How expensive is PssCaptureSnapshot? How fast is it? How much memory does it consume?

devblogs.microsoft.com/oldnewthing/20220314-00

March 14, 2022

Raymond Chen

A customer was investigating the possibility of using `Microsoft.Diagnostics.Runtime.dll`, commonly known as, ClrMD, in production. The documentation notes that if you use it to inspect a live process, it will use the `PssCaptureSnapshot` function to create a snapshot of the process, and then work on that snapshot.

The customer wanted to know how expensive `PssCaptureSnapshot` is when asked to create a full memory clone. They are worried that freezing the process and taking a snapshot could take a long time for their process, whose memory size could be in the tens of gigabytes. And how big is the snapshot? Is the snapshot also going to be tens of gigabytes?

The `PssCaptureSnapshot` function creates a copy-on-write clone of the original process memory. This is the same trick that Unix systems use in the `fork` system call to clone a process very quickly. The act of creating the clone takes on the order of milliseconds: It duplicates the virtual address space into the snapshot, sharing physical pages, so there's still only one copy of the process memory. And then it marks every page in the original process as copy-on-write.[1] This part is fast.

Making the page copy-on-write does mean that the snapshot consumes as much system commit as the amount of memory in the original process, because the system needs to be prepared for the case that every copy-on-write page is ultimately copied, which means that in the worst case, every page will be copied, resulting in two copies. If your system doesn't have that much commit available, creating the snapshot will fail.[2]

The customer double-checked their understanding: "So the memory blocks that are copied-on-write are those that are being actively written to during the brief time that `PssCapture-Snapshot` is running? If that's the case, then the memory cost of the snapshot is going to be relatively low in practice, right?"

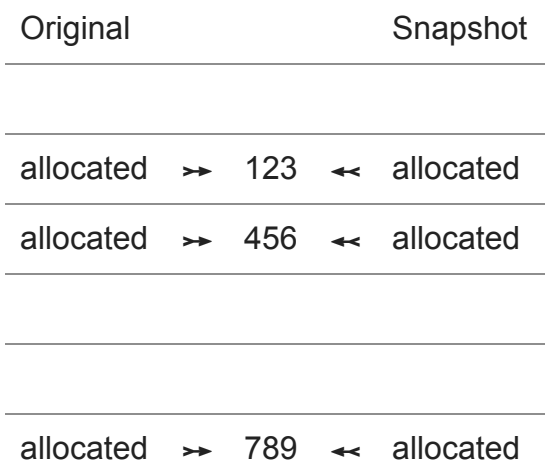
It's a good thing the customer came back to clarify, because they misunderstood.

Here's a step-by-step of what's going on. I'll say that the *original* process is the one from which the *snapshot* is being created.

1. Freeze the original process, preventing any code in the original process from running.
2. Create a snapshot process and make its address space a clone of the original process.
3. Mark every page in the original process as copy-on-write.
4. Unfreeze the original process, allowing it to resume execution.
5. Return a handle to the snapshot.

The third step is the important one. Making the pages copy-on-write in the original process means that when the original process resumes execution after unfreezing, it can't damage the copy of the memory in the snapshot: The snapshot's copy of the memory is left alone, and the original process gets a copy of the memory that it can then modify.

Let's draw a picture. Here's what the target process and its snapshot look like immediately after the snapshot is created, and before the original process is unfrozen.



The pages in the original process that are unallocated are cloned into the snapshot as unallocated pages. That's easy.

The pages in the original process that are allocated are cloned into the snapshot as identical allocated pages, sharing the same committed page (which could be a physical page or a page in the swap file, doesn't matter). The reference from the original process and the snapshow are marked copy-on-write, represented here by an arrow with a little tail.

At this point, it becomes safe to unfreeze the original process.

Suppose the original process doesn't modify any memory. The snapshot and the original process are still sharing pages, and reading from the snapshot will read the same data as reading from the original process.

But what if the original process modifies memory, say, changes the 456 to 457? If the original process and snapshot were sharing memory in the usual way, then the modification from the original process would be visible in the snapshot, which is not what we want when we create a snapshot. A snapshot is supposed to be a copy of the state of the original process as it was, unaffected by any future changes to the original process.

But since we marked the page in the original process as copy-on-write, the memory manager knows that when the original process writes to the memory, the memory manager needs to clone the page and then modify the copy. The copy gets assigned to the original process, and the snapshot keeps the unmodified page.

The result is this:

		Original				Snapshot
		allocated	➤➤	123	◀◀	allocated
457	←	allocated		456	◀◀	allocated
		allocated	➤➤	789	◀◀	allocated

What used to be a shared page containing the value 456 has now become two pages: The snapshot keeps the page with the value 456, and the original process gets a new page with the value 457.

This explains both why creating a snapshot takes only a few milliseconds (it's just creating new pointers to existing pages) and why creating a snapshot nevertheless reserves a lot of commit (in case the original program goes crazy and modifies every single page, forcing a new page to be allocated).

If you think about it, this is the same trick that the Volume Snapshot Service does to create disk snapshots quickly. It's just that the Volume Snapshot Service does it with disk clusters instead of memory.

In both cases, the actual cost of a snapshot depends on how many changes are made while the snapshot is still active. The first time a page (or cluster) is modified, you have to pay for a new one, because you're now remembering two things, the current value and the archived value.

Okay, so back to process snapshots: The process snapshot is marked with a low-priority memory configuration, which means that the system will limit how fast you can read from it. This tries to limit the impact that process snapshots have on the rest of the system. One of my colleagues did an experiment and found that the top speed of reading from a process snapshot was around 30 megabytes per second, or about the same as a 24x DVD.[3]

The hope is that you aren't going to be reading the entire contents of the snapshot, but rather walking through the memory in the snapshot, maybe following pointers to other memory, and picking out the details you need, possibly digging deeper into things that you discover which are important. For example, enumerating all the loaded DLLs doesn't access all of the memory; it accesses only the loaded module list. Walking the heap will access more memory, but at least your memory access is limited to the heap, not the entire process.

On the other hand, if you created the clone so you can create a crash dump, then that memory speed limit is going to mean that creating a full memory crash dump from a snapshot is going to take a long time.

Bonus chatter: What happens when I free the snapshot? The original process still has all these arrows with tails. Are they also going to cause extra pages to be allocated?

Here's what things look like when the snapshot is freed:

Some of the arrows are normal arrows, corresponding to pages that were already copied on write. But most of the arrows are still copy-on-write arrows (with little tails). What happens when the process tries to change the 123 to 124?

In theory, the memory manager would copy the page and modify the copy. And then it would realize that the original 123 page has no more references, so it would free the 123 page. In practice, the memory manager optimizes this code path and realizes that there's no point copying a page if you're going to throw away the source page anyway. You may as well just reuse that page for the copy!

Original
allocated → 124
allocated → 457
allocated ↠ 789

All the memory manager does is erase the tail. It's a regular arrow again.

[1] For simplicity of exposition, I'm considering only writable pages. Read-only pages are marked "currently read-only, but convert to copy-on-write if they become writable." This is hardly an unusual state. You get it all the time without realizing it. When multiple processes load a DLL, the mapping is read-only and is shared among all clients. If a process changes the memory protection to read/write, then it gets a copy-on-write version of the page that is shared with all the other processes until it is written to, at which point it turns into a private copy.

[2] Windows does not overcommit. It is always prepared for a "run on the bank", where each owner of every committed page comes to the memory manager and says, "So where's that memory you promised me?"

[3] You can test it out yourself:

```
procdump -ma -n 10 -s 2 -r devenv
```

This asks for 10 dumps (`-n 10`) to be created at two-second intervals (`-s 2`) using snapshots (`-r`). You can watch the disk performance metrics to see how fast the data gets out of the snapshot and onto the disk. My colleague likes to pick media player apps as victims of these types of experiments, to make sure that the snapshot creation is fast enough that there are no playback glitches.

Raymond Chen

Follow