Hari_Pulapaka Microsoft

Oct 17 2018 12:39 PM ...

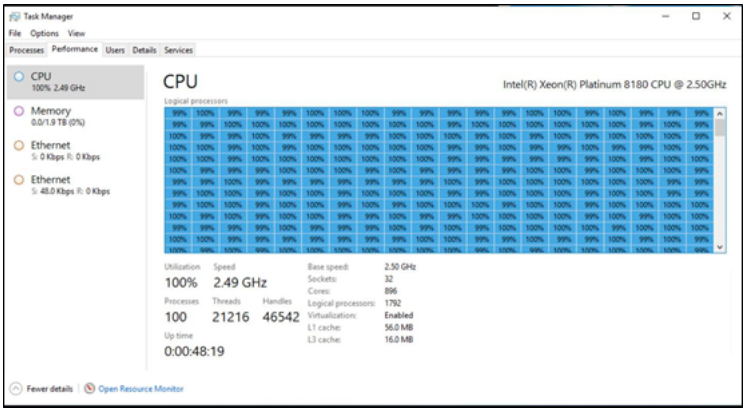
One Windows Kernel

Windows is one of the most versatile and flexible operating systems out there, running on a variety of machine architectures and available in multiple SKUs. It currently supports x86, x64, ARM and ARM64 architectures. Windows used to support Itanium, PowerPC, DEC Alpha, and MIPS (wiki entry). In addition, Windows supports a variety of SKUs that run in a multitude of environments; from data centers, laptops, Xbox, phones to embedded IOT devices such as ATM machines.

The most amazing aspect of all this is that the core of Windows, its kernel, remains virtually unchanged on all these architectures and SKUs. The Windows kernel scales dynamically depending on the architecture and the processor that it's run on to exploit the full power of the hardware. There is of course some architecture specific code in the Windows kernel, however this is kept to a minimum to allow Windows to run on a variety of architectures.

In this blog post, I will talk about the evolution of the core pieces of the Windows kernel that allows it to transparently scale across a low power NVidia Tegra chip on the Surface RT from 2012, to the giant behemoths that power Azure data centers today.

This is a picture of Windows taskmgr running on a pre-release Windows DataCenter class machine with 896 cores supporting 1792 logical processors and 2TB of RAM!

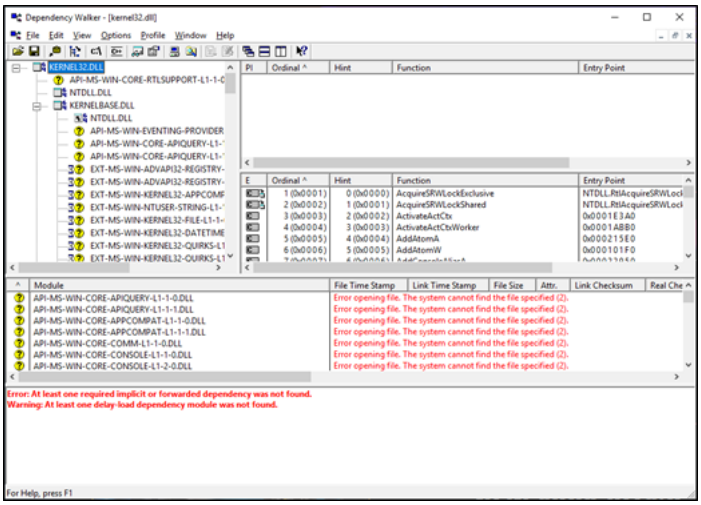


Task Manager showing 1792 logical processors

Evolution of one kernel

Before we talk about the details of the Windows kernel, I am going to take a small detour to talk about something called Windows refactoring. Windows refactoring plays a key part in increasing the reuse of Windows components across different SKUs, and platforms (e.g. client, server and phone). The basic idea of Windows refactoring is to allow the same DLL to be reused in different SKUs but support minor modifications tailored to the SKU without renaming the DLL and breaking apps.

The base technology used for Windows refactoring are a lightly documented technology (entirely by design) called API sets. API sets are a mechanism that allows Windows to decouple the DLL from where its implementation is located. For example, API sets allow win32 apps to continue to use kernel32.dll but, the implementation of all the APIs are in a different DLL. These implementation DLLs can also be different depending on your SKU. You can see API sets in action if you launch dependency walker on a traditional Windows DLL; e.g. kernel32.dll.

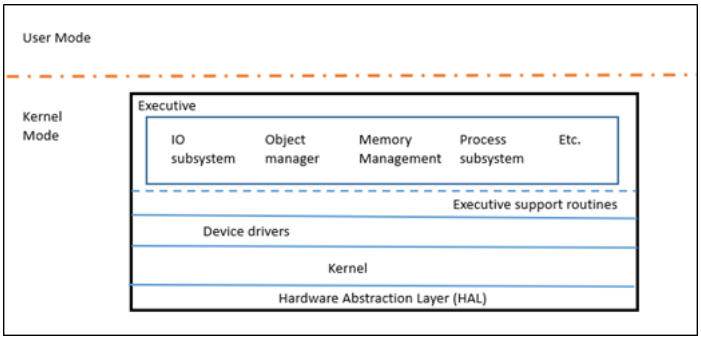


Dependency walker

With that detour into how Windows is built to maximize code reuse and sharing, let's go into the technical depths of the kernel starting with the scheduler which is key to the scaling of Windows.

Kernel Components

Windows NT is like a microkernel in the sense that it has a core Kernel (KE) that does very little and uses the Executive layer (Ex) to perform all the higher-level policy. Note that EX is still kernel mode, so it's not a true microkernel. The kernel is responsible for thread dispatching, multiprocessor synchronization, hardware exception handling, and the implementation of low-level machine dependent functions. The EX layer contains various subsystems which provide the bulk of the functionality traditionally thought of as kernel such as IO, Object Manager, Memory Manager, Process Subsystem, etc.



To get a better idea of the size of the components, here is a rough breakdown on the number of lines of code in a few key directories in the Windows kernel source tree (counting comments). There is a lot more to the Kernel not shown in this table.

Kernel subsystems	Lines of code
Memory Manager	501, 000
Registry	211,000
Power	238,000
Executive	157,000
Security	135,000
Kernel	339,000
Process sub-system	116,000

For more information on the architecture of Windows, the "Windows Internals" series of books are a good reference.

Scheduler

With that background, let's talk a little bit about the scheduler, its evolution and how Windows kernel can scale across so many different architectures with so many processors.

A thread is the basic unit that runs program code and it is this unit that is scheduled by the Window scheduler. The Windows scheduler uses the thread priority to decide which thread to run and in theory the highest priority thread on the system always gets to run even if that entails preempting a lower priority thread.

As a thread runs and experiences quantum end (minimum amount of time a thread gets to run), its dynamic priority decays, so that a high priority CPU bound thread doesn't run forever starving everyone else. When another waiting thread is awakened to run, it is given a priority boost based on the importance of the event that caused the wait to be satisfied (e.g. a large boost is for a foreground UI thread vs. a smaller one for completing disk I/O). A thread therefore runs at a high priority as long as it's interactive. When it becomes CPU (compute) bound, its priority decays, and it is considered only after other, higher priority threads get their time on the CPU. In addition, the kernel arbitrarily boosts the priority of ready threads that haven't received any processor time for a given period of time to prevent starvation and correct priority inversions.

The Windows scheduler initially had a single ready queue from where it picked up the next highest priority thread to run on the processor. However, as Windows started supporting more and more processors the single ready queue turned out to be a bottleneck and around **Windows Server 2003,** the scheduler changed to one ready queue per processor. As Windows moved to multiple per processor queues, it avoided having a single global lock protecting all the queues and allowed the scheduler to make locally optimum decisions. This means that any point the single highest priority thread in the system runs but that doesn't necessarily mean that the top N (N is number of cores) priority threads on the system are running. This proved to be good enough until Windows started moving to low power CPUs, e.g. in laptops and tablets. On these systems, not running a high priority thread (such as the foreground UI thread) caused the system to have noticeable glitches in UI. And so, in **Windows 8.1**, the scheduler changed to a hybrid model with per processor ready queues for affinitized (tied to a processor) work and shared ready queues between processors. This did not cause a noticeable impact on performance because of other architectural changes in the scheduler such as the dispatcher database lock refactoring which we will talk about later.

Windows 7 introduced something called the Dynamic Fair Share Scheduler; this feature was introduced primarily for terminal servers. The problem that this feature tried to solve was that one terminal server session which had a CPU intensive workload could impact the threads in other terminal server sessions. Since the scheduler didn't consider sessions and simply used the priority as the key to schedule threads, users in different sessions could impact the user experience of others by starving their threads. It also unfairly advantages the sessions (users) who has a lot of threads because the sessions with more threads get more opportunity to be scheduled and received CPU time. This feature tried to add policy to the scheduler such that each session was treated fairly and roughly the same amount of CPU was available to each session. Similar functionality is available in Linux as well, with its Completely Fair Scheduler. In **Windows 8**, this concept was generalized as a scheduler group and added to the Windows Scheduler with each session in an independent scheduler group. In addition to the thread priority, the scheduler uses the scheduler groups as a second level index to decide which thread should run next. In a terminal server, all the scheduler groups are weighted equally and hence all sessions (scheduler groups) receive the same amount of CPU regardless of the number or priorities of the threads in the scheduler groups. In addition to its utility in a terminal server session, scheduler groups are also used to have fine grained control on a process at runtime. In Windows 8, **Job objects** were enhanced to support CPU rate control. Using the CPU rate control APIs, one can decide how much CPU a process can use, whether it should be a hard cap or a soft cap and receive notifications when a process meets those CPU limits. This is like the resource controls features available in cgroups on Linux.

Starting with **Windows 7**, Windows Server started supporting greater than 64 logical processors in a single machine. To add support for so many processors, Windows internally introduced a new entity called "processor group". A group is a static set of up to 64 logical processors that is treated as a single scheduling entity. The kernel determines at boot time which processor belongs to which group and for machines with less than 64 cores, with the overhead of the group structure indirection is mostly not noticeable. While a single process can span groups (such as a SQL server instance), and individual thread could only execute within a single scheduling group at a time.

However, on machines with greater than 64 cores, Windows started showing some bottlenecks that prevented high performance applications such as SQL server from scaling their performance linearly with the number of processor cores. Thus, even if you added more cores and memory, the benchmarks wouldn't show much increase in performance. And one of the main problems that caused this lack of performance was the contention around the Dispatcher database lock. The dispatcher database lock protected access to those objects that needed to be dispatched; i.e. scheduled. Examples of objects that were protected by this lock included threads, timers, I/O completion ports, and other waitable kernel objects (events, semaphores, mutants, etc.). Thus, in Windows

7 due to the impetus provided by the greater than 64 processor support, work was done to eliminate the dispatcher database lock and replace it with fine grained locks such as per object locks. This allowed benchmarks such as the SQL TPC-C to show a **290%** improvement when compared to Windows 7 with a dispatcher database lock on certain machine configurations. This was one of the biggest performance boosts seen in Windows history, due to a single feature.

Windows 10 brought us another innovation in the scheduler space with CPU Sets. CPU Sets allow a process to partition the system such that its process can take over a group of processors and not allow any other process or system to run their threads on those processors. Windows Kernel even steers Interrupts from devices away from the processors that are part of your CPU set. This ensures that even devices cannot target their code on the processors which have been partitioned off by CPU sets for your app or process. Think of this as a low-tech Virtual Machine. As you can imagine this is a powerful capability and hence there are a lot of safeguards built-in to prevent an app developer from making the wrong choice within the API. CPU sets functionality are used by the customer when they use Game Mode to run their games.

Finally, this brings us to **ARM64** support with Windows 10 on ARM. The ARM architecture supports a big.LITTLE architecture, big.LITTLE is a heterogenous architecture where the "big" core runs fast, consuming more power and the "LITTLE" core runs slow consuming less power. The idea here is that you run unimportant tasks on the little core saving battery. To support big.LITTLE architecture and provide great battery life on Windows 10 on ARM, the Windows scheduler added support for **heterogenous scheduling** which took into account the app intent for scheduling on big.LITTLE architectures.

By app intent, I mean Windows tries to provide a quality of service for apps by tracking threads which are running in the foreground (or starved of CPU) and ensuring those threads always run on the big core. Whereas the background tasks, services, and other ancillary threads in the system run on the little cores. (As an aside, you can also programmatically mark your thread as unimportant which will make it run on the LITTLE core.)

Work on Behalf: In Windows, a lot of work for the foreground is done by other services running in the background. E.g. In Outlook, when you search for a mail, the search is conducted by a background service (Indexer). If we simply, run all the services on the little core, then the experience and performance of the foreground app will be affected. To ensure, that these scenarios are not slow on big.LITTLE architectures, Windows actually tracks when an app calls into another process to do work on its behalf. When this happens, we donate the foreground priority to the service thread and force run the thread in the service on the big core.

That concludes our first (huge?) One Windows Kernel post, giving you an overview of the Windows Kernel Scheduler. We will have more similarly technical posts about the internals of the Windows Kernel.

Hari Pulapaka
(Windows Kernel Team)

77 Likes

Share

Comments

nuno silva MVP Oct 17 2018 02:53 PM

1 Like

Raymond Tan Occasional Visitor Oct 18 2018 01:16 AM

0 Likes

Дмитрий Парасочка Visitor Oct 18 2018 07:51 AM

4 Likes

Vasile Paraschiv Occasional Visitor Oct 18 2018 08:13 AM

1 Like

Hari_Pulapaka Microsoft Oct 18 2018 09:16 AM

3 Likes

Torsten Schuster Senior Member Oct 18 2018 09:56 AM

0 Likes

Ofek Shilon Occasional Visitor Oct 18 2018 11:03 AM

0 Likes

Jason Galvin Occasional Visitor Oct 18 2018 11:10 AM

0 Likes

Pedro Henrique Occasional Visitor Oct 18 2018 04:03 PM

0 Likes

Brent Morris Super Contributor

Oct 19 2018 03:36 PM
- edited
Oct 19 2018 03:38 PM

- edited

0 Likes

Marie Adam Occasional Visitor

Oct 19 2018 07:53 PM

1 Like

Dylan Green Visitor

Oct 20 2018 01:16 PM

1 Like

Jan Sichula Occasional Visitor

Oct 22 2018 02:53 AM

1 Like

Vladimir Davidovich Regular Visitor

Oct 24 2018 04:14 AM

0 Likes

quanteneqx Senior Member

Oct 24 2018 08:11 PM
- edited
Oct 24 2018 08:13 PM

- edited

0 Likes

Jeff Chase Senior Member

Oct 27 2018 08:01 AM

0 Likes

Mattias Borg Super Contributor

Oct 30 2018 03:02 AM

0 Likes

Zhao Wei Liew Visitor

Oct 30 2018 05:44 AM

0 Likes

Slava Imameev Occasional Visitor

Oct 30 2018 08:37 AM
- edited
Oct 30 2018 09:09 AM

- edited

0 Likes

Stephen Brooks Occasional Visitor

Nov 06 2018 12:43 PM

0 Likes

Slava Imameev Occasional Visitor

Nov 06 2018 10:10 PM

1 Like

Yajun Yang Occasional Visitor

Nov 07 2018 07:20 PM

0 Likes

John Phillips Occasional Visitor

Nov 08 2018 04:27 PM

0 Likes

ing conti Occasional Visitor

Nov 08 2018 10:16 PM
- edited
Nov 08 2018 10:23 PM

- edited

0 Likes

Muntashir Al-Islam Frequent Visitor

Nov 10 2018 04:09 AM

0 Likes

QingXu Li Frequent Visitor

Nov 15 2018 01:19 AM

0 Likes

mendelmonteiro Regular Visitor

Nov 20 2018 03:15 PM

1 Like

Hari_Pulapaka Microsoft

Dec 07 2018 12:58 PM

0 Likes

mendelmonteiro Regular Visitor Dec 08 2018 05:44 AM

1 Like

winkeu Frequent Visitor Dec 19 2018 07:01 AM

0 Likes

Ib Tornøe Senior Member Dec 19 2018 11:05 PM

1 Like

Christian Kurs Senior Member Dec 21 2018 05:49 AM

0 Likes

vkoylazov Occasional Visitor Feb 07 2019 11:46 PM

0 Likes

MGro1 Occasional Visitor Apr 16 2019 09:16 AM

1 Like

kuudra Occasional Visitor May 08 2019 12:26 PM

0 Likes

KernelPanic Occasional Visitor

May 23 2019 10:54 AM
- edited
May 23 2019 10:55 AM

- edited

1 Like

GreggGW Occasional Visitor

Aug 26 2019 06:12 AM

0 Likes

HotCakeX Honored Contributor

Oct 09 2019 03:40 AM

0 Likes

Tarran Walker Occasional Contributor

Oct 24 2019 03:34 AM

0 Likes

NT_Addict Senior Member

Jun 05 2020 03:45 AM

0 Likes

leonidp Visitor

Jun 22 2020 05:28 AM

0 Likes

GreggGW Occasional Visitor Oct 25 2020 08:20 AM

0 Likes

Kyle_TM New Contributor Jan 01 2023 08:49 PM

0 Likes