Why is creating a new process more expensive on Windows than Linux?

Asked 16 years, 3 months ago Modified 1 year, 9 months ago

Viewed 29k times

119

I've heard that creating a new process on a Windows box is more expensive than on Linux. Is this true? Can somebody explain the technical reasons for why it's more expensive and provide any historical reasons for the design decisions behind those reasons?

windows linux performance

Share

Improve this question

Follow

edited Oct 10, 2008 at 8:59

tzot

95.8k • 30 • 149 • 208

asked Sep 6, 2008 at 21:17

readonly

355k • 109 • 206 • 206

10 Answers Sorted by: Highest score (default)

mweerden: NT has been designed for multi-user from day one, so this is not really a reason. However, you are right about that process creation plays a less important role on

NT than on Unix as NT, in contrast to Unix, favors multithreading over multiprocessing.

Rob, it is true that fork is relatively cheap when COW is used, but as a matter of fact, fork is mostly followed by an exec. And an exec has to load all images as well. Discussing the performance of fork therefore is only part of the truth.

When discussing the speed of process creation, it is probably a good idea to distinguish between NT and Windows/Win32. As far as NT (i.e. the kernel itself) goes, I do not think process creation (NtCreateProcess) and thread creation (NtCreateThread) is significantly slower as on the average Unix. There might be a little bit more going on, but I do not see the primary reason for the performance difference here.

If you look at Win32, however, you'll notice that it adds quite a bit of overhead to process creation. For one, it requires the CSRSS to be notified about process creation, which involves LPC. It requires at least kernel32 to be loaded additionally, and it has to perform a number of additional bookkeeping work items to be done before the process is considered to be a full-fledged Win32 process. And let's not forget about all the additional overhead imposed by parsing manifests, checking if the image requires a compatibility shim, checking whether software restriction policies apply, yada yada.

That said, I see the overall slowdown in the sum of all those little things that have to be done in addition to the

raw creation of a process, VA space, and initial thread. But as said in the beginning -- due to the favoring of multithreading over multitasking, the only software that is seriously affected by this additional expense is poorly ported Unix software. Although this situation changes when software like Chrome and IE8 suddenly rediscover the benefits of multiprocessing and begin to frequently start up and teardown processes...

Share Improve this answer

Follow

edited Mar 13, 2023 at 22:05

Philippe

31k ● 6 ● 62 ● 87

answered Sep 7, 2008 at 8:17

Johannes Passing

2,805 ● 17 ● 13

11 Fork is not always followed by exec(), and people care about fork() alone. Apache 1.3 uses fork() (without exec) on Linux and threads on Windows, even if in many cases processes are forked before they are needed and kept in a pool.
– Blaisorblade Jan 12, 2009 at 4:02

5 Not forgetting of course, the 'vfork' command, which is designed for the 'just call exec' scenario you describe.
– Chris Huang-Leaver Aug 3, 2009 at 13:07

7 Another kind of software that is *seriously* affected by this is any kind of shell scripting that involves the coordination of multiple processes. Bash scripting inside Cygwin, for instance, suffers greatly from it. Consider a shell loop that spawns a lot of sed, awk, and grep in pipelines. Every command spawns a process and every pipe spawns a subshell and a new process in that subshell. Unix was designed with this very kind of usage in mind, which is why

fast process creation remains the norm there.
– Dan Moulding Aug 11, 2016 at 15:51

8 The claim that software is 'poorly ported' because it doesn't run well on a poorly designed operating system full of compatibility cruft that slows down process creation is ridiculous. – mrr Feb 5, 2017 at 21:08

15 @MilesRout the goal of porting is to modify software to run on a new target system, with that system's strengths and shortcomings in mind. Poorly performing ported software *is* poorly ported software, regardless of the roadblocks the operating system provides. – Dizzyspiral Feb 17, 2017 at 15:03

31

Adding to what JP said: most of the overhead belongs to Win32 startup for the process.

The Windows NT kernel actually does support COW fork. SFU (Microsoft's UNIX environment for Windows) uses them. However, Win32 does not support fork. SFU processes are not Win32 processes. SFU is orthogonal to Win32: they are both environment subsystems built on the same kernel.

In addition to the out-of-process LPC calls to `CSRSS`, in XP and later there is an out of process call to the application compatibility engine to find the program in the application compatibility database. This step causes enough overhead that Microsoft provides a group policy option to disable the compatibility engine on WS2003 for performance reasons.

The Win32 runtime libraries (kernel32.dll, etc.) also do a lot of registry reads and initialization on startup that don't apply to UNIX, SFU or native processes.

Native processes (with no environment subsystem) are very fast to create. SFU does a lot less than Win32 for process creation, so its processes are also fast to create.

UPDATE FOR 2019: add LXSS: Windows Subsystem for Linux

Replacing SFU for Windows 10 is the LXSS environment subsystem. It is 100% kernel mode and does not require any of that IPC that Win32 continues to have. Syscall for these processes is directed directly to lxss.sys/lxcore.sys, so the fork() or other process creating call only costs 1 system call for the creator, total. [A data area called the instance] keeps track of all LX processes, threads, and runtime state.

LXSS processes are based on native processes, not Win32 processes. All the Win32 specific stuff like the compatibility engine aren't engaged at all.

edited Jan 18, 2019 at 22:14

answered Sep 15, 2008 at 21:43

Chris Smith
5,446 31 30

29

Unix has a 'fork' system call which 'splits' the current process into two, and gives you a second process that is identical to the first (modulo the return from the fork call). Since the address space of the new process is already up and running this is should be cheaper than calling 'CreateProcess' in Windows and having it load the exe image, associated dlls, etc.

In the fork case the OS can use 'copy-on-write' semantics for the memory pages associated with both new processes to ensure that each one gets their own copy of the pages they subsequently modify.

Share Improve this answer

Follow

answered Sep 6, 2008 at 21:55

Rob Walker

47.4k ● 15 ● 100 ● 137

28 This argument only holds when you're really forking. If you're starting a new process, on Unix you still have to fork and exec. Both Windows and Unix have copy on write. Windows will certainly reuse a loaded EXE if you run a second copy of an app. I don't think your explanation is correct, sorry.
– Joel Spolsky Sep 17, 2008 at 2:43

1 More on exec() and fork()
vipinkrsahu.blogspot.com/search/label/system%20programming – webkul Nov 9, 2009 at 6:50

I added some performance data in my answer. stackoverflow.com/a/51396188/537980 You can see that it is faster. – ctrl-alt-delor Jul 18, 2018 at 7:49

17

In addition to the answer of Rob Walker: Nowadays you have things like the Native POSIX Thread Library - if you want. But for a long time the only way to "delegate" the work in the unix world was to use fork() (and it's still prefered in many, many circumstances). e.g. some kind of socket server

```
socket_accept()
fork()
if (child)
    handleRequest()
else
    goOnBeingParent()
```

Therefore the implementation of fork had to be fast and lots optimizations have been implemented over time. Microsoft endorsed CreateThread or even fibers instead of creating new processes and usage of interprocess communication. I think it's not "fair" to compare CreateProcess to fork since they are not interchangeable. It's probably more appropriate to compare fork/exec to CreateProcess.

Share Improve this answer

Follow

edited Nov 22, 2018 at 11:03

ctrl-alt-delor

7,725 ● 5 ● 44 ● 54

answered Sep 6, 2008 at 22:17

VolkerK

96.1k ● 20 ● 167 ● 231

2 About your last point: fork() is not exchangeable with CreateProcess(), but one can also say that Windows should implement fork() then, because that gives more flexibility. – Blaisorblade Jan 12, 2009 at 4:04

1 But fork+exec in Linux, is faster than CreateThread on MS-Windows. And Linux can do fork on its own to be even faster. However you compare it, MS is slower. – ctrl-alt-delor Nov 22, 2018 at 11:12

13

The key to this matter is the historical usage of both systems, I think. Windows (and DOS before that) have originally been single-user systems for *personal* computers. As such, these systems typically don't have to create a lot of processes all the time; (very) simply put, a process is only created when this one lonely user requests it (and we humans don't operate very fast, relatively speaking).

Unix-based systems have originally been multi-user systems and servers. Especially for the latter it is not uncommon to have processes (e.g. mail or http daemons) that split off processes to handle specific jobs (e.g. taking care of one incoming connection). An important factor in doing this is the cheap `fork` method (that, as mentioned by Rob Walker (47865), initially uses the same memory for the newly created process) which is very useful as the new process immediately has all the information it needs.

It is clear that at least historically the need for Unix-based systems to have fast process creation is far greater than

for Windows systems. I think this is still the case because Unix-based systems are still very process oriented, while Windows, due to its history, has probably been more thread oriented (threads being useful to make responsive applications).

Disclaimer: I'm by no means an expert on this matter, so forgive me if I got it wrong.

Share Improve this answer

Follow

edited May 23, 2017 at 12:02

Community Bot

1 ● 1

answered Sep 7, 2008 at 0:08

mweerden

14k ● 5 ● 34 ● 32

11

The short answer is "software layers and components".

The windows SW architecture has a couple of additional layers and components that don't exist on Unix or are simplified and handled inside the kernel on Unix.

On Unix, fork and exec are direct calls to the kernel.

On Windows, the kernel API is not used directly, there is win32 and certain other components on top of it, so process creation must go through extra layers and then the new process must start up or connect to those layers and components.

For quite some time researchers and corporations have attempted to break up Unix in a vaguely similar way, usually basing their experiments on the Mach kernel; a well-known example is OS X.. Every time they try, though, it gets so slow they end up at least partially merging the pieces back into the kernel either permanently or for production shipments.

Share Improve this answer

Follow

answered Dec 14, 2009 at 0:54

DigitalRoss

146k ● 25 ● 252 ● 331

Layers don't necessarily slow things down: I wrote a device driver, with lots of layers, in C. Clean code, literate programming, easy to read. It was faster (marginally), than a version written in highly optimised assembler, without layers. – ctrl-alt-delor Jul 18, 2018 at 7:42

The irony is that NT is a huge kernel (not a micro kernel) – ctrl-alt-delor Jan 10, 2019 at 10:49

9

Uh, there seems to be a lot of "it's better this way" sort of justification going on.

I think people could benefit from reading "Showstopper"; the book about the development of Windows NT.

The whole reason the services run as DLLs in one process on Windows NT was that they were too slow as separate processes.

If you got down and dirty you'd find that the library loading strategy is the problem.

On Unices (in general) the Shared libraries (DLLs) code segments are actually shared.

Windows NT loads a copy of the DLL per process, becauase it manipulates the library code segment (and executable code segment) after loading. (Tells it where is your data?)

This results in code segments in libraries that are not reusable.

So, the NT process create is actually pretty expensive. And on the down side, it makes DLLs no appreciable saving in memory, but a chance for inter-app dependency problems.

Sometimes it pays in engineering to step back and say, "now, if we were going to design this to really suck, what would it look like?"

I worked with an embedded system that was quite temperamental once upon a time, and one day looked at it and realized it was a cavity magnetron, with the electronics in the microwave cavity. We made it much more stable (and less like a microwave) after that.

edited Apr 9, 2021 at 16:50

ctrl-alt-delor

7,725 ● 5 ● 44 ● 54

answered Sep 8, 2008 at 23:58

Tim Williscroft

3,735 ● 25 ● 37

3 Code segments are reusable as long as the DLL loads at its preferred base address. Traditionally you should ensure that you set non-conflicting base addresses for all DLLs that would load into your processes, but that doesn't work with ASLR. – Mike Dimmick Sep 17, 2008 at 12:47

There is some tool to rebase all DLLs, isn't there? Not sure what it does with ASLR. – Zan Lynx Oct 21, 2008 at 0:11

3 Sharing of code sections works on ASLR-enabled systems as well. – Johannes Passing Dec 4, 2009 at 19:31

@MikeDimmick so everybody, making a DLL has to cooperate, to ensure that there are no conflicts, or do you patch them all at a system level, before loading?
– ctrl-alt-delor Jan 10, 2019 at 10:46

5

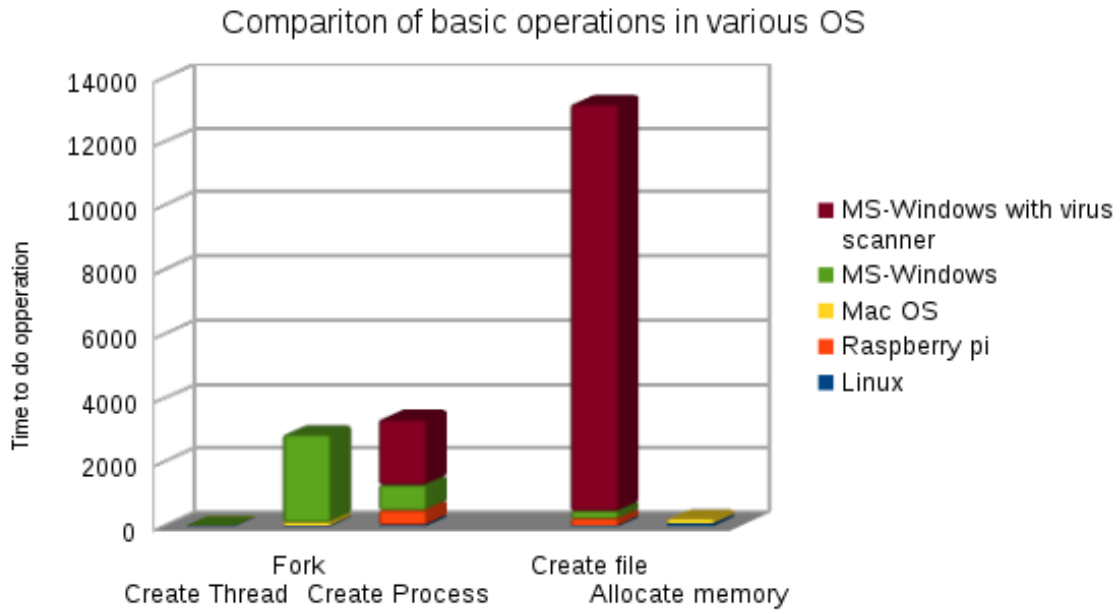
As there seems to be some justification of MS-Windows in some of the answers e.g.

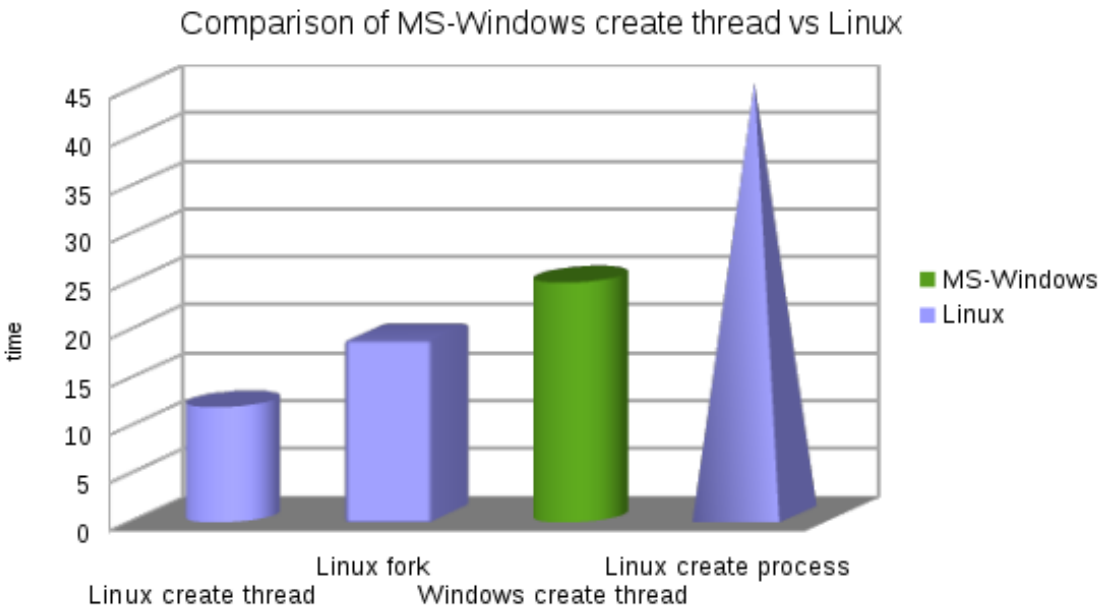
- “NT kernel and Win32, are not the same thing. If you program to NT kernel then it is not so bad” — True, but unless you are writing a Posix subsystem, then who cares. You will be writing to win32.
- “It is not fair to compare fork, with ProcessCreate, as they do different things, and Windows does not have fork“ — True, So I will compare like with like. However I will also compare fork, because it has many many use cases, such as process isolation

(e.g. each tab of a web browser runs in a different process).

Now let us look at the facts, what is the difference in performance?

Data summerised from http://www.bitsnbites.eu/benchmarking-os-primitives/.
Because bias is inevitable, when summarising, I did it in favour of MS-Windows
Hardware for most tests i7 8 core 3.2GHz. Except Raspberry-Pi running Gnu/Linux





Notes: On linux, `fork` is faster that MS-Window's preferred method `CreateThread`.

Numbers for process creation type operations (because it is hard to see the value for Linux in the chart).

In order of speed, fastest to slowest (numbers are time, small is better).

- Linux CreateThread 12
- Mac CreateThread 15
- Linux Fork 19
- Windows CreateThread 25
- Linux CreateProcess (fork+exec) 45

- Mac Fork 105
- Mac CreateProcess (fork+exec) 453
- Raspberry-Pi CreateProcess (fork+exec) 501
- Windows CreateProcess 787
- Windows CreateProcess With virus scanner 2850
- Windows Fork (simulate with CreateProcess + fixup) grater than 2850

Numbers for other measurements

- Creating a file.
  - Linux 13
  - Mac 113
  - Windows 225
  - Raspberry-Pi (with slow SD card) 241
  - Windows with defender and virus scanner etc 12950
- Allocating memory
  - Linux 79
  - Windows 93
  - Mac 152

 edited Jun 20, 2020 at 9:12

Follow

Community Bot
1 • 1

answered Jul 18, 2018 at 7:36

ctrl-alt-delor
7,725 • 5 • 44 • 54

1

All that plus there's the fact that on the Win machine most probably an antivirus software will kick in during the CreateProcess... That's usually the biggest slowdown.

Share Improve this answer

Follow

answered Sep 7, 2008 at 18:25

gabr
26.8k • 10 • 79 • 142

1 Yes it is the biggest, but not the only significant slowdown. – ctrl-alt-delor Jul 1, 2019 at 8:12

1

It's also worth noting that the security model in Windows is vastly more complicated than in unix-based OSs, which adds a lot of overhead during process creation. Yet another reason why multithreading is preferred to multiprocessing in Windows.

Share Improve this answer

Follow

edited Jul 18, 2018 at 7:39

ctrl-alt-delor
7,725 • 5 • 44 • 54

answered Jan 23, 2011 at 18:43

hacksoncode
19 • 1

1 I would expect a more complicated security model to be more secure; but facts shows otherwise. – Lie Ryan May 25, 2011 at 15:05

4 SELinux is also a very complex security model, and it doesn't impose a significant overhead on `fork()` – Spudd86 Aug 2, 2011 at 18:48

7 @LieRyan, In software design (in my experience), more complicated very rarely means more secure.
– Woodrow Douglass Dec 30, 2011 at 15:00