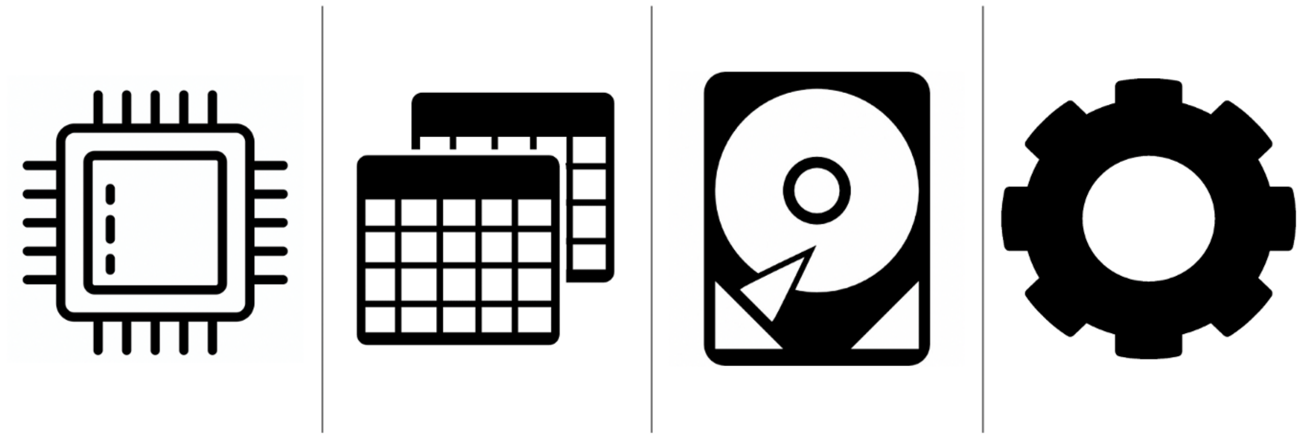
Published in Marionete

Rui Grafino Follow
Dec 16, 2020 · 11 min read · Listen

Linux Disk Cache was always there...



Native cache on improving read/write performance and how to improve its benefits while being more aware

Table of contents

Disclaimer:

Some of the explanations and description are far more abbreviated and simpler than what's really occurring under the hood for the sake of simplicity and understanding of those not much involved or interested in the subject. Avoiding a deep dive into technical details like pages, block sizes or more complex terms and operations is deliberate.

Linux Read/Write Performance

While managing memory the Linux Kernel uses a native caching mechanism called **page cache** or **disk cache** to improve performance of reads and writes.

To put it simple: Its main purpose is to copy data and binary files from storage to memory, thus reducing disk I/O and improving overall performance. This is especially true if there is some kind of workloads that opens frequently the same files or some other kind of I/O expensive operations.

The page cache is also used while writing to disk, not only reading, but we will get there in a moment.

By default, all free physical memory is used by the operating system for the page cache purposes and depending on the workloads the Operating System manages its state, caching, re-using and evicting files as needed.

Even with the current faster Solid-State Drives throughput having the files cached in memory brings a performance improvement to the system.

It's also quite common some kind of misconception that the Operating System (Linux, Unix or BSD based), specially on mobile users that we might be short on free memory and that it needs some kind of optimisation to free memory as time elapses. In fact

Important to keep always present that a cached memory is always a free available memory that will be reclaimed as needed without any penalty for those new running processes requiring it.

An optimum system is a system making use of practically all available resources (mainly true for RAM, CPU) for whats intended to do, immediately before any type of contention or decreased performance penalty start to emerge.

Simple Reads

Let's look at a simple example using a 2GB file. For the sake of simplicity, the VM specs at the moment are not relevant, meaning that for now we just want to make sure we have **more available memory than the size of the file** we are reading.

First, we check our memory status to see how much we have available overall, buffers and cache values.

```
# free -wh
       total  used   free   shared   buffers  cache   available
Mem:    5.8G   94M   5.7G     608K      2.1M    49M        5.6G
Swap:     0B    0B     0B
```

We them generate a dummy file with 2GB in size and flush and delete all caches that might have occurred in the operation.

```
# head -c 2G </dev/urandom > dummy.file
# echo 3 > /proc/sys/vm/drop_caches
```

Then we count the number of lines in the file timing the execution duration:

```
# time wc -l dummy.file
8387042 dummy.file
real 0m3.731s
user 0m0.278s
```

```
# time wc -l dummy.file
8387042 dummy.file
real 0m1.045s
user 0m0.575s
sys 0m0.471s
```

As we can clearly see the last execution took much less time and that's because the file was cached in the page cache and that just by itself improved the reading operation. Let's confirm the memory status values again.

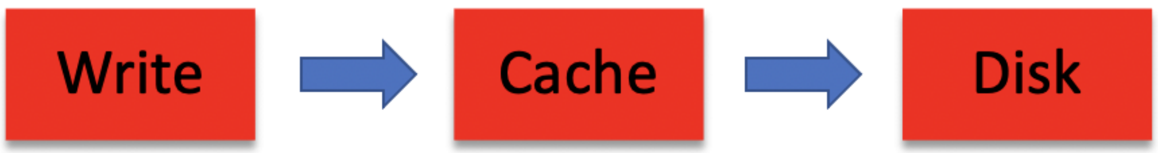
```
# free -wh
     total    used    free    shared    buffers    cache   available
Mem:   5.8G     94M    3.7G     608K        2.1M     2.1G         5.5G
Swap:    0B      0B      0B
```

Now we can see that we have 2GB less in the free memory and now a lot more usage on buffers and cache.

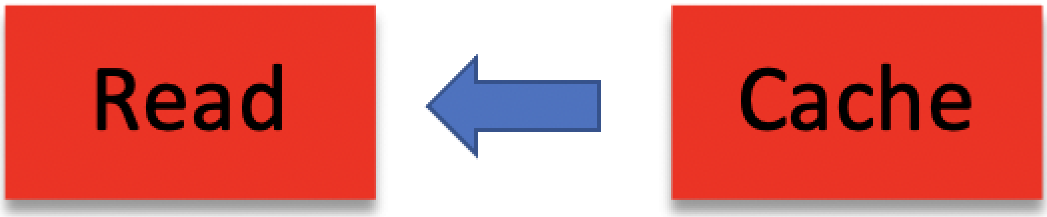
The **cache** is the part of the memory which stores files and binaries, like shared libs, data so that future requests for that data can be served faster.

The **buffers** are metadata related to the cache.

Non cached read



Cached read



Now we will clean the caches again so that we return to our original state, but we will use a different command so that we are all aware of this possibility.

```
# sysctl -w vm.drop_caches=3
```

Doing the line count again takes the full time required to put the file in memory again, but before doing that let use a command to see if the file is cached.

The command **vmtouch** must be installed separately and can be used to check if a file or part of it is cached.

```
# vmtouch -v dummy.file
dummy.file
[                                                                ] 0/524288
Files: 1
Directories: 0
Resident Pages: 0/524288 0/2G 0%
Elapsed: 0.013999 seconds
```

Doing the line count after confirming that the file is not cached and is 0%.

```
# time wc -l dummy.file
8387042 dummy.file
real 0m3.732s
user 0m0.284s
sys 0m1.539s
```

```
# vmtouch -v dummy.file
dummy.file
[OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO] 524288/524288
Files: 1
Directories: 0
Resident Pages: 524288/524288 2G/2G 100%
Elapsed: 0.024309 seconds
```

The file is fully cached so the next reads will be faster.

Appending to a cached file

If you append or change information on a cached file the end result will also be cached, if the available memory does not limit it. And following on the same rational the **small.file** was also cached since it was also read.

```
# cat small.file >> dummy.file

# vmtouch -v dummy.file
dummy.file
[OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO] 786433/786433
Files: 1
Directories: 0
Resident Pages: 786433/786433 3G/3G 100%
Elapsed: 0.035199 seconds
```

To those who haven't noticed this was a write operation, so let's move to understand how write and cache works.

Writing files

Like we mentioned earlier the page cache is also used when we write to disk to improve overall performance.

In Linux when we create a file, that file is initially written to page cache and only after some time or condition the file is flushed to disk.

It's worth mention that this behaviour will depend on several factors like the type of

- The files are immediately flushed to disk.
- Possible performance issues if I/O subsystem or scheduler is very busy.
- Many Storage and Databases systems use this method so that they control the Writes and not the Operating System.
- Some filesystems support it by mounting flags.

Buffered I/O

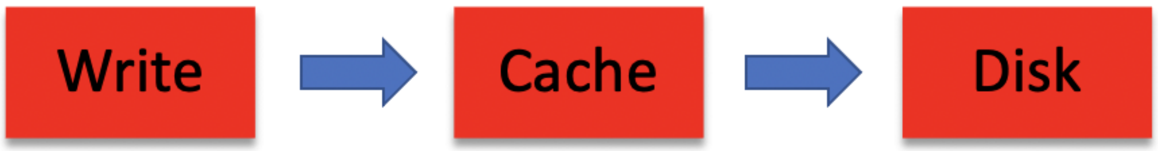
- Uses the file system buffer cache.
- Much faster writes, because it's done to memory first.
- Creates some temporary dirty pages.
- Potential loss or corruption of data if power is suddenly unplugged. (very rare)

With page cache we are using Buffered I/O access, let's look at a simple example:

```
# free -wh
     total  used   free  shared   buffers  cache   available
Mem:  5.8G   92M   5.7G   608K      1.9M.   50M        5.6G
Swap:   0B    0B     0B

# head -c 1G </dev/urandom > small.file

# free -wh
     total  used   free  shared   buffers   cache  available
Mem:  5.8G   92M   4.7G   608K      1.9M    1.1G        5.5G
Swap: 0B      0B     0B
```

If there are no memory constraints the whole file is first written to memory and keeps **Dirty Pages** until is fully flushed.

Dirty Pages is the amount of information that is on page cache that is not yet synced to disk. The Kernel will eventually flush and sync all in memory information to disk.



We can check the amount of data the is dirty:

```
# cat /proc/meminfo | grep Dirty
Dirty: 369536 kB
```

We can also force the **sync** to occur.

```
# sync

# cat /proc/meminfo | grep Dirty
Dirty: 0 kB
```

Caching multiple files

Now let's look into a more complex example where we have multiple files and the sum size of all those files is greater than the available memory for caching.

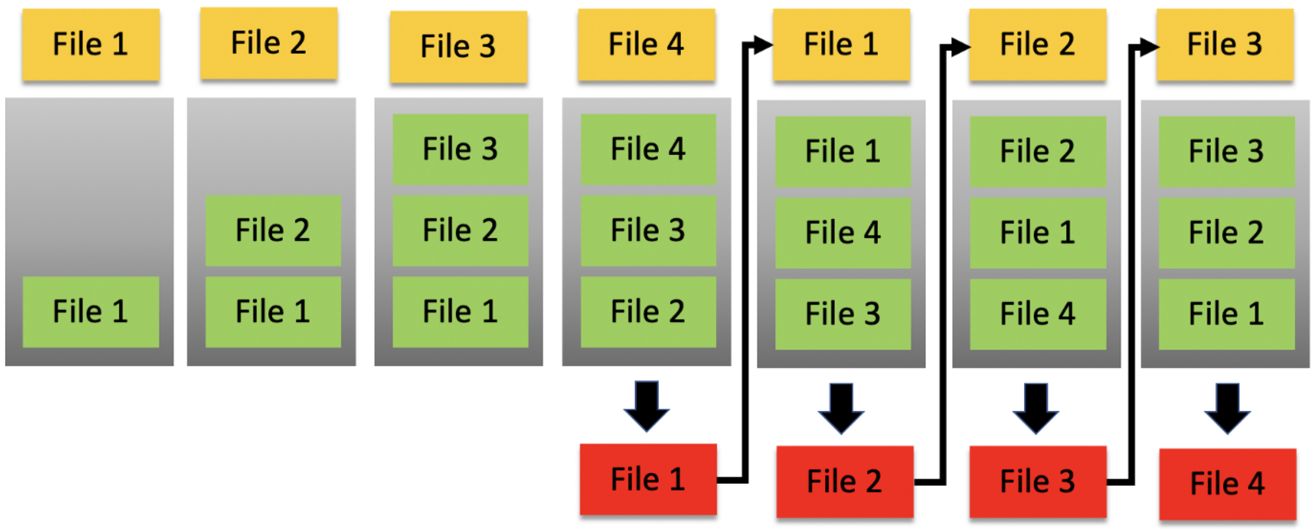
Let's create 4 files of 2GB each, but now there is only available cache memory for 3 of them:

```
# head -c 2G </dev/urandom > dummy.file1
# head -c 2G </dev/urandom > dummy.file2
# head -c 2G </dev/urandom > dummy.file3
# head -c 2G </dev/urandom > dummy.file4
```

As we **read (yellow)** each file in order they will be **stacked (green)** on page cache.

When we read the next file and there is not enough memory for all 4 files to be cached

To put it simply the one that gets pushed out of the cache now is the Least Recently Used aka **LRU** in **Page Replacement Policy,** replacing the cache entry that has not been used for the longest time in the past.



It's important to mention that the exact algorithm is more complex that just a **LRU** and involves more than one list, for more information refer to the current kernel documentation.

If we reuse an existing cached file it will go up in the stack as is a more frequently accessed file.

Depending on the number of files, their size and available memory, the cached content can be just some portion of file/s.

We can see that with the **vmtouch** command:

```
# vmtouch -v dummy.file1
dummy.file1
[                                           oOOOOO] 45414/524288
Files: 1
Directories: 0
Resident Pages: 45414/524288 177M/2G 8.66%
Elapsed: 0.035156 seconds
```

What happens to page cache when we read a file much bigger that the available memory on our system?

If a file is much bigger than the available memory by now should be clear that only portions of the file will be kept in page cache. But how it will impact performance while computing data from that file?

Let’s try to cache a big file

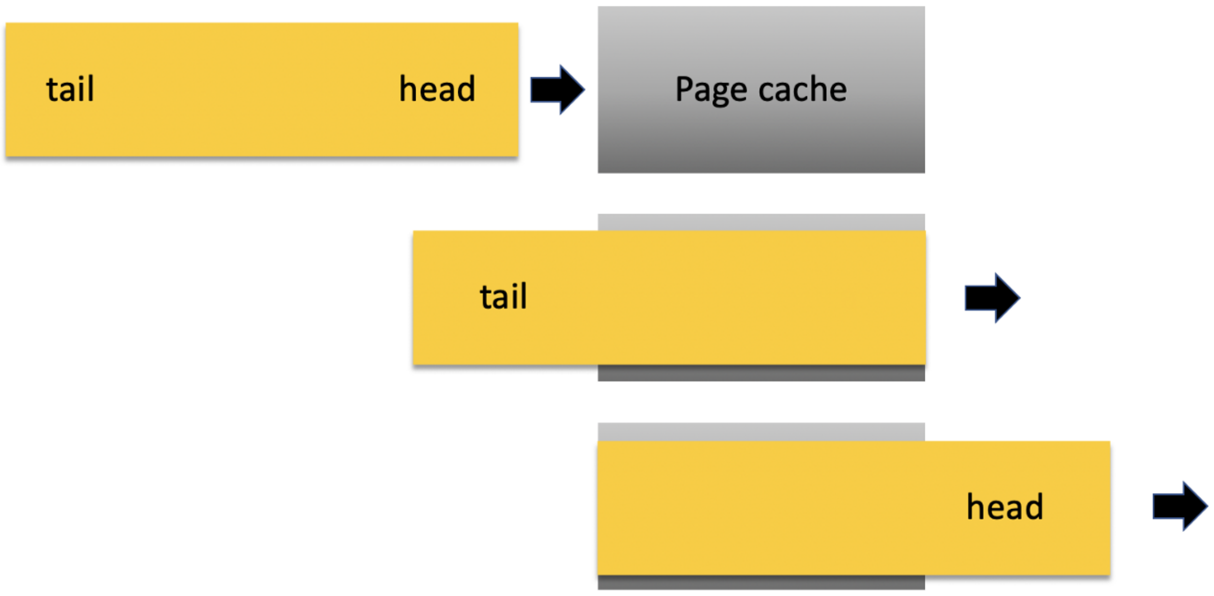
```
# time wc -l big.file
33544011 big.file
real 0m14.426s
user 0m1.202s
sys 0m6.737s

# vmtouch -v big.file
big.file
[         oOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO] 1621005/2097152
Files: 1
Directories: 0
Resident Pages: 1621005/2097152 6G/8G 77.3%
Elapsed: 0.077948 seconds
```

The file is 8GB but as we can see from the output of **vmtouch** only 6GB could fit in page cache. What does this mean in performance? Lets reprocess the file to see if we gain performance.

```
# time wc -l big.file
33544011 big.file
real 0m14.260s
user 0m1.959s
sys 0m4.741s
```

The performance didn’t improve while reading the hole file again. But why?

As the file is being read and doing the read caching it doesn’t fit all in memory like we saw in **vmtouch**. What is happening is a kind of sliding window and the beginning of



The file head is out of cache, but if we try to read the file tail?

```
# time tail -5000000 big.file >/dev/null
real 0m4.696s
user 0m2.687s
sys 0m2.001s
```

How this compares with an un-cached tail? Let’s clean all caches and do just the same tail.

```
# sysctl -w vm.drop_caches=3

# time tail -5000000 big.file >/dev/null
real 0m34.682s
user 0m2.987s
sys 0m4.379s

# time tail -5000000 big.file >/dev/null
real 0m4.524s
user 0m2.649s
sys 0m1.873s

# vmtouch -v big.file
```

```
Resident Pages: 312460/2097152 1G/8G 14.9%
Elapsed: 0.080427 seconds
```

Clearly, we had a huge benefit from doing a tail when he read the whole file because although it didn't fit all in page cache, the **portion we were using was cached**.

As processing the end of file worked, the same would happen if we did just that for the head or some other part of the file.

We need to have this in mind for big data files as we can clearly get a performance improvement if we have cached the portions of the file we intend to work with.

Some beforehand strategy in caching files can increase performance of read and writes.

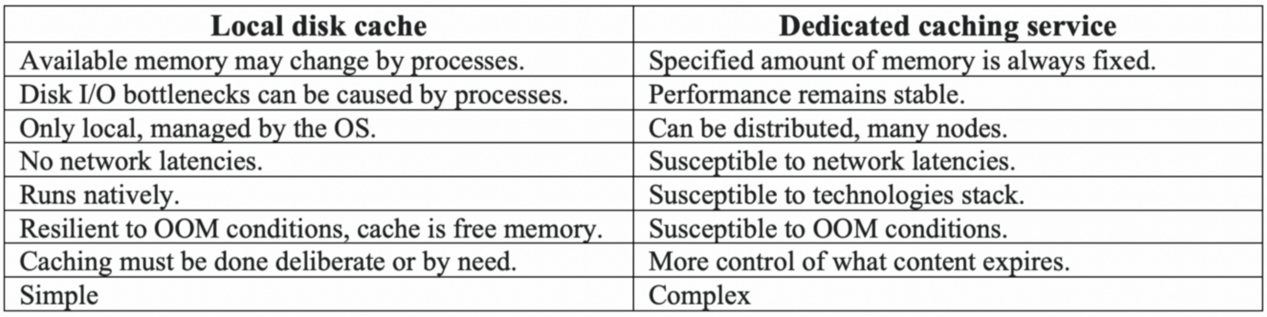
Native Page Cache vs Dedicated Caching Services

How does Linux page cache perform vs dedicated caching services, either storage cache or content based?

First, we need to understand that a dedicated caching services or a in memory processing service serves a very specific purpose and workload, and any direct comparison is always unbalanced and, in some cases, does not apply.

But for sure any local disk caching performed by the kernel itself will always be faster than using a local dedicated service on top of it, and much faster than using a remote caching server.

Any of these services have (most of the times) is configured on top of Linux itself. Specially if it is a distributed service not only, we have to account for all coordination and discovery processing between nodes but also all possible network latencies and roundtrips.



Local disk cache	Dedicated caching service
Available memory may change by processes.	Specified amount of memory is always fixed.
Disk I/O bottlenecks can be caused by processes.	Performance remains stable.
Only local, managed by the OS.	Can be distributed, many nodes.
No network latencies.	Susceptible to network latencies.
Runs natively.	Susceptible to technologies stack.
Resilient to OOM conditions, cache is free memory.	Susceptible to OOM conditions.
Caching must be done deliberate or by need.	More control of what content expires.
Simple	Complex

Some different types of caching:

Client-side caching

- Disk Cache
- DNS Cache
- Browser Cache

Network based caching

- Content Delivery Networks
- Web Proxy servers

Server level Caching

- Webserver Caching
- Application Caching or In Memory Processing
- Database and Storage Caching

Summary

Page cache its simple, its native to the OS, and provides clear performance improvements while reading and writing data. Plays an important role in the OS native processes and sharing data and libs between processes optimising kernel and user space executions.

Many services make use of it natively and some of them implement different layers of caching for specific workloads, especially databases.

Much more information could have been explored like the memory management subsystem, the cache and performance tuning, managing swap, different filesystems, kernel parameters and specific caching services deep dives but that was not the purpose for this article.

References

Welcome to LWN.net

Security] Posted Dec 14, 2020 16:02 UTC (Mon) by ris Security updates have been issued by Debian (lxml, openexr...

lwn.net

Documentation for /proc/sys/vm/ - The Linux Kernel documentation

kernel version 2.6.29 For general info and legal blurb, please look in index.rst. This file contains the documentation...

www.kernel.org

Linux - Memory Management insights

Nowadays the Linux memory management of a SAP system (application server) or SAP HANA system getting more important...

blogs.sap.com

Linux_Kernel_Newbies - Linux Kernel Newbies

Kernelnewbies can be found on the MailingList, IRC (irc.oftc.net #kernelnewbies), and this wiki. So you want to be a...

kernelnewbies.org

en.wikipedia.org