Experiments and fun with the Linux disk cache

Hopefully you are now convinced that Linux didn't just eat your ram. Here are some interesting things you can do to learn how the disk cache works.

Effects of disk cache on application memory allocation

Since I've already promised that disk cache doesn't prevent applications from getting the memory they want, let's start with that. Here is a C app (munch.c) that gobbles up as much memory as it can, or to a specified limit:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char** argv) {
    int max = -1;
    int mb = 0;
    char* buffer;

    if(argc > 1)
        max = atoi(argv[1]);

    while((buffer=malloc(1024*1024)) != NULL && mb != max) {
        memset(buffer, 0, 1024*1024);
        mb++;
        printf("Allocated %d MB\n", mb);
    }

    return 0;
}
```

Running out of memory isn't fun, but the OOM killer should end just this process and hopefully the rest will remain undisturbed. We'll definitely want to disable swap for this, or the app will gobble up that as well.

```
$ sudo swapoff -a

$ free -m
             total       used       free     shared    buffers     cached
Mem:          1504       1490         14          0         24        809
-/+ buffers/cache:        656        848
Swap:            0          0          0

$ gcc munch.c -o munch

$ ./munch
Allocated 1 MB
Allocated 2 MB
(...)
Allocated 877 MB
Allocated 878 MB
Allocated 879 MB
Killed

$ free -m
```

```
             total       used       free     shared    buffers     cached
Mem:          1504        650        854          0          1         67
-/+ buffers/cache:        581        923
Swap:            0          0          0

$
```

Even though it said 14MB "free", that didn't stop the application from grabbing 879MB. Afterwards, the cache is pretty empty[2], but it will gradually fill up again as files are read and written. Give it a try.

Effects of disk cache on swapping

I also said that disk cache won't cause applications to use swap. Let's try that as well, with the same 'munch' app as in the last experiment. This time we'll run it with swap on, and limit it to a few hundred megabytes:

```
$ free -m
             total       used       free     shared    buffers     cached
Mem:          1504       1490         14          0         10        874
-/+ buffers/cache:        605        899
Swap:         2047          6       2041

$ ./munch 400
Allocated 1 MB
Allocated 2 MB
(...)
Allocated 399 MB
Allocated 400 MB

$ free -m
             total       used       free     shared    buffers     cached
Mem:          1504       1090        414          0          5        485
-/+ buffers/cache:        598        906
Swap:         2047          6       2041
```

munch ate 400MB of ram, which was taken from the disk cache without resorting to swap. Likewise, we can fill the disk cache again and it will not start eating swap either. If you run `watch free -m` in one terminal, and `find . -type f -exec cat {} + > /dev/null` in another, you can see that "cached" will rise while "free" falls. After a while, it tapers off but swap is never touched[1]

Clearing the disk cache

For experimentation, it's very convenient to be able to drop the disk cache. For this, we can use the special file `/proc/sys/vm/drop_caches`. By writing 3 to it, we can clear most of the disk cache:

```
$ free -m
             total       used       free     shared    buffers     cached
Mem:          1504       1471         33          0         36        801
-/+ buffers/cache:        633        871
Swap:         2047          6       2041

$ echo 3 | sudo tee /proc/sys/vm/drop_caches
3

$ free -m
             total       used       free     shared    buffers     cached
```

```
Mem:          1504        763        741          0          0        134
-/+ buffers/cache:        629        875
Swap:         2047          6       2041
```

Notice how "buffers" and "cached" went down, free mem went up, and free+buffers/cache stayed the same.

Effects of disk cache on load times

Let's make two test programs, one in Python and one in Java. Python and Java both come with pretty big runtimes, which have to be loaded in order to run the application. This is a perfect scenario for disk cache to work its magic.

```
$ cat hello.py
print "Hello World! Love, Python"

$ cat Hello.java
class Hello {
    public static void main(String[] args) throws Exception {
        System.out.println("Hello World! Regards, Java");
    }
}

$ javac Hello.java

$ python hello.py
Hello World! Love, Python

$ java Hello
Hello World! Regards, Java

$
```

Our hello world apps work. Now let's drop the disk cache, and see how long it takes to run them.

```
$ echo 3 | sudo tee /proc/sys/vm/drop_caches
3

$ time python hello.py
Hello World! Love, Python

real    0m1.026s
user    0m0.020s
sys         0m0.020s

$ time java Hello
Hello World! Regards, Java

real    0m2.174s
user    0m0.100s
sys         0m0.056s

$
```

Wow. 1 second for Python, and 2 seconds for Java? That's a lot just to say hello. However, now all the file required to run them will be in the disk cache so they can be fetched straight from memory. Let's try again:

```
$ time python hello.py
Hello World! Love, Python
```

```
real    0m0.022s
user    0m0.016s
sys     0m0.008s

$ time java Hello
Hello World! Regards, Java

real    0m0.139s
user    0m0.060s
sys     0m0.028s

$
```

Yay! Python now runs in just 22 milliseconds, while java uses 139ms. That's 45 and 15 times faster! All your apps get this boost automatically!

Effects of disk cache on file reading

Let's make a big file and see how disk cache affects how fast we can read it. I'm making a 200mb file, but if you have less free ram, you can adjust it.

```
$ echo 3 | sudo tee /proc/sys/vm/drop_caches
3

$ free -m
             total       used       free     shared    buffers     cached
Mem:          1504        546        958          0          0         85
-/+ buffers/cache:        461       1043
Swap:         2047          6       2041

$ dd if=/dev/zero of=bigfile bs=1M count=200
200+0 records in
200+0 records out
209715200 bytes (210 MB) copied, 6.66191 s, 31.5 MB/s

$ ls -lh bigfile
-rw-r--r-- 1 vidar vidar 200M 2009-04-25 12:30 bigfile

$ free -m
             total       used       free     shared    buffers     cached
Mem:          1504        753        750          0          0        285
-/+ buffers/cache:        468       1036
Swap:         2047          6       2041

$
```

Since the file was just written, it will go in the disk cache. The 200MB file caused a 200MB bump in "cached". Let's read it, clear the cache, and read it again to see how fast it is:

```
$ time cat bigfile > /dev/null

real    0m0.139s
user    0m0.008s
sys     0m0.128s

$ echo 3 | sudo tee /proc/sys/vm/drop_caches
3

$ time cat bigfile > /dev/null

real    0m8.688s
```

```
user    0m0.020s
sys     0m0.336s

$
```

That's more than fifty times faster!

Conclusions

The Linux disk cache is very unobtrusive. It uses spare memory to greatly increase disk access speeds, and without taking any memory away from applications. A fully used store of ram on Linux is efficient hardware use, not a warning sign.

LinuxAteMyRam.com was presented by VidarHolen.net

These pages do simplify a little:

1. While newly allocated memory will always (though see point #2) be taken from the disk cache instead of swap, Linux can be configured to preemptively swap out other unused applications in the background to free up memory for cache. The is tunable through the 'swappiness' setting, accessible through /proc/sys/vm/swappiness.

   A server might want to swap out unused apps to speed up disk access of running ones (making the system faster), while a desktop system might want to keep apps in memory to prevent lag when the user finally uses them (making the system more responsive). This is the subject of much debate.

2. Some parts of the cache can't be dropped, not even to accomodate new applications. This includes mmap'd pages that have been mlocked by some application, dirty pages that have not yet been written to storage, and data stored in tmpfs (such as in /dev/shm). The mmap'd, mlocked pages are stuck in the page cache. Dirty pages will for the most part swiftly be written out. Data in tmpfs will be swapped out if possible.