applications shared the same address space...now, under Win32, all processes have their own address space and this parameter is mostly meaningless now...indeed, if you don't change the "defaults" then your linker will likely set EVERY application's "HINSTANCE" to 0x400000...well, it's hardly a "unique ID" when every application has the same value, right? It exists only for "compatibility" with the

Re: The revelation of St. f0dder the Divine

earlier Win16 model)...

Anyway, yes...because "hinstPrevious" doesn't work anymore since the change to Win32, Microsoft "recommend" instead that an application creates a system-wide "named mutex" the second it starts...and, simply, only the `_FIRST_` instance can "succeed" in creating the new "named mutex"...all the other instances would get a "mutex already exists" return, when trying to create it...and, simply, to make sure there's only one instance of an application running, every instance that detects it isn't the first instance, immediately terminates itself...

So, I know this method should work because it's the same method Microsoft are using (successfully) to ensure that, say, only one instance of "Media Player" is running...my "trick" is simply to use the "mutex" for the opposite reason: Detecting when there `_ISN'T_` any "instance" of the process running, as this means the "child" must have terminated...

That's, of course, the whole point of these "concurrency synchronisation primitives"...you're supposed to use them to "co-ordinate" things between separate concurrent processes...that's the point of their existence...

Dekker and Dijkstra went to a lot of effort to work out how to make these "synchronisation primitives" work in concurrent programming...indeed, for a while, some theoreticians even believed it an "impossible" task to come up with generalised solutions...

Unfortunately, because modern OSes go to such lengths to present applications with a "you own the entire machine" `_OPTICAL ILLUSION_`, many programmers don't realise that the second they start "launching multiple processes" in a multi-tasking operating system, they are entering "concurrent" territory...and that concurrency is `_NOT_` simple...it's `_NOT_` "just like the DOS batch file stuff but, like, more of it going on at the same time"...

Perhaps hutch will never get this point but "polling" and "blocking" are `_DIFFERENT_` things...if you "poll" and "drop priority" then you risk "starvation" by other processes...if you "poll" and "sleep" then, again, you risk "starvation"...you have to be a lot more careful and a lot smarter with concurrency than "DOS batch file programming"...really...

You can create "deadlock" (all processes waiting on conditions that the fact that all of them are waiting – not running – means it's a condition that'll never happen: "polling" or "blocking" is irrelevant to the basic "deadlock" problem, by the way)...or there's "livelock" (a condition never happens `_BECAUSE_` none of the processes will "block" as they should: A rarer problem but just as nasty – total system "lock up" – as "deadlock")...you can run programs 100 times and they "seem fine"...run it the 101th time and – BANG! – it all goes wrong because of the "coincidence" of where the scheduler decided to "pre-empt" the

processes...

Well, nevermind, Frank...you've got Linux running there...which has the full array of "IPC" stuff (more than Windows, which leaves out "shared memory" and such)...so, it's a good platform for learning about these "concurrent" things...and I can show you what the problems are and how to avoid them...if hutch wants to carry on "blind" that there's nothing to worry about from such a "cavalier" attitude to concurrent programming, then leave him...

Indeed, I don't know because I've not seen the code...BUT, I'm tempted to think that Windows is not responsible for the "bug" at all...hutch is responsible for it because he's accidentally setting up one of the various concurrent conditions that inherently cause "lock ups", "deadlocks", "crashes", "freezes", etc....again, many who don't appreciate that concurrency is a whole subject of great importance in itself, probably don't realise that you CAN create "error situations" in concurrency that have NOTHING to do with "wrong code" in and of itself...

But, indeed, actions speak louder than words...let your own machine show you this:

Concur.asm (MASM syntax)

----- 8< -----

.486

.model flat

NULL equ 0

FALSE equ 0

TRUE equ 1

CREATE_SUSPENDED equ 000000004h

INFINITE equ 0FFFFFFFFh

ExitProcess proto stdcall :DWORD

CreateThread proto stdcall :DWORD, :DWORD, :DWORD, \
:DWORD, :DWORD, :DWORD

ResumeThread proto stdcall :DWORD

CloseHandle proto stdcall :DWORD

ExitThread proto stdcall :DWORD

WaitForMultipleObjectsEx proto stdcall :DWORD, :DWORD, :DWORD, \
:DWORD, :DWORD

wsprintfA proto C :DWORD, :VARARG

MessageBoxA proto stdcall :DWORD, :DWORD, :DWORD, \
:DWORD

ThreadMain proto

```
.data

; Global "count" variable:
;
Count DWORD 0

; Thread handles
;
hFirst DWORD ?
hSecond DWORD ?
hThird DWORD ?
hFourth DWORD ?
hFifth DWORD ?
hSixth DWORD ?
hSeventh DWORD ?

; String stuff for message box
;
TitleText BYTE "Concurrent Count Program", 0
Format BYTE "Count: %u", 0
Text BYTE 256 dup (0)

.code

_start proc

; Start up seven threads
;
; (all running the "ThreadMain"
; procedure later on)...
;
invoke CreateThread, \
NULL, NULL, offset ThreadMain, \
NULL, NULL, NULL
mov hFirst, eax

invoke CreateThread, \
NULL, NULL, offset ThreadMain, \
NULL, NULL, NULL
mov hSecond, eax

invoke CreateThread, \
NULL, NULL, offset ThreadMain, \
NULL, NULL, NULL
mov hThird, eax

invoke CreateThread, \
NULL, NULL, offset ThreadMain, \
NULL, NULL, NULL
mov hFourth, eax
```

Re: The revelation of St. f0dder the Divine

```
invoke CreateThread, \  
NULL, NULL, offset ThreadMain, \  
NULL, NULL, NULL  
mov hFifth, eax
```

```
invoke CreateThread, \  
NULL, NULL, offset ThreadMain, \  
NULL, NULL, NULL  
mov hSixth, eax
```

```
invoke CreateThread, \  
NULL, NULL, offset ThreadMain, \  
NULL, NULL, NULL  
mov hSeventh, eax
```

```
; Wait for all threads to finish  
;  
invoke WaitForMultipleObjectsEx, \  
7, offset hFirst,  
TRUE, INFINITE,  
NULL
```

```
; Release handles to thread objects  
;  
invoke CloseHandle, hFirst  
invoke CloseHandle, hSecond  
invoke CloseHandle, hThird  
invoke CloseHandle, hFourth  
invoke CloseHandle, hFifth  
invoke CloseHandle, hSixth  
invoke CloseHandle, hSeventh
```

```
; Format output string  
;  
invoke wsprintfA, \  
offset Text, \  
offset Format, \  
Count
```

```
; Show result  
;  
invoke MessageBoxA, \  
NULL,  
offset Text,  
offset TitleText,  
NULL
```

```
; Exit program  
;  
invoke ExitProcess, 0
```

```
_start endp

ThreadMain proc

; Increment "count" 1 million times
;
mov ecx, 1000000
Again: mov eax, Count
inc eax
mov Count, eax

; the following serves two purposes
;
; 1. Wastes time to ensure that
; scheduler does "overlap" the
; threads (well, ASM code does
; run very quickly ;)...
;
; 2. It does a lot of PUSHing and
; POPing, just to assist with
; "trashing the cache" (as CPU
; optimisations with the cache
; can actual "counter" the
; effect I'm trying to get :)...
;
mov edx, 1000
PushLoop: push NULL
dec edx
jnz PushLoop
mov edx, 1000
PopLoop: pop eax
dec edx
jnz PopLoop

dec ecx
jnz Again

; Bye-bye thread!
;
invoke ExitThread, 0

ThreadMain endp

end _start
```

----->8-----

Right, the basic idea here is that the very simple program launches 7 threads...each of these threads increments "count" 1 million times...and then, when they've all finished, the program reports the value of "count" at the end of it all...

Time for another "Beth quiz":

"Count" starts at zero...7 threads are started...each thread increments "count" 1 million times...

So, what result are you going to get in the message box at the end?

Careful! It's a trick question...the actual answer is there's no way to know in advance...no possible way at all to predict it...

Yes, you heard right...there is no "trickery" in this program, as you can see for yourself...it starts 7 threads which each add 1 to the "Count" variable exactly 1 million times...these "input conditions" are _ALWAYS THE SAME_: Always 7 threads...each always incrementing the variable 1 million times...

Yet, when you run this program, you might expect an answer of 7 million (7 threads * 1 million increments)...but that may or may not be the answer you get...oh, it is possible to run it and get exactly 7 million...but it's also possible to run it and a little under 7 million...or a LOT under 7 million...

And the point is that the answer you'll get on a particular run is "indeterminate"...

Yes, you heard right...it's a program running "deterministic" instructions on a "deterministic" machine...and, yet, the result is quite unpredictable...non-deterministic..."chaotic", in the formal sense of the word...

In one sense, there is no "error" in the above code...and, yet, it produces an "indeterminate" result...as "chaotic" influence is introduced by the scheduler – hooked up to real world time – and the caches and millisecond delays in hard drive motors...

Welcome to chaos...welcome to unpredictable...welcome to non-determinism...

....welcome to the real world! ;)

Beth :)

.

-
- *Follow-Ups:*
 - ◆ *Re: The revelation of St. f0dder the Divine*
 - ◇ *From: Frank Kotler*

Re: The revelation of St. f0dder the Divine

- **References:**

- ◆ ***The revelation of St. f0dder the Divine***

- ◇ *From:* hutch--

- ◆ ***Re: The revelation of St. f0dder the Divine***

- ◇ *From:* Beth

- ◆ ***Re: The revelation of St. f0dder the Divine***

- ◇ *From:* Frank Kotler

- Prev by Date: ***Re: model of Z***

- Next by Date: ***Re: The revelation of St. f0dder the Divine***

- Previous by thread: ***Re: The revelation of St. f0dder the Divine***

- Next by thread: ***Re: The revelation of St. f0dder the Divine***

- Index(es):

- ◆ ***Date***

- ◆ ***Thread***