In defence of swap: common misconceptions

source : https://chrisdown.name/2018/01/02/in-defence-of-swap.html

tl;dr:

1. Having swap is a reasonably important part of a well functioning system. Without it, sane memory management becomes harder to achieve.
2. Swap is not generally about getting emergency memory, it's about making memory reclamation egalitarian and efficient. In fact, using it as "emergency memory" is generally actively harmful.
3. Disabling swap does not prevent disk I/O from becoming a problem under memory contention, it simply shifts the disk I/O thrashing from anonymous pages to file pages. Not only may this be less efficient, as we have a smaller pool of pages to select from for reclaim, but it may also contribute to getting into this high contention state in the first place.
4. The swapper on kernels before 4.0 has a lot of pitfalls, and has contributed to a lot of people's negative perceptions about swap due to its overeagerness to swap out pages. On kernels >4.0, the situation is significantly better.
5. On SSDs, swapping out anonymous pages and reclaiming file pages are essentially equivalent in terms of performance/latency. On older spinning disks, swap reads are slower due to random reads, so a lower `vm.swappiness` setting makes sense there (read on for more about `vm.swappiness`).
6. Disabling swap doesn't prevent pathological behaviour at near-OOM, although it's true that having swap may prolong it. Whether the system global OOM killer is invoked with or without swap, or was invoked sooner or later, the result is the same: you are left with a system in an unpredictable state. Having no swap doesn't avoid this.
7. You can achieve better swap behaviour under memory pressure and prevent thrashing using `memory.low` and friends in cgroup v2.

As part of my work improving and using cgroup v2, I've been talking to a lot of engineers about attitudes towards memory management, especially around application behaviour under pressure and operating system heuristics used under the hood for memory management.

A repeated topic in these discussions has been swap. Swap is a hotly contested and poorly understood topic, even by those who have been working with Linux for many years. Many see it as useless or actively harmful: a relic of a time where memory was scarce, and disks were a necessary evil to provide much-needed space for paging. This is a statement that I still see being batted around with relative frequency in recent years, and I've had many discussions with colleagues, friends, and industry peers to help them understand why swap is still a useful concept on modern computers with significantly more physical memory available than in the past.

There's also a lot of misunderstanding about the purpose of swap – many people just see it as a kind of "slow extra memory" for use in emergencies, but don't understand how it can contribute during normal load to the healthy operation of an operating system as a whole.

Many of us have heard most of the usual tropes about memory: "Linux uses too much memory", "swap should be double your physical memory size", and the like. While these are either trivial to dispel, or discussion around them has become more nuanced in recent years, the myth of "useless" swap is much more grounded in heuristics and arcana rather than something that can be explained by simple analogy, and requires somewhat more understanding of memory management to reason about.

This post is mostly aimed at those who administrate Linux systems and are interested in hearing the counterpoints to running with undersized/no swap or running with `vm.swappiness` set to 0.

Background

It's hard to talk about why having swap and swapping out pages are good things in normal operation without a shared understanding of some of the basic underlying mechanisms at play in Linux memory management, so let's make sure we're on the same page.

Types of memory

There are many different types of memory in Linux, and each type has its own properties. Understanding the nuances of these is key to understanding why swap is important.

- For example, there are *pages* ("blocks" of memory, typically 4k) responsible for holding the code for each process being run on your computer.

- There are also pages responsible for caching data and metadata related to files accessed by those programs in order to speed up future access. These are part of the page cache, and I will refer to them as *file* memory.
- There are also pages which are responsible for the memory allocations made inside that code, for example, when new memory that has been allocated with `malloc` is written to, or when using `mmap`'s `MAP_ANONYMOUS` flag. These are "anonymous" pages – so called because they are not backed by anything – and I will refer to them as *anon* memory.
- There are other types of memory too – *shared* memory, *slab* memory, *kernel stack* memory, *buffers*, and the like – but anonymous memory and file memory are the most well known and easy to understand ones, so I will use these in my examples, although they apply equally to these types too.

Reclaimable/unreclaimable memory

One of the most fundamental questions when thinking about a particular type of memory is whether it is able to be reclaimed or not. "Reclaim" here means that the system can, without losing data, purge pages of that type from physical memory.

For some page types, this is typically fairly trivial. For example, in the case of *clean* (unmodified) page cache memory, we're simply caching something that we have on disk for performance, so we can drop the page without having to do any special operations.

For some page types, this is possible, but not trivial. For example, in the case of *dirty* (modified) page cache memory, we can't just drop the page, because the disk doesn't have our modifications yet. As such we either need to deny reclamation or first get our changes back to disk before we can drop this memory.

For some page types, this is not possible. For example, in the case of the anonymous pages mentioned previously, they only exist in memory and in no other backing store, so they have to be kept there.

On the nature of swap

If you look for descriptions of the purpose of swap on Linux, you'll inevitably find many people talking about it as if it is merely an extension of the physical RAM for use in emergencies. For example, here is a random post I got as one of the top results from typing "what is swap" in Google:

> ```
> Swap is essentially emergency memory; a space set aside for times
> when your system temporarily needs more physical memory than you have
> available in RAM. It's considered "bad" in the sense that it's slow
> ```

> and inefficient, and if your system constantly needs to use swap then it obviously doesn't have enough memory. […] If you have enough RAM to handle all of your needs, and don't expect to ever max it out, then you should be perfectly safe running without a swap space.

To be clear, I don't blame the poster of this comment at all for the content of their post – this is accepted as "common knowledge" by a lot of Linux sysadmins and is probably one of the most likely things that you will hear from one if you ask them to talk about swap. It is unfortunately also, however, a misunderstanding of the purpose and use of swap, especially on modern systems.

Above, I talked about reclamation for anonymous pages being "not possible", as anonymous pages by their nature have no backing store to fall back to when being purged from memory – as such, their reclamation would result in complete data loss for those pages. What if we could create such a store for these pages, though?

Well, this is precisely what swap is for. Swap is a storage area for these seemingly "unreclaimable" pages that allows us to page them out to a storage device on demand. This means that they can now be considered as equally eligible for reclaim as their more trivially reclaimable friends, like clean file pages, allowing more efficient use of available physical memory.

Swap is primarily a mechanism for equality of reclamation, not for emergency "extra memory". Swap is not what makes your application slow – entering overall memory contention is what makes your application slow.

So in what situations under this "equality of reclamation" scenario would we legitimately choose to reclaim anonymous pages? Here are, abstractly, some not uncommon scenarios:

1. During initialisation, a long-running program may allocate and use many pages. These pages may also be used as part of shutdown/cleanup, but are not needed once the program is "started" (in an application-specific sense). This is fairly common for daemons which have significant dependencies to initialise.
2. During the program's normal operation, we may allocate memory which is only used rarely. It may make more sense for overall system performance to required a major fault to page these in from disk on demand, instead using the memory for something else that's more important.

Examining what happens with/without swap

Let's look at typical situations, and how they perform with and without swap present. I talk about metrics around "memory contention" in my talk on cgroup v2.

Under no/low memory contention

- **With swap:** We can choose to swap out rarely-used anonymous memory that may only be used during a small part of the process lifecycle, allowing us to use this memory to improve cache hit rate, or do other optimisations.
- **Without swap:** We cannot swap out rarely-used anonymous memory, as it's locked in memory. While this may not immediately present as a problem, on some workloads this may represent a non-trivial drop in performance due to stale, anonymous pages taking space away from more important use.

Under moderate/high memory contention

- **With swap:** All memory types have an equal possibility of being reclaimed. This means we have more chance of being able to reclaim pages successfully – that is, we can reclaim pages that are not quickly faulted back in again (thrashing).
- **Without swap:** Anonymous pages are locked into memory as they have nowhere to go. The chance of successful long-term page reclamation is lower, as we have only some types of memory eligible to be reclaimed at all. The risk of page thrashing is higher. The casual reader might think that this would still be better as it might avoid having to do disk I/O, but this isn't true – we simply transfer the disk I/O of swapping to dropping hot page caches and dropping code segments we need soon.

Under temporary spikes in memory usage

- **With swap:** We're more resilient to temporary spikes, but in cases of severe memory starvation, the period from memory thrashing beginning to the OOM killer may be prolonged. We have more visibility into the instigators of memory pressure and can act on them more reasonably, and can perform a controlled intervention.
- **Without swap:** The OOM killer is triggered more quickly as anonymous pages are locked into memory and cannot be reclaimed. We're more likely to thrash on memory, but the time between thrashing and OOMing is reduced. Depending on your application, this may be better or worse. For example, a queue-based application may desire this quick transfer from thrashing to killing. That said, this is still too late to be really useful – the OOM killer is only invoked at moments of severe starvation, and relying on this method for such behaviour would be better replaced with more opportunistic killing of processes as memory contention is reached in the first place.

Ok, so I want system swap, but how can I tune it for individual applications?

You didn't think you'd get through this entire post without me plugging cgroup v2, did you? ;-)

Obviously, it's hard for a generic heuristic algorithm to be right all the time, so it's important for you to be able to give guidance to the kernel. Historically the only tuning you could do was at the system level, using `vm.swappiness`. This has two problems: `vm.swappiness` is incredibly hard to reason about because it only feeds in as a small part of a larger heuristic system, and it also is system-wide instead of being granular to a smaller set of processes.

You can also use `mlock` to lock pages into memory, but this requires either modifying program code, fun with `LD_PRELOAD`, or doing horrible things with a debugger at runtime. In VM-based languages this also doesn't work very well, since you generally have no control over allocation and end up having to `mlockall`, which has no precision towards the pages you actually care about.

cgroup v2 has a tunable per-cgroup in the form of `memory.low`, which allows us to tell the kernel to prefer other applications for reclaim below a certain threshold of memory used. This allows us to not prevent the kernel from swapping out parts of our application, but prefer to reclaim from other applications under memory contention. Under normal conditions, the kernel's swap logic is generally pretty good, and allowing it to swap out pages opportunistically generally increases system performance. Swap thrash under heavy memory contention is not ideal, but it's more a property of simply running out of memory entirely than a problem with the swapper. In these situations, you typically want to fail fast by self-killing non-critical processes when memory pressure starts to build up.

You can not simply rely on the OOM killer for this. The OOM killer is only invoked in situations of dire failure when we've *already* entered a state where the system is severely unhealthy and may well have been so for a while. You need to opportunistically handle the situation yourself before ever thinking about the OOM killer.

Determination of memory pressure is somewhat difficult using traditional Linux memory counters, though. We have some things which seem somewhat related, but are merely tangential – memory usage, page scans, etc – and from these metrics alone it's very hard to tell an efficient memory configuration from one that's trending towards memory contention. There is a group of us at Facebook, spearheaded by Johannes, working on developing new metrics that expose memory pressure more easily that should help with this in future. If you're interested in hearing more about this, I go into detail about one metric being considered in my talk on cgroup v2.

Tuning

How much swap do I need, then?

In general, the minimum amount of swap space required for optimal memory management depends on the number of anonymous pages pinned into memory that are rarely reaccessed by

an application, and the value of reclaiming those anonymous pages. The latter is mostly a question of which pages are no longer purged to make way for these infrequently accessed anonymous pages.

If you have a bunch of disk space and a recent (4.0+) kernel, more swap is almost always better than less. In older kernels `kswapd`, one of the kernel processes responsible for managing swap, was historically very overeager to swap out memory aggressively the more swap you had. In recent times, swapping behaviour when a large amount of swap space is available has been significantly improved. If you're running kernel 4.0+, having a larger swap on a modern kernel size should not result in overzealous swapping. As such, *if you have the space, having a swap size of a few GB keeps your options open on modern kernels.*

If you're more constrained with disk space, then the answer really depends on the tradeoffs you have to make, and the nature of the environment. Ideally you should have enough swap to make your system operate optimally at normal and peak (memory) load. What I'd recommend is setting up a few testing systems with 2-3GB of swap or more, and monitoring what happens over the course of a week or so under varying (memory) load conditions. As long as you haven't encountered severe memory starvation during that week – in which case the test will not have been very useful – you will probably end up with some number of MB of swap occupied. As such, it's probably worth having at least that much swap available, in addition to a little buffer for changing workloads. `atop` in logging mode can also show you which applications are having their pages swapped out in the `SWAPSZ` column, so if you don't already use it on your servers to log historic server state you probably want to set it up on these test machines with logging mode as part of this experiment. This also tells you *when* your application started swapping out pages, which you can tie to log events or other key data.

Another thing worth considering is the nature of the swap medium. Swap reads tend to be highly random, since we can't reliably predict which pages will be refaulted and when. On an SSD this doesn't matter much, but on spinning disks, random I/O is extremely expensive since it requires physical movement to achieve. On the other hand, refaulting of file pages is likely less random, since files related to the operation of a single application at runtime tend to be less fragmented. This might mean that on a spinning disk you may want to bias more towards reclaiming file pages instead of swapping out anonymous pages, but again, you need to test and evaluate how this balances out for your workload.

For laptop/desktop users who want to hibernate to swap, this also needs to be taken into account – in this case your swap file should be at least your physical RAM size.

What should my swappiness setting be?

First, it's important to understand what `vm.swappiness` does. `vm.swappiness` is a sysctl that biases memory reclaim either towards reclamation of anonymous pages, or towards file pages.

It does this using two different attributes: file_prio (our willingness to reclaim file pages) and anon_prio (our willingness to reclaim anonymous pages). vm.swappiness plays into this, as it becomes the default value for anon_prio, and it also is subtracted from the default value of 200 for file_prio, which means for a value of vm.swappiness = 50, the outcome is that anon_prio is 50, and file_prio is 150 (the exact numbers don't matter as much as their relative weight compared to the other).

This means that, in general, vm.swappiness *is simply a ratio of how costly reclaiming and refaulting anonymous memory is compared to file memory for your hardware and workload*.

- The lower the value, the more you tell the kernel that infrequently accessed anonymous pages are expensive to swap out and in on your hardware.
- The higher the value, the more you tell the kernel that the cost of swapping anonymous pages and file pages is similar on your hardware.

The memory management subsystem will still try to mostly decide whether it swaps file or anonymous pages based on how hot the memory is, but swappiness tips the cost calculation either more towards swapping or more towards dropping filesystem caches when it could go either way. On SSDs these are basically as expensive as each other, so setting vm.swappiness = 100 (full equality) may work well. On spinning disks, swapping may be significantly more expensive since swapping in generally requires random reads, so you may want to bias more towards a lower value.

The reality is that most people don't really have a feeling about which their hardware demands, so it's non-trivial to tune this value based on instinct alone – this is something that you need to test using different values. You can also spend time evaluating the memory composition of your system and core applications and their behaviour under mild memory reclamation.

When talking about vm.swappiness, an extremely important change to consider from recent(ish) times is this change to vmscan by Satoru Moriya in 2012, which changes the way that vm.swappiness = 0 is handled quite significantly.

Essentially, the patch makes it so that we are extremely biased against scanning (and thus reclaiming) any anonymous pages at all with vm.swappiness = 0, unless we are already encountering severe memory contention. As mentioned previously in this post, that's generally not what you want, since this prevents equality of reclamation prior to extreme memory pressure occurring, which may actually *lead* to this extreme memory pressure in the first place. vm.swappiness = 1 is the lowest you can go without invoking the special casing for anonymous page scanning implemented in that patch.

The kernel default here is vm.swappiness = 60. This value is generally not too bad for most workloads, but it's hard to have a general default that suits all workloads. As such, a valuable extension to the tuning mentioned in the "how much swap do I need" section above would be

to test these systems with differing values for `vm.swappiness`, and monitor your application and system metrics under heavy (memory) load. Some time in the near future, once we have a decent implementation of refault detection in the kernel, you'll also be able to determine this somewhat workload-agnostically by looking at cgroup v2's page refaulting metrics.

In conclusion

- Swap is a useful tool to allow equality of reclamation of memory pages, but its purpose is frequently misunderstood, leading to its negative perception across the industry. If you use swap in the spirit intended, though – as a method of increasing equality of reclamation – you'll find that it's a useful tool instead of a hindrance.
- Disabling swap does not prevent disk I/O from becoming a problem under memory contention, it simply shifts the disk I/O thrashing from anonymous pages to file pages. Not only may this be less efficient, as we have a smaller pool of pages to select from for reclaim, but it may also contribute to getting into this high contention state in the first place.
- Swap can make a system slower to OOM kill, since it provides another, slower source of memory to thrash on in out of memory situations – the OOM killer is only used by the kernel as a last resort, after things have already become monumentally screwed. The solutions here depend on your system:
    - You can opportunistically change the system workload depending on cgroup-local or global memory pressure. This prevents getting into these situations in the first place, but solid memory pressure metrics are lacking throughout the history of Unix. Hopefully this should be better soon with the addition of refault detection.
    - You can bias reclaiming (and thus swapping) away from certain processes per-cgroup using `memory.low`, allowing you to protect critical daemons without disabling swap entirely.