André Carvalho

Systems programming, linux and go

Using cgroups to limit I/O

Oct 18, 2017

Developers running their apps on tsuru can choose plans based on memory and cpu usage. We were looking at adding I/O usage as one of the plan's features, so one could choose how many I/O operations per second or bytes per second a container may be able to read/write.

Being able to limit I/O is particulary important when running a diverse cloud, where applications with different workloads and needs are running together sharing multiple resources. We need to make sure that an application that starts to behave badly does not interfere with others.

Our main scheduler is a docker based and docker exposes some flags to limit a container IOPS (I/O operations per second) and BPS (bytes per second). Docker relies on a linux kernel feature, called cgroups, to be able to limit a process resource usage.

Before exposing this as a possible parameter on tsuru plans, I decided to investigate and do some experimentation using cgroups to limit a process I/O (IOPS or BPS). I came to a conclusion that currently, it is not possible to fulfill our needs and decided to delay the implementation.

In the next section, we are going to discuss cgroups, the main kernel feature used to limit resource usage.

Introduction to cgroups

Cgroups are a mechanism available in Linux to aggregate/partition a set of tasks and all their future children. Different subsystems may hook into cgroups to provide different behaviors, such as resource accounting/limiting (this particular kind of subsystem is called controller).

Cgroups (along with namespaces) are one building blocks of containers, but you don't need a container runtime to make use of them.

Managing cgroups is done by interacting with the cgroup filesystem, by creating directories and writing to certain files. There are two versions of cgroups available in newest kernels: v1 and v2. Cgroups v2 completely changes the interface between userspace and the kernel and, as of today, container runtimes only support cgroups v1, so we will focus on v1 first.

Cgroups v1

Cgroups v1 has a per-resource (memory, blkio etc) hierarchy, where each resource hierarchy contains cgroups for that resource. They all live in `/sys/fs/cgroup`:

```
/sys/fs/cgroup/
        resourceA/
                cgroup1/
                cgroup2/
        resourceB/
                cgroup3/
                cgroup4/
```

Each PID is in exactly one cgroup per resource. If a PID is not assigned to a specific cgroup for a resource, it is in the root cgroup for that particular resource.

Important: Even if a cgroup has the same name in `resourceA` and `resourceB` they are considered distinct. Some of the available resources are:

- blkio: Block IO controller - limit I/O usage
- cpuacct: CPU accounting controller - accouting for CPU usage
- cpuset: CPU set controller - allows assigning a set of CPUs and memory nodes to a set of tasks
- memory: Memory resource controller - limit memory usage
- hugeTLB: Huge TLB controller - allows limiting the usage of huge pages
- devices: Device whitelist controller - enforce open and mknode restrictions on device files
- pids: Process number controller - limit the number of tasks

In the next section, we are going to investigate the use of the `blkio` controller to limit the I/O bytes per second of a task running under a cgroup.

Limiting I/O with cgroups v1

To limit I/O we need to create a cgroup in the `blkio` controller.

```
$ mkdir -p /sys/fs/cgroup/blkio/g1
```

We are going to set our limit using `blkio.throttle.write_bps_device` file. This requires us to specify limits by device, so we must find out our device major and minor version:

```
$ cat /proc/partitions
major minor  #blocks  name

   8        0   10485760 sda
   8        1   10484719 sda1
   8       16      10240 sdb
```

Let’s limit the write bytes per second to 1048576 (1MB/s) on the sda device (8:0):

```
$ echo "8:0 1048576" > /sys/fs/cgroup/blkio/g1/blkio.throttle.write_bps_device
```

Let’s place our shell into the cgroup, by writing its PID to the `cgroup.procs` file, so every command we start will run in this cgroup:

```
$ echo $$ > /sys/fs/cgroup/blkio/g1/cgroup.procs
```

Let’s run `dd` to generate some I/O workload while watching the I/O workload using iostat:

```
$ dd if=/dev/zero of=/tmp/file1 bs=512M count=1
1+0 records in
1+0 records out
536870912 bytes (537 MB, 512 MiB) copied, 1.25273 s, 429 MB/s
```

At the same time, we get this output from iostat (redacted):

```
$ iostat 1 -d -h -y -k -p sda
Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda
                609.00         4.00    382400.00          4     382400
sda1
                609.00         4.00    382400.00          4     382400

Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda
                260.00       212.00    133696.00        212     133696
sda1
                260.00       212.00    133696.00        212     133696

Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda
                  0.99         0.00       859.41          0        868
sda1
                  0.99         0.00       859.41          0        868
...
```

We were able to write 429 MB/s; we can see from `iostat` output that all 512MB were writen in the same second. If we try the same command but opening the file with `O_DIRECT` flag (passing `oflag=direct` to `dd`):

```
$ dd if=/dev/zero of=/tmp/file1 bs=512M count=1 oflag=direct
1+0 records in
1+0 records out
536870912 bytes (537 MB, 512 MiB) copied, 539.509 s, 995 kB/s
```

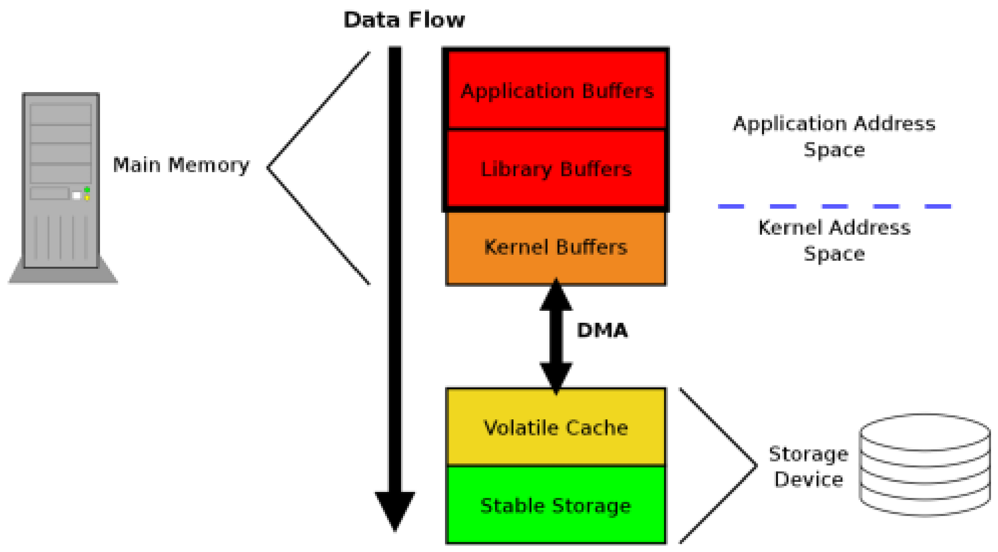
At the same time, we get this output from iostat (redacted):

```
$ iostat 1 -d -h -y -k -p sda
Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda
                  1.00         0.00      1024.00          0       1024
sda1
                  1.00         0.00      1024.00          0       1024

Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda
                  1.00         0.00      1024.00          0       1024
sda1
                  1.00         0.00      1024.00          0       1024

Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda
                  1.00         0.00      1024.00          0       1024
sda1
                  1.00         0.00      1024.00          0       1024
...
```

Et voila! Our writes were below 1.0 MB/s. Why didn't it work on the first try? Lets try to understand on the next section.

The I/O Path

So, whats the difference between opening the file with `O_DIRECT` and opening the file with no flags? The article Ensuring Data Reaches the Disk does a pretty good job explaining all the I/O flavors in Linux and their different paths on the kernel.



> That data starts out as one or more blocks of memory, or buffers, in the application itself. Those buffers can also be handed to a library, which may perform its own buffering. Regardless of whether data is buffered in application buffers or by a library, the data lives in the application's address space. The next layer that the data goes through is the kernel, which keeps its own version of a write-back cache called the page cache. Dirty pages can live in the page cache for an indeterminate amount of time, depending on overall system load and I/O patterns. When dirty data is finally evicted from the kernel's page cache, it is written to a storage device (such as a hard disk). The storage device may further buffer the data in a volatile write-back cache. If power is lost while data is in this cache, the data will be lost. Finally, at the very

> bottom of the stack is the non-volatile storage. When the data hits this layer, it is considered to be “safe.”

Basically, when we write to a file (opened without any special flags), the data travels across a bunch of buffers and caches before it is effectively writen to the disk.

Opening a file with `O_DIRECT` (available since Linux 2.4.10), means (from man pages):

> Try to minimize cache effects of the I/O to and from this file. In general this will degrade performance, but it is useful in special situations, such as when applications do their own caching. File I/O is done directly to/from user-space buffers.

So, for some reason, when we bypassed the kernel’s page cache, the cgroup was able to enforce the I/O limit specified.

This commit in Linux adds some documentation that explains exactly what is happening.

```
On traditional cgroup hierarchies, relationships between different controllers cannot be estab
lished making it impossible for writeback to operate accounting for cgroup resource restrictio
ns and all writeback IOs are attributed to the root cgroup.
```

It’s important to notice that this was added when cgroups v2 were already a reality (but still experimental). So the “traditional cgroup hierarchies” means cgroups v1. Since in cgroups v1, different resources/controllers (memory, blkio) live in different hierarchies on the filesystem, even when those cgroups have the same name, they are completely independent. So, when the memory page is finally being flushed to disk, there is no way that the memory controller can know what blkio cgroup wrote that page. That means it is going to use the root cgroup for the blkio controller.

Right below that statement, we find:

```
If both the blkio and memory controllers are used on the v2 hierarchy and the filesystem suppo
rts cgroup writeback writeback operations correctly follow the resource restrictions imposed b
y both memory and blkio controllers.
```

So, in order to limit I/O when this I/O may hit the writeback kernel cache, we need to use both memory and io controllers in the cgroups v2!

Cgroups v2

Since kernel 4.5, the cgroups v2 implementation was marked non-experimental.

In cgroups v2 there is only a single hierarchy, instead of one hierarchy for resource. Supposing the cgroups v2 file system is mounted in `/sys/fs/cgroup/:

```
/sys/fs/cgroup/
              cgroup1/
                cgroup3/
              cgroup2/
                cgroup4/
```

This hierarchy means that we may impose limits to both I/O and memory by writing to files in the `cgroup1` cgroup. Also, those limits may be inherited by `cgroup3`. Not every controller supported in cgroups v1 is available in cgroups v2. Currently, one may use: `memory`, `io`, `rdma` and `pids` controller.

Let's try the same experiment as before using cgroups v2 this time. The following example uses Ubuntu 17.04 (4.10.0-35-generic).

Disabling cgroups v1

First of all, we need to disable cgroups v1. To do that, I've added `GRUB_CMDLINE_LINUX_DEFAULT="cgroup_no_v1=all"` to `/etc/default/grub` and rebooted. That kernel config flag disables cgroup v1 for all controllers (blkio, memory, cpu and so on). This guarantees that both the io and memory controllers will be used on the v2 hierarchy (one of the requirements mentioned by the doc on Writeback mentioned earlier).

Limiting I/O

First, let's mount the cgroups v2 filesystem in `/cgroup2`:

```
$ mount -t cgroup2 nodev /cgroup2
```

Now, create a new cgroup, called `cg2` by creating a directory under the mounted fs:

```
$ mkdir /cgroup2/cg2
```

To be able to edit the I/O limits using the the I/O controller on the newly created cgroup, we need to write "+io" to the `cgroup.subtree_control` file in the parent (in this case, root) cgroup:

```
$ echo "+io" > /cgroup2/cgroup.subtree_control
```

Checking the `cgroup.controllers` file for the `cg2` cgroup, we see that the `io` controller is enabled:

```
$ cat /cgroup2/cg2/cgroup.controllers
io
```

To limit I/O to 1MB/s, as done previously, we write into the `io.max` file:

```
$ echo "8:0 wbps=1048576" > io.max
```

Let's add our bash session to the `cg2` cgroup, by writing its PID to `cgroup.procs`:

```
$ echo $$ > /cgroup2/cg2/cgroup.procs
```

Now, lets use `dd` to generate some I/O workload and watch with `iostat`:

```
dd if=/dev/zero of=/tmp/file1 bs=512M count=1
1+0 records in
1+0 records out
536870912 bytes (537 MB, 512 MiB) copied, 468.137 s, 1.1 MB/s
```

At the same time, we get this output from iostat (redacted):

```
Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda
                  1.02         0.00       693.88          0        680
sda1
                  1.02         0.00       693.88          0        680

Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda
                  2.00         0.00       732.00          0        732
sda1
                  2.00         0.00       732.00          0        732

Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda
                  0.99         0.00       669.31          0        676
sda1
                  0.99         0.00       669.31          0        676

Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda
                  1.00         0.00       672.00          0        672
sda1
                  1.00         0.00       672.00          0        672
```

```
Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda
                  1.00         0.00      1024.00          0       1024
sda1
                  1.00         0.00      1024.00          0       1024
...
```

So, even relying on the writeback kernel cache we are able to limit the I/O on the disk.

Wrapping Up

We've seen how limiting I/O is hard using cgroups. After some experimentation and research we found out that using cgroups v2 is the only way to properly limit I/O (if you can't change your application to do direct or sync I/O).

As a result of this experiment, we decided to not limit I/O using docker at the moment since it does not support cgroups v2 (yet).