Processes, commit, RAM, threads, and how high can you go?

devblogs.microsoft.com/oldnewthing/20110106-00

January 6, 2011

Raymond Chen

Back in 2008, Igor Levicki made a boatload of incorrect assumptions in an attempt to calculate the highest a process ID can go on Windows NT. Let's look at them one at a time.

> So if you can't create more than 2,028 threads in one process (because of 2GB per process limit) and each process needs at least one thread, that means you are capped by the amount of physical RAM available for stack.

One assumption is that *each process needs at least one thread*. Really? What about a process that has exited? (Some people call these *zombie* processes.) There are no threads remaining in this process, but the process object hangs around until all handles are closed.

Next, the claim is that *you are capped by the amount of physical RAM available for stack*. This assumes that stacks are non-pageable, which is an awfully strange assumption. User-mode stacks are most certainly pageable. In fact, *everything* in user-mode is pageable unless you take special steps to make it not pageable.

> Given that the smallest stack allocation is 4KB and assuming 32-bit address space:
>
> 4,294,967,296 / 4,096 = 1,048,576 PIDs

This assumes that all the stacks live in the same address space, but user mode stacks from different processes most certainly do not; that's the whole point of separate address spaces! (Okay, kernel stacks live in the same address space, but the discussion about "initial stack commit" later makes it clear he's talking about user-mode stacks.)

> Since they have to be a multiple of 4:
>
> 1,048,576 / 4 = 262,144 PIDs

It's not clear why we are dividing by four here. Yes, process IDs are a multiple of four (implementation detail, not contractual, do not rely on it), but that doesn't mean that three quarters of the stacks are no longer any good. It just means that we can't use more than

4,294,967,296/4 of them since we'll run out of names after 1,073,741,824 of them. In other words, this is not a division but rather a `min` operation. And we already dropped below 1 billion when we counted kernel stacks, so this `min` step has no effect.

It's like saying, "This street is 80 meters long. The minimum building line is 4 meters, which means that you can have at most 20 houses on this side of the street. But house numbers on this side of the street must be even, so the maximum number of houses is half that, or 10." No, the requirement that house numbers be even doesn't cut the number of houses in half; it just means you have to be more careful how you assign the numbers.

> Having 262,144 processes would consume 1GB of RAM just for the initial stack commit assuming that all processes are single-threaded. If they commited 1MB of stack each you would need 256 GB of memory.

Commit does not consume RAM. Commit is merely a promise from the memory manager that the RAM will there when you need it, but the memory manager doesn't have to produce it immediately (and certainly doesn't have to keep the RAM reserved for you until you free it). Indeed, that's the whole point of virtual memory, to decouple commit from RAM! (If commit consumed RAM, then what's the page file for?)

This calculation also assumes that process IDs are allocated "smallest available first", but it's clear that it's not as simple as that: Fire up Task Manager and look at the highest process ID. (I've got one as high as 4040.) If process IDs are allocated smallest-available-first, then a process ID of 4040 implies that at some point there were 1010 processes in the system simultaneously! Unlikely.

Here's a much simpler demonstration that process IDs are not allocated smallest-available-first: Fire up Task Manager, tell it to *Show processes from all users*, go to the Processes tab, and enable the PID column if you haven't already. Now launch Calc. Look for Calc in the process list and observe that it was not assigned the lowest available PID. If your system is like mine, you have PID zero assigned to the System Idle Process (not really a process but it gets a number anyway), and PID 4 assigned to the System process (again, not really a process but it gets a number anyway), and then you have a pretty big gap before the next process ID (for me, it's 372). And yet Calc was given a process ID in the 2000's. Proof by counterexample that the system does not assign PIDs smallest-available-first.

So if they aren't assigned smallest-available-first, what's to prevent one from having a process ID of 4000000000?

(Advanced readers may note that kernel stacks do all share a single address space, but even in that case, a thread that doesn't exist doesn't have a stack. And it's clear that Igor was referring to user-mode stacks since he talked about 1MB stack commits, a value which applies to user mode and not kernel mode.)

Just for fun, I tried to see how high I could get my process ID.

```
#include <windows.h>
int __cdecl _tmain(int argc, TCHAR **argv)
{
 DWORD dwPid = 0;
 TCHAR szSelf[MAX_PATH];
 GetModuleFileName(NULL, szSelf, MAX_PATH);
 int i;
 for (i = 0; i < 10000; i++) {
  STARTUPINFO si = { 0 };
  PROCESS_INFORMATION pi;
  if (!CreateProcess(szSelf, TEXT("Bogus"),
        NULL, NULL, FALSE, CREATE_SUSPENDED, NULL, NULL,
        &si, &pi)) break;
  TerminateProcess(pi.hProcess, 0);
  CloseHandle(pi.hThread);
  // intentionally leak the process handle so the
  // process object is not destroyed
  // CloseHandle(pi.hProcess); // leak
  if (dwPid < pi.dwProcessId) dwPid = pi.dwProcessId;
 }
 _tprintf(_TEXT("\nCreated %d processes, ")
          _TEXT("highest pid seen was %d\n"), i, dwPid);
 _fgetts(szSelf, MAX_PATH, stdin);
 return 0;
}
```

In order to get the program to complete before I got bored, I ran it on a Windows 2000 virtual machine with 128MB of memory. It finally conked out at 5245 processes with a PID high water mark of 21776. Along the way, it managed to consume 2328KB of non-paged pool, 36KB of paged pool, and 36,092KB of commit. If you divide this by the number of processes, you'll see that a terminated process consumes about 450 bytes of non-paged pool, a negligible amount of paged pool, and 6KB of commit. (The commit is probably left over page tables and other detritus.) I suspect commit is the limiting factor in the number of processes.

I ran the same program on a Windows 7 machine with 1GB of RAM, and it managed to create all 10,000 processes with a high process ID of 44264. I cranked the loop limit up to 65535, and it still comfortably created 65535 processes with a high process Id of 266,232, easily exceeding the limit of 262,144 that Igor calculated.

I later learned that the Windows NT folks do try to keep the numerical values of process ID from getting too big. Earlier this century, the kernel team experimented with letting the numbers get really huge, in order to reduce the rate at which process IDs get reused, but they had to go back to small numbers, not for any technical reasons, but because people complained that the large process IDs looked ugly in Task Manager. (One customer even asked if something was wrong with his computer.)

That's not saying that the kernel folks won't go back and try the experiment again someday. After all, they managed to get rid of the dispatcher lock. Who knows what other crazy things will change next? (And once they get process IDs to go above 65535—like they were in Windows 95, by the way—or if they decided to make process IDs no longer multiples of 4 in order to keep process IDs low, this guy's program will stop working, and it'll be Microsoft's fault.)

Raymond Chen

Follow