Subject: [PATCH 2/3] i/o bandwidth controller infrastructure
Posted by Andrea Righi on Fri, 20 Jun 2008 10:05:34 GMT
View Forum Message <> Reply to Message

This is the core io-throttle kernel infrastructure. It creates the basic
interfaces to cgroups and implements the I/O measurement and throttling
functions.

Signed-off-by: Andrea Righi <righi.andrea@gmail.com>
---
 block/Makefile | 2 +
 block/blk-io-throttle.c | 393 ++
 include/linux/blk-io-throttle.h | 12 ++
 include/linux/cgroup_subsys.h | 6 +
 init/Kconfig | 10 +
 5 files changed, 423 insertions(+), 0 deletions(-)
 create mode 100644 block/blk-io-throttle.c
 create mode 100644 include/linux/blk-io-throttle.h

diff --git a/block/Makefile b/block/Makefile
index 5a43c7d..8dec69b 100644
--- a/block/Makefile
+++ b/block/Makefile
@@ -14,3 +14,5 @@ obj-$(CONFIG_IOSCHED_CFQ) += cfq-iosched.o

 obj-$(CONFIG_BLK_DEV_IO_TRACE) += blktrace.o
 obj-$(CONFIG_BLOCK_COMPAT) += compat_ioctl.o
+
+obj-$(CONFIG_CGROUP_IO_THROTTLE) += blk-io-throttle.o
diff --git a/block/blk-io-throttle.c b/block/blk-io-throttle.c
new file mode 100644
index 0000000..4ec02bb
--- /dev/null
+++ b/block/blk-io-throttle.c
@@ -0,0 +1,393 @@
+/*
+ * blk-io-throttle.c
+ *
+ * This program is free software; you can redistribute it and/or
+ * modify it under the terms of the GNU General Public
+ * License as published by the Free Software Foundation; either
+ * version 2 of the License, or (at your option) any later version.
+ *
+ * This program is distributed in the hope that it will be useful,
+ * but WITHOUT ANY WARRANTY; without even the implied warranty of
+ * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
+ * General Public License for more details.
+ *

```
+ * You should have received a copy of the GNU General Public
+ * License along with this program; if not, write to the
+ * Free Software Foundation, Inc., 59 Temple Place - Suite 330,
+ * Boston, MA 021110-1307, USA.
+ *
+ * Copyright (C) 2008 Andrea Righi <righi.andrea@gmail.com>
+ */
+
+#include <linux/init.h>
+#include <linux/module.h>
+#include <linux/cgroup.h>
+#include <linux/slab.h>
+#include <linux/gfp.h>
+#include <linux/err.h>
+#include <linux/sched.h>
+#include <linux/fs.h>
+#include <linux/jiffies.h>
+#include <linux/hardirq.h>
+#include <linux/list.h>
+#include <linux/spinlock.h>
+#include <linux/uaccess.h>
+#include <linux/vmalloc.h>
+#include <linux/blk-io-throttle.h>
+
+#define ONE_SEC 1000000L /* # of microseconds in a second */
+#define KBS(x) ((x) * ONE_SEC >> 10)
+
+struct iothrottle_node {
+ struct list_head node;
+ dev_t dev;
+ unsigned long iorate;
+ unsigned long timestamp;
+ atomic_long_t stat;
+};
+
+struct iothrottle {
+ struct cgroup_subsys_state css;
+ /* protects the list below, not the single elements */
+ spinlock_t lock;
+ struct list_head list;
+};
+
+static inline struct iothrottle *cgroup_to_iothrottle(struct cgroup *cont)
+{
+ return container_of(cgroup_subsys_state(cont, iothrottle_subsys_id),
+       struct iothrottle, css);
+}
+
```

```
+static inline struct iothrottle *task_to_iothrottle(struct task_struct *task)
+{
+ return container_of(task_subsys_state(task, iothrottle_subsys_id),
+       struct iothrottle, css);
+}
+
+static inline struct iothrottle_node *iothrottle_search_node(
+     const struct iothrottle *iot,
+     dev_t dev)
+{
+ struct iothrottle_node *n;
+
+ list_for_each_entry_rcu(n, &iot->list, node)
+  if (n->dev == dev)
+   return n;
+ return NULL;
+}
+
+static inline void iothrottle_insert_node(struct iothrottle *iot,
+      struct iothrottle_node *n)
+{
+ list_add_rcu(&n->node, &iot->list);
+}
+
+static inline struct iothrottle_node *iothrottle_replace_node(
+     struct iothrottle *iot,
+     struct iothrottle_node *old,
+     struct iothrottle_node *new)
+{
+ list_replace_rcu(&old->node, &new->node);
+ return old;
+}
+
+static inline struct iothrottle_node *iothrottle_delete_node(
+     struct iothrottle *iot,
+     dev_t dev)
+{
+ struct iothrottle_node *n;
+
+ list_for_each_entry(n, &iot->list, node)
+  if (n->dev == dev) {
+   list_del_rcu(&n->node);
+   return n;
+  }
+ return NULL;
+}
+
+/*
```

```
+ * Note: called from kernel/cgroup.c with cgroup_lock() held.
+ */
+static struct cgroup_subsys_state *iothrottle_create(
+   struct cgroup_subsys *ss, struct cgroup *cont)
+{
+ struct iothrottle *iot;
+
+ iot = kmalloc(sizeof(*iot), GFP_KERNEL);
+ if (unlikely(!iot))
+  return ERR_PTR(-ENOMEM);
+
+ INIT_LIST_HEAD(&iot->list);
+ spin_lock_init(&iot->lock);
+
+ return &iot->css;
+}
+
+/*
+ * Note: called from kernel/cgroup.c with cgroup_lock() held.
+ */
+static void iothrottle_destroy(struct cgroup_subsys *ss, struct cgroup *cont)
+{
+ struct iothrottle_node *n, *p;
+ struct iothrottle *iot = cgroup_to_iothrottle(cont);
+
+ /*
+  * don't worry about locking here, at this point there must be not any
+  * reference to the list.
+  */
+ list_for_each_entry_safe(n, p, &iot->list, node)
+  kfree(n);
+ kfree(iot);
+}
+
+static ssize_t iothrottle_read(struct cgroup *cont,
+         struct cftype *cft,
+         struct file *file,
+         char __user *userbuf,
+         size_t nbytes,
+         loff_t *ppos)
+{
+ struct iothrottle *iot;
+ char *buffer;
+ int s = 0;
+ struct iothrottle_node *n;
+ ssize_t ret;
+
+ buffer = kmalloc(nbytes + 1, GFP_KERNEL);
```

```
+ if (!buffer)
+  return -ENOMEM;
+
+ cgroup_lock();
+ if (cgroup_is_removed(cont)) {
+  ret = -ENODEV;
+  goto out;
+ }
+
+ iot = cgroup_to_iothrottle(cont);
+ rcu_read_lock();
+ list_for_each_entry_rcu(n, &iot->list, node) {
+  unsigned long delta, rate;
+
+  BUG_ON(!n->dev);
+  delta = jiffies_to_usecs((long)jiffies - (long)n->timestamp);
+  rate = delta ? KBS(atomic_long_read(&n->stat) / delta) : 0;
+  s += scnprintf(buffer + s, nbytes - s,
+         "=== device (%u,%u) ===\n"
+         "  bandwidth limit: %lu KiB/sec\n"
+         "current i/o usage: %lu KiB/sec\n",
+         MAJOR(n->dev), MINOR(n->dev),
+         n->iorate, rate);
+ }
+ rcu_read_unlock();
+ ret = simple_read_from_buffer(userbuf, nbytes, ppos, buffer, s);
+out:
+ cgroup_unlock();
+ kfree(buffer);
+ return ret;
+}
+
+static inline dev_t devname2dev_t(const char *buf)
+{
+ struct block_device *bdev;
+ dev_t ret;
+
+ bdev = lookup_bdev(buf);
+ if (IS_ERR(bdev))
+  return 0;
+
+ BUG_ON(!bdev->bd_inode);
+ ret = bdev->bd_inode->i_rdev;
+ bdput(bdev);
+
+ return ret;
+}
+
```

```
+static inline int iothrottle_parse_args(char *buf, size_t nbytes,
+     dev_t *dev, unsigned long *val)
+{
+ char *p;
+
+ p = memchr(buf, ':', nbytes);
+ if (!p)
+  return -EINVAL;
+ *p++ = '\0';
+
+ /* i/o bandiwth is expressed in KiB/s */
+ *val = ALIGN(memparse(p, &p), 1024) >> 10;
+ if (*p)
+  return -EINVAL;
+
+ *dev = devname2dev_t(buf);
+ if (!*dev)
+  return -ENOTBLK;
+
+ return 0;
+}
+
+static ssize_t iothrottle_write(struct cgroup *cont,
+    struct cftype *cft,
+    struct file *file,
+    const char __user *userbuf,
+    size_t nbytes, loff_t *ppos)
+{
+ struct iothrottle *iot;
+ struct iothrottle_node *n, *tmpn = NULL;
+ char *buffer, *tmpp;
+ dev_t dev;
+ unsigned long val;
+ int ret;
+
+ if (!nbytes)
+  return -EINVAL;
+
+ /* Upper limit on largest io-throttle rule string user might write. */
+ if (nbytes > 1024)
+  return -E2BIG;
+
+ buffer = kmalloc(nbytes + 1, GFP_KERNEL);
+ if (!buffer)
+  return -ENOMEM;
+
+ if (copy_from_user(buffer, userbuf, nbytes)) {
+  ret = -EFAULT;
```

```
+  goto out1;
+ }
+
+ buffer[nbytes] = '\0';
+ tmpp = strstrip(buffer);
+
+ ret = iothrottle_parse_args(tmpp, nbytes, &dev, &val);
+ if (ret)
+  goto out1;
+
+ if (val) {
+  tmpn = kmalloc(sizeof(*tmpn), GFP_KERNEL);
+  if (!tmpn) {
+   ret = -ENOMEM;
+   goto out1;
+  }
+  atomic_long_set(&tmpn->stat, 0);
+  tmpn->timestamp = jiffies;
+  tmpn->iorate = val;
+  tmpn->dev = dev;
+ }
+
+ cgroup_lock();
+ if (cgroup_is_removed(cont)) {
+  ret = -ENODEV;
+  goto out2;
+ }
+
+ iot = cgroup_to_iothrottle(cont);
+ spin_lock(&iot->lock);
+ if (!val) {
+  /* Delete a block device limiting rule */
+  n = iothrottle_delete_node(iot, dev);
+  goto out3;
+ }
+ n = iothrottle_search_node(iot, dev);
+ if (n) {
+  /* Update a block device limiting rule */
+  iothrottle_replace_node(iot, n, tmpn);
+  goto out3;
+ }
+ /* Add a new block device limiting rule */
+ iothrottle_insert_node(iot, tmpn);
+out3:
+ ret = nbytes;
+ spin_unlock(&iot->lock);
+ if (n) {
+  synchronize_rcu();
```

```
+  kfree(n);
+ }
+out2:
+ cgroup_unlock();
+out1:
+ kfree(buffer);
+ return ret;
+}
+
+static struct cftype files[] = {
+ {
+  .name = "bandwidth",
+  .read = iothrottle_read,
+  .write = iothrottle_write,
+ },
+};
+
+static int iothrottle_populate(struct cgroup_subsys *ss, struct cgroup *cont)
+{
+ return cgroup_add_files(cont, ss, files, ARRAY_SIZE(files));
+}
+
+struct cgroup_subsys iothrottle_subsys = {
+ .name = "blockio",
+ .create = iothrottle_create,
+ .destroy = iothrottle_destroy,
+ .populate = iothrottle_populate,
+ .subsys_id = iothrottle_subsys_id,
+};
+
+static inline int __cant_sleep(void)
+{
+ return in_atomic() || in_interrupt() || irqs_disabled();
+}
+
+void cgroup_io_throttle(struct block_device *bdev, size_t bytes)
+{
+ struct iothrottle *iot;
+ struct iothrottle_node *n;
+ unsigned long delta, t;
+ long sleep;
+
+ if (unlikely(!bdev || !bytes))
+  return;
+
+ iot = task_to_iothrottle(current);
+ if (unlikely(!iot))
+  return;
```

```
+
+ BUG_ON(!bdev->bd_inode);
+
+ rcu_read_lock();
+ n = iothrottle_search_node(iot, bdev->bd_inode->i_rdev);
+ if (!n || !n->iorate)
+  goto out;
+
+ /* Account the i/o activity */
+ atomic_long_add(bytes, &n->stat);
+
+ /* Evaluate if we need to throttle the current process */
+ delta = (long)jiffies - (long)n->timestamp;
+ if (!delta)
+  goto out;
+
+ t = usecs_to_jiffies(KBS(atomic_long_read(&n->stat) / n->iorate));
+ if (!t)
+  goto out;
+
+ sleep = t - delta;
+ if (unlikely(sleep > 0)) {
+  rcu_read_unlock();
+  if (__cant_sleep())
+   return;
+  pr_debug("io-throttle: task %p (%s) must sleep %lu jiffies\n",
+    current, current->comm, delta);
+  schedule_timeout_killable(sleep);
+  return;
+ }
+ /* Reset i/o statistics */
+ atomic_long_set(&n->stat, 0);
+ /*
+  * NOTE: be sure i/o statistics have been resetted before updating the
+  * timestamp, otherwise a very small time delta may possibly be read by
+  * another CPU w.r.t. accounted i/o statistics, generating unnecessary
+  * long sleeps.
+  */
+ smp_wmb();
+ n->timestamp = jiffies;
+out:
+ rcu_read_unlock();
+}
+EXPORT_SYMBOL(cgroup_io_throttle);
diff --git a/include/linux/blk-io-throttle.h b/include/linux/blk-io-throttle.h
new file mode 100644
index 0000000..3e08738
--- /dev/null
```

```
+++ b/include/linux/blk-io-throttle.h
@@ -0,0 +1,12 @@
+#ifndef BLK_IO_THROTTLE_H
+#define BLK_IO_THROTTLE_H
+
+#ifdef CONFIG_CGROUP_IO_THROTTLE
+extern void cgroup_io_throttle(struct block_device *bdev, size_t bytes);
+#else
+static inline void cgroup_io_throttle(struct block_device *bdev, size_t bytes)
+{
+}
+#endif /* CONFIG_CGROUP_IO_THROTTLE */
+
+#endif /* BLK_IO_THROTTLE_H */
diff --git a/include/linux/cgroup_subsys.h b/include/linux/cgroup_subsys.h
index e287745..0caf3c2 100644
--- a/include/linux/cgroup_subsys.h
+++ b/include/linux/cgroup_subsys.h
@@ -48,3 +48,9 @@ SUBSYS(devices)
 #endif

 /* */
+
+#ifdef CONFIG_CGROUP_IO_THROTTLE
+SUBSYS(iothrottle)
+#endif
+
+/* */
diff --git a/init/Kconfig b/init/Kconfig
index 6199d11..3117d99 100644
--- a/init/Kconfig
+++ b/init/Kconfig
@@ -306,6 +306,16 @@ config CGROUP_DEVICE
    Provides a cgroup implementing whitelists for devices which
    a process in the cgroup can mknod or open.

+config CGROUP_IO_THROTTLE
+ bool "Enable cgroup I/O throttling (EXPERIMENTAL)"
+ depends on CGROUPS && EXPERIMENTAL
+ help
+   This allows to limit the maximum I/O bandwidth for specific
+   cgroup(s).
+   See Documentation/controllers/io-throttle.txt for more information.
+
+   If unsure, say N.
+
 config CPUSETS
  bool "Cpuset support"
```

depends on SMP && CGROUPS
--
1.5.4.3

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [PATCH 2/3] i/o bandwidth controller infrastructure
Posted by Andrea Righi on Sun, 22 Jun 2008 13:11:13 GMT
View Forum Message <> Reply to Message

Carl Henrik Lunde wrote:
> Did you consider using token bucket instead of this (leaky bucket?)?
>
> I've attached a patch which implements token bucket. Although not as
> precise as the leaky bucket the performance is better at high bandwidth
> streaming loads.
>
> The leaky bucket stops at around 53 MB/s while token bucket works for
> up to 64 MB/s. The baseline (no cgroups) is 66 MB/s.
>
> benchmark:
> two streaming readers (fio) with block size 128k, bucket size 4 MB
> 90% of the bandwidth was allocated to one process, the other gets 10%
>
> bw-limit: actual bw algorithm bw1 bw2
> 5 MiB/s: 5.0 MiB/s leaky_bucket 0.5 4.5
> 5 MiB/s: 5.2 MiB/s token_bucket 0.6 4.6
> 10 MiB/s: 10.0 MiB/s leaky_bucket 1.0 9.0
> 10 MiB/s: 10.3 MiB/s token_bucket 1.0 9.2
> 15 MiB/s: 15.0 MiB/s leaky_bucket 1.5 13.5
> 15 MiB/s: 15.4 MiB/s token_bucket 1.5 13.8
> 20 MiB/s: 19.9 MiB/s leaky_bucket 2.0 17.9
> 20 MiB/s: 20.5 MiB/s token_bucket 2.1 18.4
> 25 MiB/s: 24.4 MiB/s leaky_bucket 2.5 21.9
> 25 MiB/s: 25.6 MiB/s token_bucket 2.6 23.0
> 30 MiB/s: 29.2 MiB/s leaky_bucket 3.0 26.2
> 30 MiB/s: 30.7 MiB/s token_bucket 3.1 27.7
> 35 MiB/s: 34.3 MiB/s leaky_bucket 3.4 30.9
> 35 MiB/s: 35.9 MiB/s token_bucket 3.6 32.3
> 40 MiB/s: 39.7 MiB/s leaky_bucket 3.9 35.8
> 40 MiB/s: 41.0 MiB/s token_bucket 4.1 36.9
> 45 MiB/s: 44.0 MiB/s leaky_bucket 4.3 39.7
> 45 MiB/s: 46.1 MiB/s token_bucket 4.6 41.5
> 50 MiB/s: 47.9 MiB/s leaky_bucket 4.7 43.2

> 50 MiB/s: 51.0 MiB/s token_bucket 5.1 45.9
> 55 MiB/s: 50.5 MiB/s leaky_bucket 5.0 45.5
> 55 MiB/s: 56.2 MiB/s token_bucket 5.6 50.5
> 60 MiB/s: 52.9 MiB/s leaky_bucket 5.2 47.7
> 60 MiB/s: 61.0 MiB/s token_bucket 6.1 54.9
> 65 MiB/s: 53.0 MiB/s leaky_bucket 5.4 47.6
> 65 MiB/s: 63.7 MiB/s token_bucket 6.6 57.1
> 70 MiB/s: 53.8 MiB/s leaky_bucket 5.5 48.4
> 70 MiB/s: 64.1 MiB/s token_bucket 7.1 57.0

Carl,

based on your token bucket solution I've implemented a run-time leaky
bucket / token bucket switcher:

leaky bucket
echo 0 > /cgroups/foo/blockio.throttling_strategy
token bucket
echo 1 > /cgroups/foo/blockio.throttling_strategy

The -rc of the new io-throttle patch 2/3 is below, 1/3 and 3/3 are the
same as patchset version 3, even if documentation must be updated. It
would be great if you could review the patch, in particular the
token_bucket() implementation and repeat your tests.

The all-in-one patch is available here:
http://download.systemimager.org/~arighi/linux/patches/io-throttle/cgroup-io-throttle-v4-rc1.patch

I also did some quick tests similar to yours, the benchmark I've used is
available here as well:
http://download.systemimager.org/~arighi/linux/patches/io-throttle/benchmark/iobw.c

= Results =
I/O scheduler: cfq
filesystem: ext3
Command: ionice -c 1 -n 0 iobw -direct 2 4m 32m
Bucket size: 4MiB

testing 2 parallel streams, chunk_size 4096KiB, data_size 32768KiB

=== no throttling ===
testing 2 parallel streams, chunk_size 4096KiB, data_size 32768KiB
[task   2] time: 2.929, bw: 10742 KiB/s (WRITE)
[task   2] time: 2.878, bw: 10742 KiB/s (READ)
[task   1] time: 2.377, bw: 13671 KiB/s (WRITE)
[task   1] time: 3.979, bw: 7812 KiB/s (READ)
[parent 0] time: 6.397, bw: 19531 KiB/s (TOTAL)

=== bandwidth limit: 4MiB/s (leaky bucket) ===
[task 2] time: 15.880, bw: 1953 KiB/s (WRITE)
[task 2] time: 14.278, bw: 1953 KiB/s (READ)
[task 1] time: 14.711, bw: 1953 KiB/s (WRITE)
[task 1] time: 16.563, bw: 1953 KiB/s (READ)
[parent 0] time: 31.316, bw: 3906 KiB/s (TOTAL)

=== bandwidth limit: 4MiB/s (token bucket) ===
[task 2] time: 11.864, bw: 1953 KiB/s (WRITE)
[task 2] time: 15.958, bw: 1953 KiB/s (READ)
[task 1] time: 19.233, bw: 976 KiB/s (WRITE)
[task 1] time: 12.643, bw: 1953 KiB/s (READ)
[parent 0] time: 31.917, bw: 3906 KiB/s (TOTAL)

=== bandwidth limit: 8MiB/s (leaky bucket) ===
[task 2] time: 7.198, bw: 3906 KiB/s (WRITE)
[task 2] time: 8.012, bw: 3906 KiB/s (READ)
[task 1] time: 7.891, bw: 3906 KiB/s (WRITE)
[task 1] time: 7.846, bw: 3906 KiB/s (READ)
[parent 0] time: 15.780, bw: 7812 KiB/s (TOTAL)

=== bandwidth limit: 8MiB/s (token bucket) ===
[task 1] time: 6.996, bw: 3906 KiB/s (WRITE)
[task 1] time: 6.529, bw: 4882 KiB/s (READ)
[task 2] time: 10.341, bw: 2929 KiB/s (WRITE)
[task 2] time: 5.681, bw: 4882 KiB/s (READ)
[parent 0] time: 16.079, bw: 7812 KiB/s (TOTAL)

=== bandwidth limit: 12MiB/s (leaky bucket) ===
[task 2] time: 4.992, bw: 5859 KiB/s (WRITE)
[task 2] time: 5.077, bw: 5859 KiB/s (READ)
[task 1] time: 5.500, bw: 5859 KiB/s (WRITE)
[task 1] time: 5.061, bw: 5859 KiB/s (READ)
[parent 0] time: 10.603, bw: 11718 KiB/s (TOTAL)

=== bandwidth limit: 12MiB/s (token bucket) ===
[task 1] time: 5.057, bw: 5859 KiB/s (WRITE)
[task 1] time: 4.329, bw: 6835 KiB/s (READ)
[task 2] time: 5.771, bw: 4882 KiB/s (WRITE)
[task 2] time: 4.961, bw: 5859 KiB/s (READ)
[parent 0] time: 10.786, bw: 11718 KiB/s (TOTAL)

=== bandwidth limit: 16MiB/s (leaky bucket) ===
[task 1] time: 3.737, bw: 7812 KiB/s (WRITE)
[task 1] time: 3.988, bw: 7812 KiB/s (READ)
[task 2] time: 4.043, bw: 7812 KiB/s (WRITE)
[task 2] time: 3.954, bw: 7812 KiB/s (READ)
[parent 0] time: 8.040, bw: 15625 KiB/s (TOTAL)

```
=== bandwidth limit: 16MiB/s (token bucket) ===
[task   1] time:  3.224, bw:    9765 KiB/s (WRITE)
[task   1] time:  3.550, bw:    8789 KiB/s (READ )
[task   2] time:  5.085, bw:    5859 KiB/s (WRITE)
[task   2] time:  3.033, bw:   10742 KiB/s (READ )
[parent 0] time:  8.160, bw:   15625 KiB/s (TOTAL)

=== bandwidth limit: 20MiB/s (leaky bucket) ===
[task   1] time:  3.265, bw:    9765 KiB/s (WRITE)
[task   1] time:  3.339, bw:    9765 KiB/s (READ )
[task   2] time:  3.001, bw:   10742 KiB/s (WRITE)
[task   2] time:  3.840, bw:    7812 KiB/s (READ )
[parent 0] time:  6.884, bw:   18554 KiB/s (TOTAL)

=== bandwidth limit: 20MiB/s (token bucket) ===
[task   1] time:  2.897, bw:   10742 KiB/s (WRITE)
[task   1] time:  3.071, bw:    9765 KiB/s (READ )
[task   2] time:  3.697, bw:    8789 KiB/s (WRITE)
[task   2] time:  2.925, bw:   10742 KiB/s (READ )
[parent 0] time:  6.657, bw:   19531 KiB/s (TOTAL)

=== bandwidth limit: 24MiB/s (leaky bucket) ===
[task   1] time:  2.283, bw:   13671 KiB/s (WRITE)
[task   1] time:  3.626, bw:    8789 KiB/s (READ )
[task   2] time:  3.892, bw:    7812 KiB/s (WRITE)
[task   2] time:  2.774, bw:   11718 KiB/s (READ )
[parent 0] time:  6.724, bw:   18554 KiB/s (TOTAL)

=== bandwidth limit: 24MiB/s (token bucket) ===
[task   2] time:  3.215, bw:    9765 KiB/s (WRITE)
[task   2] time:  2.767, bw:   11718 KiB/s (READ )
[task   1] time:  2.615, bw:   11718 KiB/s (WRITE)
[task   1] time:  3.958, bw:    7812 KiB/s (READ )
[parent 0] time:  6.610, bw:   19531 KiB/s (TOTAL)
```

In conclusion, results seem to confirm that leaky bucket is more precise (more smoothed) than token bucket; token bucket, instead, is better in terms of efficiency when approaching to the disk's I/O physical limit, as the theory claims.

It would be also interesting to test how token bucket performance changes using different bucket size values. I'll do more accurate tests ASAP.

Signed-off-by: Andrea Righi <righi.andrea@gmail.com>
---
 block/Makefile | 2 +

```
 block/blk-io-throttle.c        |  490 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 include/linux/blk-io-throttle.h |   12 +
 include/linux/cgroup_subsys.h   |    6 +
 init/Kconfig                    |   10 +
 5 files changed, 520 insertions(+), 0 deletions(-)

diff --git a/block/Makefile b/block/Makefile
index 5a43c7d..8dec69b 100644
--- a/block/Makefile
+++ b/block/Makefile
@@ -14,3 +14,5 @@ obj-$(CONFIG_IOSCHED_CFQ) += cfq-iosched.o

 obj-$(CONFIG_BLK_DEV_IO_TRACE) += blktrace.o
 obj-$(CONFIG_BLOCK_COMPAT) += compat_ioctl.o
+
+obj-$(CONFIG_CGROUP_IO_THROTTLE) += blk-io-throttle.o
diff --git a/block/blk-io-throttle.c b/block/blk-io-throttle.c
new file mode 100644
index 0000000..c6af273
--- /dev/null
+++ b/block/blk-io-throttle.c
@@ -0,0 +1,490 @@
+/*
+ * blk-io-throttle.c
+ *
+ * This program is free software; you can redistribute it and/or
+ * modify it under the terms of the GNU General Public
+ * License as published by the Free Software Foundation; either
+ * version 2 of the License, or (at your option) any later version.
+ *
+ * This program is distributed in the hope that it will be useful,
+ * but WITHOUT ANY WARRANTY; without even the implied warranty of
+ * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
+ * General Public License for more details.
+ *
+ * You should have received a copy of the GNU General Public
+ * License along with this program; if not, write to the
+ * Free Software Foundation, Inc., 59 Temple Place - Suite 330,
+ * Boston, MA 021110-1307, USA.
+ *
+ * Copyright (C) 2008 Andrea Righi <righi.andrea@gmail.com>
+ */
+
+#include <linux/init.h>
+#include <linux/module.h>
+#include <linux/cgroup.h>
+#include <linux/slab.h>
+#include <linux/gfp.h>
```

```
+#include <linux/err.h>
+#include <linux/sched.h>
+#include <linux/fs.h>
+#include <linux/jiffies.h>
+#include <linux/hardirq.h>
+#include <linux/list.h>
+#include <linux/spinlock.h>
+#include <linux/uaccess.h>
+#include <linux/vmalloc.h>
+#include <linux/blk-io-throttle.h>
+
+#define ONE_SEC 1000000L /* # of microseconds in a second */
+#define KBS(x) ((x) * ONE_SEC >> 10)
+
+struct iothrottle_node {
+ struct list_head node;
+ dev_t dev;
+ unsigned long iorate;
+ unsigned long timestamp;
+ atomic_long_t stat;
+ long bucket_size;
+ atomic_long_t token;
+};
+
+struct iothrottle {
+ struct cgroup_subsys_state css;
+ /* protects the list below, not the single elements */
+ spinlock_t lock;
+ struct list_head list;
+ int strategy;
+};
+
+static inline struct iothrottle *cgroup_to_iothrottle(struct cgroup *cont)
+{
+ return container_of(cgroup_subsys_state(cont, iothrottle_subsys_id),
+       struct iothrottle, css);
+}
+
+static inline struct iothrottle *task_to_iothrottle(struct task_struct *task)
+{
+ return container_of(task_subsys_state(task, iothrottle_subsys_id),
+       struct iothrottle, css);
+}
+
+static inline struct iothrottle_node *iothrottle_search_node(
+     const struct iothrottle *iot,
+     dev_t dev)
+{
```

```
+ struct iothrottle_node *n;
+
+ list_for_each_entry_rcu(n, &iot->list, node)
+  if (n->dev == dev)
+   return n;
+ return NULL;
+}
+
+static inline void iothrottle_insert_node(struct iothrottle *iot,
+      struct iothrottle_node *n)
+{
+ list_add_rcu(&n->node, &iot->list);
+}
+
+static inline struct iothrottle_node *iothrottle_replace_node(
+     struct iothrottle *iot,
+     struct iothrottle_node *old,
+     struct iothrottle_node *new)
+{
+ list_replace_rcu(&old->node, &new->node);
+ return old;
+}
+
+static inline struct iothrottle_node *iothrottle_delete_node(
+     struct iothrottle *iot,
+     dev_t dev)
+{
+ struct iothrottle_node *n;
+
+ list_for_each_entry(n, &iot->list, node)
+  if (n->dev == dev) {
+   list_del_rcu(&n->node);
+   return n;
+  }
+ return NULL;
+}
+
+/*
+ * Note: called from kernel/cgroup.c with cgroup_lock() held.
+ */
+static struct cgroup_subsys_state *iothrottle_create(
+   struct cgroup_subsys *ss, struct cgroup *cont)
+{
+ struct iothrottle *iot;
+
+ iot = kmalloc(sizeof(*iot), GFP_KERNEL);
+ if (unlikely(!iot))
+  return ERR_PTR(-ENOMEM);
```

```
+
+ INIT_LIST_HEAD(&iot->list);
+ spin_lock_init(&iot->lock);
+ iot->strategy = 0;
+
+ return &iot->css;
+}
+
+/*
+ * Note: called from kernel/cgroup.c with cgroup_lock() held.
+ */
+static void iothrottle_destroy(struct cgroup_subsys *ss, struct cgroup *cont)
+{
+ struct iothrottle_node *n, *p;
+ struct iothrottle *iot = cgroup_to_iothrottle(cont);
+
+ /*
+  * don't worry about locking here, at this point there must be not any
+  * reference to the list.
+  */
+ list_for_each_entry_safe(n, p, &iot->list, node)
+  kfree(n);
+ kfree(iot);
+}
+
+static ssize_t iothrottle_read(struct cgroup *cont,
+          struct cftype *cft,
+          struct file *file,
+          char __user *userbuf,
+          size_t nbytes,
+          loff_t *ppos)
+{
+ struct iothrottle *iot;
+ char *buffer;
+ int s = 0;
+ struct iothrottle_node *n;
+ ssize_t ret;
+
+ buffer = kmalloc(nbytes + 1, GFP_KERNEL);
+ if (!buffer)
+  return -ENOMEM;
+
+ cgroup_lock();
+ if (cgroup_is_removed(cont)) {
+  ret = -ENODEV;
+  goto out;
+ }
+
```

```
+ iot = cgroup_to_iothrottle(cont);
+ rcu_read_lock();
+ list_for_each_entry_rcu(n, &iot->list, node) {
+  unsigned long delta, rate;
+
+  BUG_ON(!n->dev);
+  delta = jiffies_to_usecs((long)jiffies - (long)n->timestamp);
+  rate = delta ? KBS(atomic_long_read(&n->stat) / delta) : 0;
+  s += scnprintf(buffer + s, nbytes - s,
+         "device: %u,%u\n"
+         "bandwidth: %lu KiB/sec\n"
+         "usage: %lu KiB/sec\n"
+         "bucket size: %lu KiB\n"
+         "bucket fill: %li KiB\n",
+         MAJOR(n->dev), MINOR(n->dev),
+         n->iorate, rate,
+         n->bucket_size,
+         atomic_long_read(&n->token) >> 10);
+ }
+ rcu_read_unlock();
+ ret = simple_read_from_buffer(userbuf, nbytes, ppos, buffer, s);
+out:
+ cgroup_unlock();
+ kfree(buffer);
+ return ret;
+}
+
+static inline dev_t devname2dev_t(const char *buf)
+{
+ struct block_device *bdev;
+ dev_t ret;
+
+ bdev = lookup_bdev(buf);
+ if (IS_ERR(bdev))
+  return 0;
+
+ BUG_ON(!bdev->bd_inode);
+ ret = bdev->bd_inode->i_rdev;
+ bdput(bdev);
+
+ return ret;
+}
+
+static inline int iothrottle_parse_args(char *buf, size_t nbytes,
+     dev_t *dev, unsigned long *iorate,
+     unsigned long *bucket_size)
+{
+ char *ioratep, *bucket_sizep;
```

```
+
+ ioratep = memchr(buf, ':', nbytes);
+ if (!ioratep)
+  return -EINVAL;
+ *ioratep++ = '\0';
+
+ bucket_sizep = memchr(ioratep, ':', nbytes + ioratep - buf);
+ if (!bucket_sizep)
+  return -EINVAL;
+ *bucket_sizep++ = '\0';
+
+ /* i/o bandiwth is expressed in KiB/s */
+ *iorate = ALIGN(memparse(ioratep, &ioratep), 1024) >> 10;
+ if (*ioratep)
+  return -EINVAL;
+ *bucket_size = ALIGN(memparse(bucket_sizep, &bucket_sizep), 1024) >> 10;
+ if (*bucket_sizep)
+  return -EINVAL;
+
+ *dev = devname2dev_t(buf);
+ if (!*dev)
+  return -ENOTBLK;
+
+ return 0;
+}
+
+static ssize_t iothrottle_write(struct cgroup *cont,
+    struct cftype *cft,
+    struct file *file,
+    const char __user *userbuf,
+    size_t nbytes, loff_t *ppos)
+{
+ struct iothrottle *iot;
+ struct iothrottle_node *n, *tmpn = NULL;
+ char *buffer, *tmpp;
+ dev_t dev;
+ unsigned long iorate, bucket_size;
+ int ret;
+
+ if (!nbytes)
+  return -EINVAL;
+
+ /* Upper limit on largest io-throttle rule string user might write. */
+ if (nbytes > 1024)
+  return -E2BIG;
+
+ buffer = kmalloc(nbytes + 1, GFP_KERNEL);
+ if (!buffer)
```

```
+  return -ENOMEM;
+
+ if (copy_from_user(buffer, userbuf, nbytes)) {
+  ret = -EFAULT;
+  goto out1;
+ }
+
+ buffer[nbytes] = '\0';
+ tmpp = strstrip(buffer);
+
+ ret = iothrottle_parse_args(tmpp, nbytes, &dev, &iorate, &bucket_size);
+ if (ret)
+  goto out1;
+
+ if (iorate) {
+  tmpn = kmalloc(sizeof(*tmpn), GFP_KERNEL);
+  if (!tmpn) {
+   ret = -ENOMEM;
+   goto out1;
+  }
+  atomic_long_set(&tmpn->stat, 0);
+  tmpn->timestamp = jiffies;
+  tmpn->iorate = iorate;
+  tmpn->bucket_size = bucket_size;
+  atomic_long_set(&tmpn->token, 0);
+  tmpn->dev = dev;
+ }
+
+ cgroup_lock();
+ if (cgroup_is_removed(cont)) {
+  ret = -ENODEV;
+  goto out2;
+ }
+
+ iot = cgroup_to_iothrottle(cont);
+ spin_lock(&iot->lock);
+ if (!iorate) {
+  /* Delete a block device limiting rule */
+  n = iothrottle_delete_node(iot, dev);
+  goto out3;
+ }
+ n = iothrottle_search_node(iot, dev);
+ if (n) {
+  /* Update a block device limiting rule */
+  iothrottle_replace_node(iot, n, tmpn);
+  goto out3;
+ }
+ /* Add a new block device limiting rule */
```

```
+ iothrottle_insert_node(iot, tmpn);
+out3:
+ ret = nbytes;
+ spin_unlock(&iot->lock);
+ if (n) {
+  synchronize_rcu();
+  kfree(n);
+ }
+out2:
+ cgroup_unlock();
+out1:
+ kfree(buffer);
+ return ret;
+}
+
+static s64 iothrottle_strategy_read(struct cgroup *cont, struct cftype *cft)
+{
+ struct iothrottle *iot;
+ s64 ret;
+
+ cgroup_lock();
+ if (cgroup_is_removed(cont)) {
+  cgroup_unlock();
+  return -ENODEV;
+ }
+ iot = cgroup_to_iothrottle(cont);
+ ret = iot->strategy;
+ cgroup_unlock();
+ return ret;
+}
+
+static int iothrottle_strategy_write(struct cgroup *cont,
+         struct cftype *cft, s64 val)
+{
+ struct iothrottle *iot;
+
+ cgroup_lock();
+ if (cgroup_is_removed(cont)) {
+  cgroup_unlock();
+  return -ENODEV;
+ }
+ iot = cgroup_to_iothrottle(cont);
+ iot->strategy = (int)val;
+ cgroup_unlock();
+ return 0;
+}
+
+static struct cftype files[] = {
```

```
+ {
+  .name = "bandwidth",
+  .read = iothrottle_read,
+  .write = iothrottle_write,
+ },
+ {
+  .name = "throttling_strategy",
+  .read_s64 = iothrottle_strategy_read,
+  .write_s64 = iothrottle_strategy_write,
+ },
+};
+
+static int iothrottle_populate(struct cgroup_subsys *ss, struct cgroup *cont)
+{
+ return cgroup_add_files(cont, ss, files, ARRAY_SIZE(files));
+}
+
+struct cgroup_subsys iothrottle_subsys = {
+ .name = "blockio",
+ .create = iothrottle_create,
+ .destroy = iothrottle_destroy,
+ .populate = iothrottle_populate,
+ .subsys_id = iothrottle_subsys_id,
+};
+
+static inline int __cant_sleep(void)
+{
+ return in_atomic() || in_interrupt() || irqs_disabled();
+}
+
+static long leaky_bucket(struct iothrottle_node *n, size_t bytes)
+{
+ unsigned long delta, t;
+ long sleep;
+
+ /* Account the i/o activity */
+ atomic_long_add(bytes, &n->stat);
+
+ /* Evaluate if we need to throttle the current process */
+ delta = (long)jiffies - (long)n->timestamp;
+ if (!delta)
+  return 0;
+
+ t = usecs_to_jiffies(KBS(atomic_long_read(&n->stat) / n->iorate));
+ if (!t)
+  return 0;
+
+ sleep = t - delta;
```

```
+ if (unlikely(sleep > 0))
+  return sleep;
+
+ /* Reset i/o statistics */
+ atomic_long_set(&n->stat, 0);
+ /*
+  * NOTE: be sure i/o statistics have been resetted before updating the
+  * timestamp, otherwise a very small time delta may possibly be read by
+  * another CPU w.r.t. accounted i/o statistics, generating unnecessary
+  * long sleeps.
+  */
+ smp_wmb();
+ n->timestamp = jiffies;
+ return 0;
+}
+
+/* XXX: need locking in order to evaluate a consistent sleep??? */
+static long token_bucket(struct iothrottle_node *n, size_t bytes)
+{
+ unsigned long delta;
+ long tok;
+
+ atomic_long_sub(bytes, &n->token);
+
+ delta = (long)jiffies - (long)n->timestamp;
+ if (!delta)
+  return 0;
+
+ n->timestamp = jiffies;
+ tok = atomic_long_read(&n->token) + jiffies_to_msecs(delta) * n->iorate;
+ if (tok > n->bucket_size)
+  tok = n->bucket_size;
+ atomic_long_set(&n->token, tok);
+
+ return (tok < 0) ? msecs_to_jiffies(-tok / n->iorate) : 0;
+}
+
+void cgroup_io_throttle(struct block_device *bdev, size_t bytes)
+{
+ struct iothrottle *iot;
+ struct iothrottle_node *n;
+ long sleep;
+
+ if (unlikely(!bdev || !bytes))
+  return;
+
+ iot = task_to_iothrottle(current);
+ if (unlikely(!iot))
```

```
+  return;
+
+ BUG_ON(!bdev->bd_inode);
+
+ rcu_read_lock();
+ n = iothrottle_search_node(iot, bdev->bd_inode->i_rdev);
+ if (!n || !n->iorate) {
+  rcu_read_unlock();
+  return;
+ }
+ switch (iot->strategy) {
+ case 0:
+  sleep = leaky_bucket(n, bytes);
+  break;
+ case 1:
+  sleep = token_bucket(n, bytes);
+  break;
+ default:
+  sleep = 0;
+ }
+ if (unlikely(sleep)) {
+  rcu_read_unlock();
+  if (__cant_sleep())
+   return;
+  pr_debug("io-throttle: task %p (%s) must sleep %lu jiffies\n",
+    current, current->comm, sleep);
+  schedule_timeout_killable(sleep);
+  return;
+ }
+ rcu_read_unlock();
+}
+EXPORT_SYMBOL(cgroup_io_throttle);
diff --git a/include/linux/blk-io-throttle.h b/include/linux/blk-io-throttle.h
new file mode 100644
index 0000000..3e08738
--- /dev/null
+++ b/include/linux/blk-io-throttle.h
@@ -0,0 +1,12 @@
+#ifndef BLK_IO_THROTTLE_H
+#define BLK_IO_THROTTLE_H
+
+#ifdef CONFIG_CGROUP_IO_THROTTLE
+extern void cgroup_io_throttle(struct block_device *bdev, size_t bytes);
+#else
+static inline void cgroup_io_throttle(struct block_device *bdev, size_t bytes)
+{
+}
+#endif /* CONFIG_CGROUP_IO_THROTTLE */
```

```
+
+#endif /* BLK_IO_THROTTLE_H */
diff --git a/include/linux/cgroup_subsys.h b/include/linux/cgroup_subsys.h
index e287745..0caf3c2 100644
--- a/include/linux/cgroup_subsys.h
+++ b/include/linux/cgroup_subsys.h
@@ -48,3 +48,9 @@ SUBSYS(devices)
 #endif

 /* */
+
+#ifdef CONFIG_CGROUP_IO_THROTTLE
+SUBSYS(iothrottle)
+#endif
+
+/* */
diff --git a/init/Kconfig b/init/Kconfig
index 6199d11..3117d99 100644
--- a/init/Kconfig
+++ b/init/Kconfig
@@ -306,6 +306,16 @@ config CGROUP_DEVICE
    Provides a cgroup implementing whitelists for devices which
    a process in the cgroup can mknod or open.

+config CGROUP_IO_THROTTLE
+ bool "Enable cgroup I/O throttling (EXPERIMENTAL)"
+ depends on CGROUPS && EXPERIMENTAL
+ help
+   This allows to limit the maximum I/O bandwidth for specific
+   cgroup(s).
+   See Documentation/controllers/io-throttle.txt for more information.
+
+   If unsure, say N.
+
 config CPUSETS
  bool "Cpuset support"
  depends on SMP && CGROUPS
```

_____

Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [PATCH 2/3] i/o bandwidth controller infrastructure
Posted by akpm on Thu, 26 Jun 2008 00:29:00 GMT
View Forum Message <> Reply to Message

On Fri, 20 Jun 2008 12:05:34 +0200

Andrea Righi <righi.andrea@gmail.com> wrote:

> This is the core io-throttle kernel infrastructure. It creates the basic
> interfaces to cgroups and implements the I/O measurement and throttling
> functions.
>
> Signed-off-by: Andrea Righi <righi.andrea@gmail.com>
> ---
> block/Makefile | 2 +
> block/blk-io-throttle.c | 393 ++
> include/linux/blk-io-throttle.h | 12 ++
> include/linux/cgroup_subsys.h | 6 +
> init/Kconfig | 10 +
> 5 files changed, 423 insertions(+), 0 deletions(-)
> create mode 100644 block/blk-io-throttle.c
> create mode 100644 include/linux/blk-io-throttle.h
>
> diff --git a/block/Makefile b/block/Makefile
> index 5a43c7d..8dec69b 100644
> --- a/block/Makefile
> +++ b/block/Makefile
> @@ -14,3 +14,5 @@ obj-$(CONFIG_IOSCHED_CFQ) += cfq-iosched.o
>
> obj-$(CONFIG_BLK_DEV_IO_TRACE) += blktrace.o
> obj-$(CONFIG_BLOCK_COMPAT) += compat_ioctl.o
> +
> +obj-$(CONFIG_CGROUP_IO_THROTTLE) += blk-io-throttle.o
> diff --git a/block/blk-io-throttle.c b/block/blk-io-throttle.c
> new file mode 100644
> index 0000000..4ec02bb
> --- /dev/null
> +++ b/block/blk-io-throttle.c
> @@ -0,0 +1,393 @@
> +/*
> + * blk-io-throttle.c
> + *
> + * This program is free software; you can redistribute it and/or
> + * modify it under the terms of the GNU General Public
> + * License as published by the Free Software Foundation; either
> + * version 2 of the License, or (at your option) any later version.
> + *
> + * This program is distributed in the hope that it will be useful,
> + * but WITHOUT ANY WARRANTY; without even the implied warranty of
> + * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
> + * General Public License for more details.
> + *
> + * You should have received a copy of the GNU General Public
> + * License along with this program; if not, write to the

> + * Free Software Foundation, Inc., 59 Temple Place - Suite 330,
> + * Boston, MA 021110-1307, USA.
> + *
> + * Copyright (C) 2008 Andrea Righi <righi.andrea@gmail.com>
> + */
> +
> +#include <linux/init.h>
> +#include <linux/module.h>
> +#include <linux/cgroup.h>
> +#include <linux/slab.h>
> +#include <linux/gfp.h>
> +#include <linux/err.h>
> +#include <linux/sched.h>
> +#include <linux/fs.h>
> +#include <linux/jiffies.h>
> +#include <linux/hardirq.h>
> +#include <linux/list.h>
> +#include <linux/spinlock.h>
> +#include <linux/uaccess.h>
> +#include <linux/vmalloc.h>
> +#include <linux/blk-io-throttle.h>
> +
> +#define ONE_SEC 1000000L /* # of microseconds in a second */

Remove this, use USEC_PER_SEC throughout.

> +#define KBS(x) ((x) * ONE_SEC >> 10)

Convert to lower-case-named C function, please.

> +
> +struct iothrottle_node {
> + struct list_head node;
> + dev_t dev;
> + unsigned long iorate;
> + unsigned long timestamp;
> + atomic_long_t stat;
> +};

Please document each field in structures. This is usually more useful
and important than documenting the code whcih manipulates those fields.

It is important that the units of fields such as iorate and timestamp
and stamp be documented.

> +struct iothrottle {
> + struct cgroup_subsys_state css;
> + /* protects the list below, not the single elements */

> + spinlock_t lock;
> + struct list_head list;
> +};

Looking elsewhere in the code it appears that some RCU-based locking is performed. That should be documented somewhere. Fully. At the definition site of the data whcih is RCU-protceted would be a good site.

> +static inline struct iothrottle *cgroup_to_iothrottle(struct cgroup *cont)
> +{
> + return container_of(cgroup_subsys_state(cont, iothrottle_subsys_id),
> + struct iothrottle, css);
> +}
> +
> +static inline struct iothrottle *task_to_iothrottle(struct task_struct *task)
> +{
> + return container_of(task_subsys_state(task, iothrottle_subsys_id),
> + struct iothrottle, css);
> +}
> +
> +static inline struct iothrottle_node *iothrottle_search_node(
> + const struct iothrottle *iot,
> + dev_t dev)
> +{
> + struct iothrottle_node *n;
> +
> + list_for_each_entry_rcu(n, &iot->list, node)
> + if (n->dev == dev)
> + return n;
> + return NULL;
> +}

This will be too large for inlining.

This function presumably has caller-provided locking requirements? They should be documented here.

> +static inline void iothrottle_insert_node(struct iothrottle *iot,
> + struct iothrottle_node *n)
> +{
> + list_add_rcu(&n->node, &iot->list);
> +}
> +
> +static inline struct iothrottle_node *iothrottle_replace_node(
> + struct iothrottle *iot,
> + struct iothrottle_node *old,

> + struct iothrottle_node *new)
> +{
> + list_replace_rcu(&old->node, &new->node);
> + return old;
> +}

Dittoes.

> +static inline struct iothrottle_node *iothrottle_delete_node(
> + struct iothrottle *iot,
> + dev_t dev)
> +{
> + struct iothrottle_node *n;
> +
> + list_for_each_entry(n, &iot->list, node)
> + if (n->dev == dev) {
> + list_del_rcu(&n->node);
> + return n;
> + }
> + return NULL;
> +}

Too large for inlining.

Was list_for_each_entry_rcu() needed?

Does this function have any caller-provided locking requirements?

> +/*
> + * Note: called from kernel/cgroup.c with cgroup_lock() held.
> + */
> +static struct cgroup_subsys_state *iothrottle_create(
> + struct cgroup_subsys *ss, struct cgroup *cont)

static struct cgroup_subsys_state *iothrottle_create(struct cgroup_subsys *ss,
 struct cgroup *cont)

would be more typical code layout (here and elsewhere)

static struct cgroup_subsys_state *
iothrottle_create(struct cgroup_subsys *ss, struct cgroup *cont)

is another way.

> +{
> + struct iothrottle *iot;
> +

```
> + iot = kmalloc(sizeof(*iot), GFP_KERNEL);
> + if (unlikely(!iot))
> +  return ERR_PTR(-ENOMEM);
> +
> + INIT_LIST_HEAD(&iot->list);
> + spin_lock_init(&iot->lock);
> +
> + return &iot->css;
> +}
> +
> +/*
> + * Note: called from kernel/cgroup.c with cgroup_lock() held.
> + */
> +static void iothrottle_destroy(struct cgroup_subsys *ss, struct cgroup *cont)
> +{
> + struct iothrottle_node *n, *p;
> + struct iothrottle *iot = cgroup_to_iothrottle(cont);
> +
> + /*
> +  * don't worry about locking here, at this point there must be not any
> +  * reference to the list.
> +  */
> + list_for_each_entry_safe(n, p, &iot->list, node)
> +  kfree(n);
> + kfree(iot);
> +}
> +
> +static ssize_t iothrottle_read(struct cgroup *cont,
> +         struct cftype *cft,
> +         struct file *file,
> +         char __user *userbuf,
> +         size_t nbytes,
> +         loff_t *ppos)
> +{
> + struct iothrottle *iot;
> + char *buffer;
> + int s = 0;
> + struct iothrottle_node *n;
> + ssize_t ret;
> +
> + buffer = kmalloc(nbytes + 1, GFP_KERNEL);
> + if (!buffer)
> +  return -ENOMEM;
> +
> + cgroup_lock();
> + if (cgroup_is_removed(cont)) {
> +  ret = -ENODEV;
> +  goto out;
```

```
> + }
> +
> + iot = cgroup_to_iothrottle(cont);
> + rcu_read_lock();
> + list_for_each_entry_rcu(n, &iot->list, node) {
> +  unsigned long delta, rate;
> +
> +  BUG_ON(!n->dev);
> +  delta = jiffies_to_usecs((long)jiffies - (long)n->timestamp);
> +  rate = delta ? KBS(atomic_long_read(&n->stat) / delta) : 0;
> +  s += scnprintf(buffer + s, nbytes - s,
> +         "=== device (%u,%u) ===\n"
> +         "  bandwidth limit: %lu KiB/sec\n"
> +         "current i/o usage: %lu KiB/sec\n",
> +         MAJOR(n->dev), MINOR(n->dev),
> +         n->iorate, rate);
> + }
> + rcu_read_unlock();
> + ret = simple_read_from_buffer(userbuf, nbytes, ppos, buffer, s);
> +out:
> + cgroup_unlock();
> + kfree(buffer);
> + return ret;
> +}
```

This is a kernel->userspace interface. It is part of the kernel ABI.
We will need to support in a back-compatible fashion for ever. Hence
it is important. The entire proposed kernel<->userspace interface
should be completely described in the changelog or the documentation so
that we can understand and review what you are proposing.

```
> +static inline dev_t devname2dev_t(const char *buf)
> +{
> + struct block_device *bdev;
> + dev_t ret;
> +
> + bdev = lookup_bdev(buf);
> + if (IS_ERR(bdev))
> +  return 0;
> +
> + BUG_ON(!bdev->bd_inode);
> + ret = bdev->bd_inode->i_rdev;
> + bdput(bdev);
> +
> + return ret;
> +}
```

Too large to inline. I get tired of telling people this. Please just

remove all the inlining from all the patches. Then go back and
selectively inline any functions which really do need to be inlined
(overall reduction in generated .text is a good heuristic).

How can this function not be racy? We're returning a dev_t which
refers to a device upon which we have no reference. A better design
might be to rework the who9le thing to operate on a `struct
block_device *' upon whcih this code holds a reference, rather than
using bare dev_t.

I _guess_ it's OK doing an in-kernel filesystem lookup here. But did
you consider just passing the dev_t in from userspace? It's just a
stat().

Does all this code treat /dev/sda1 as a separate device from /dev/sda2?
 If so, that would be broken.

> +static inline int iothrottle_parse_args(char *buf, size_t nbytes,
> +   dev_t *dev, unsigned long *val)
> +{
> + char *p;
> +
> + p = memchr(buf, ':', nbytes);
> + if (!p)
> + return -EINVAL;
> + *p++ = '\0';
> +
> + /* i/o bandiwth is expressed in KiB/s */

typo.

This comment is incorrect, isn't it? Or at least misleading. The
bandwidth can be expressed in an exotically broad number of different
ways.

> + *val = ALIGN(memparse(p, &p), 1024) >> 10;
> + if (*p)
> + return -EINVAL;
> +
> + *dev = devname2dev_t(buf);
> + if (!*dev)
> + return -ENOTBLK;
> +
> + return 0;
> +}

uninline...

I think the whole memparse() thing is over the top:

+- BANDWIDTH is the maximum I/O bandwidth on DEVICE allowed by CGROUP (we can
+ use a suffix k, K, m, M, g or G to indicate bandwidth values in KB/s, MB/s
+ or GB/s),

For starters, we don't _display_ the bacndwidth back to the user in the
units with which it was written, so what's the point?

Secondly, we hope and expect that humans won't be diorectly echoing raw
data into kernel pseudo files. We shouild expect and plan for (or even
write) front-end management applications. And such applications won't
need these ornate designed-for-human interfaces.

IOW: I'd suggest this interface be changed to accept a plain old 64-bit
bytes-per-second and leave it at that.

> +static ssize_t iothrottle_write(struct cgroup *cont,
> +   struct cftype *cft,
> +   struct file *file,
> +   const char __user *userbuf,
> +   size_t nbytes, loff_t *ppos)
> +{
> + struct iothrottle *iot;
> + struct iothrottle_node *n, *tmpn = NULL;
> + char *buffer, *tmpp;

Please avoid variables called tmp or things derived from it. Surely we
can think of some more communucative identifier?

> + dev_t dev;
> + unsigned long val;
> + int ret;
> +
> + if (!nbytes)
> + return -EINVAL;
> +
> + /* Upper limit on largest io-throttle rule string user might write. */
> + if (nbytes > 1024)
> + return -E2BIG;
> +
> + buffer = kmalloc(nbytes + 1, GFP_KERNEL);
> + if (!buffer)
> + return -ENOMEM;
> +
> + if (copy_from_user(buffer, userbuf, nbytes)) {
> + ret = -EFAULT;
> + goto out1;

```
> + }
> +
> + buffer[nbytes] = '\0';
```

strncpy_from_user()? (I'm not sure that strncpy_from_user() does the
null-termination as desired).

```
> + tmpp = strstrip(buffer);
> +
> + ret = iothrottle_parse_args(tmpp, nbytes, &dev, &val);
> + if (ret)
> +  goto out1;
> +
> + if (val) {
> +  tmpn = kmalloc(sizeof(*tmpn), GFP_KERNEL);
> +  if (!tmpn) {
> +   ret = -ENOMEM;
> +   goto out1;
> +  }
> +  atomic_long_set(&tmpn->stat, 0);
> +  tmpn->timestamp = jiffies;
> +  tmpn->iorate = val;
> +  tmpn->dev = dev;
> + }
> +
> + cgroup_lock();
> + if (cgroup_is_removed(cont)) {
> +  ret = -ENODEV;
> +  goto out2;
> + }
> +
> + iot = cgroup_to_iothrottle(cont);
> + spin_lock(&iot->lock);
> + if (!val) {
> +  /* Delete a block device limiting rule */
> +  n = iothrottle_delete_node(iot, dev);
> +  goto out3;
> + }
> + n = iothrottle_search_node(iot, dev);
> + if (n) {
> +  /* Update a block device limiting rule */
> +  iothrottle_replace_node(iot, n, tmpn);
> +  goto out3;
> + }
> + /* Add a new block device limiting rule */
> + iothrottle_insert_node(iot, tmpn);
> +out3:
> + ret = nbytes;
```

```
> + spin_unlock(&iot->lock);
> + if (n) {
> +  synchronize_rcu();
> +  kfree(n);
> + }
> +out2:
> + cgroup_unlock();
> +out1:
> + kfree(buffer);
> + return ret;
> +}
> +
> +static struct cftype files[] = {
> + {
> +  .name = "bandwidth",
> +  .read = iothrottle_read,
> +  .write = iothrottle_write,
> + },
> +};
> +
> +static int iothrottle_populate(struct cgroup_subsys *ss, struct cgroup *cont)
> +{
> + return cgroup_add_files(cont, ss, files, ARRAY_SIZE(files));
> +}
> +
> +struct cgroup_subsys iothrottle_subsys = {
> + .name = "blockio",
> + .create = iothrottle_create,
> + .destroy = iothrottle_destroy,
> + .populate = iothrottle_populate,
> + .subsys_id = iothrottle_subsys_id,
> +};
> +
> +static inline int __cant_sleep(void)
> +{
> + return in_atomic() || in_interrupt() || irqs_disabled();
> +}
```

err, no.

I don't know what this is doing or why it was added, but whatever it is
it's a hack and it all needs to go away.

Please describe what problem this is trying to solve and let's take a
look at it. That should have been covered in code comments anyway.
But because it wasn't, I am presently unable to help.

> +void cgroup_io_throttle(struct block_device *bdev, size_t bytes)

```
> +{
> + struct iothrottle *iot;
> + struct iothrottle_node *n;
> + unsigned long delta, t;
> + long sleep;
> +
> + if (unlikely(!bdev || !bytes))
> +  return;
> +
> + iot = task_to_iothrottle(current);
> + if (unlikely(!iot))
> +  return;
> +
> + BUG_ON(!bdev->bd_inode);
> +
> + rcu_read_lock();
> + n = iothrottle_search_node(iot, bdev->bd_inode->i_rdev);
> + if (!n || !n->iorate)
> +  goto out;
> +
> + /* Account the i/o activity */
> + atomic_long_add(bytes, &n->stat);
> +
> + /* Evaluate if we need to throttle the current process */
> + delta = (long)jiffies - (long)n->timestamp;
> + if (!delta)
> +  goto out;
> +
> + t = usecs_to_jiffies(KBS(atomic_long_read(&n->stat) / n->iorate));
```

Are you sure that n->iorate cannot be set to zero between the above test and this division? Taking a copy into a local variable would fix that small race.

```
> + if (!t)
> +  goto out;
> +
> + sleep = t - delta;
> + if (unlikely(sleep > 0)) {
> +  rcu_read_unlock();
> +  if (__cant_sleep())
> +   return;
> +  pr_debug("io-throttle: task %p (%s) must sleep %lu jiffies\n",
> +    current, current->comm, delta);
> +  schedule_timeout_killable(sleep);
> +  return;
> + }
> + /* Reset i/o statistics */
```

> + atomic_long_set(&n->stat, 0);
> + /*
> + * NOTE: be sure i/o statistics have been resetted before updating the
> + * timestamp, otherwise a very small time delta may possibly be read by
> + * another CPU w.r.t. accounted i/o statistics, generating unnecessary
> + * long sleeps.
> + */
> + smp_wmb();
> + n->timestamp = jiffies;
> +out:
> + rcu_read_unlock();
> +}
> +EXPORT_SYMBOL(cgroup_io_throttle);

I'm confused. This code is using jiffies but the string "HZ" doesn't
appears anywhere in the diff. Where are we converting from the
kernel-internal HZ rate into suitable-for-exposing-to-userspace units?

HZ can vary from 100 to 1000 (approx). What are the implications of
this for the accuracy of this code?

I have no comments on the overall design. I'm not sure that I understand
it yet.

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [PATCH 2/3] i/o bandwidth controller infrastructure
Posted by Andrea Righi on Thu, 26 Jun 2008 22:36:46 GMT
View Forum Message <> Reply to Message

Andrew,

thanks for your time and the detailed revision. I'll try to fix
everything you reported and better document the code according to your
suggestions. I'll re-submit a new patchset version ASAP.

A few comments below.

Andrew Morton wrote:
[snip]
>> +static ssize_t iothrottle_read(struct cgroup *cont,
>> +   struct cftype *cft,
>> +   struct file *file,

```
>> +          char __user *userbuf,
>> +          size_t nbytes,
>> +          loff_t *ppos)
>> +{
>> + struct iothrottle *iot;
>> + char *buffer;
>> + int s = 0;
>> + struct iothrottle_node *n;
>> + ssize_t ret;
>> +
>> + buffer = kmalloc(nbytes + 1, GFP_KERNEL);
>> + if (!buffer)
>> +  return -ENOMEM;
>> +
>> + cgroup_lock();
>> + if (cgroup_is_removed(cont)) {
>> +  ret = -ENODEV;
>> +  goto out;
>> + }
>> +
>> + iot = cgroup_to_iothrottle(cont);
>> + rcu_read_lock();
>> + list_for_each_entry_rcu(n, &iot->list, node) {
>> +  unsigned long delta, rate;
>> +
>> +  BUG_ON(!n->dev);
>> +  delta = jiffies_to_usecs((long)jiffies - (long)n->timestamp);
>> +  rate = delta ? KBS(atomic_long_read(&n->stat) / delta) : 0;
>> +  s += scnprintf(buffer + s, nbytes - s,
>> +          "=== device (%u,%u) ===\n"
>> +          "  bandwidth limit: %lu KiB/sec\n"
>> +          "current i/o usage: %lu KiB/sec\n",
>> +          MAJOR(n->dev), MINOR(n->dev),
>> +          n->iorate, rate);
>> + }
>> + rcu_read_unlock();
>> + ret = simple_read_from_buffer(userbuf, nbytes, ppos, buffer, s);
>> +out:
>> + cgroup_unlock();
>> + kfree(buffer);
>> + return ret;
>> +}
>
> This is a kernel->userspace interface.  It is part of the kernel ABI.
> We will need to support in a back-compatible fashion for ever.  Hence
> it is important.  The entire proposed kernel<->userspace interface
> should be completely described in the changelog or the documentation so
> that we can understand and review what you are proposing.
```

and BTW I was actually wondering if the output could be changed with
something less human readable and better parseable, I means just print
only raw numbers. And describe the semantic in the documentation.

>> +static inline dev_t devname2dev_t(const char *buf)
>> +{
>> + struct block_device *bdev;
>> + dev_t ret;
>> +
>> + bdev = lookup_bdev(buf);
>> + if (IS_ERR(bdev))
>> + return 0;
>> +
>> + BUG_ON(!bdev->bd_inode);
>> + ret = bdev->bd_inode->i_rdev;
>> + bdput(bdev);
>> +
>> + return ret;
>> +}
>
> Too large to inline. I get tired of telling people this. Please just
> remove all the inlining from all the patches. Then go back and
> selectively inline any functions which really do need to be inlined
> (overall reduction in generated .text is a good heuristic).
>
> How can this function not be racy? We're returning a dev_t which
> refers to a device upon which we have no reference. A better design
> might be to rework the who9le thing to operate on a `struct
> block_device *' upon whcih this code holds a reference, rather than
> using bare dev_t.

However, holding a reference wouldn't allow to unplug the device, i.e.
a USB disk. As reported in Documentation/controllers/io-throttle.txt:

WARNING: per-block device limiting rules always refer to the dev_t
device number. If a block device is unplugged (i.e. a USB device) the
limiting rules defined for that device persist and they are still valid
if a new device is plugged in the system and it uses the same major and
minor numbers.

This would be a feature in my case, but I don't know if it would be a
bug in general.

> I _guess_ it's OK doing an in-kernel filesystem lookup here. But did
> you consider just passing the dev_t in from userspace? It's just a
> stat().

Yes, and that seems to be more reasonable, since we display major,minor
numbers in the output.

> Does all this code treat /dev/sda1 as a separate device from /dev/sda2?
> If so, that would be broken.

Yes, all the partitions are treated as separate devices with
(potentially) different limiting rules, but I don't understand why it
would be broken... dev_t has both minor and major numbers, so it would
be possible to select single partitions as well.

>> +static inline int iothrottle_parse_args(char *buf, size_t nbytes,
>> +   dev_t *dev, unsigned long *val)
>> +{
>> + char *p;
>> +
>> + p = memchr(buf, ':', nbytes);
>> + if (!p)
>> + return -EINVAL;
>> + *p++ = '\0';
>> +
>> + /* i/o bandiwth is expressed in KiB/s */
>
> typo.
>
> This comment is incorrect, isn't it? Or at least misleading. The
> bandwidth can be expressed in an exotically broad number of different
> ways.

Yes.

>
>> + *val = ALIGN(memparse(p, &p), 1024) >> 10;
>> + if (*p)
>> + return -EINVAL;
>> +
>> + *dev = devname2dev_t(buf);
>> + if (!*dev)
>> + return -ENOTBLK;
>> +
>> + return 0;
>> +}
>
> uninline...
>
> I think the whole memparse() thing is over the top:
>
> +- BANDWIDTH is the maximum I/O bandwidth on DEVICE allowed by CGROUP (we can

> + use a suffix k, K, m, M, g or G to indicate bandwidth values in KB/s, MB/s
> + or GB/s),
>
> For starters, we don't _display_ the bacndwidth back to the user in the
> units with which it was written, so what's the point?
>
> Secondly, we hope and expect that humans won't be diorectly echoing raw
> data into kernel pseudo files. We shouild expect and plan for (or even
> write) front-end management applications. And such applications won't
> need these ornate designed-for-human interfaces.
>
> IOW: I'd suggest this interface be changed to accept a plain old 64-bit
> bytes-per-second and leave it at that.

I agree.

>> +void cgroup_io_throttle(struct block_device *bdev, size_t bytes)
>> +{
>> + struct iothrottle *iot;
>> + struct iothrottle_node *n;
>> + unsigned long delta, t;
>> + long sleep;
>> +
>> + if (unlikely(!bdev || !bytes))
>> + return;
>> +
>> + iot = task_to_iothrottle(current);
>> + if (unlikely(!iot))
>> + return;
>> +
>> + BUG_ON(!bdev->bd_inode);
>> +
>> + rcu_read_lock();
>> + n = iothrottle_search_node(iot, bdev->bd_inode->i_rdev);
>> + if (!n || !n->iorate)
>> + goto out;
>> +
>> + /* Account the i/o activity */
>> + atomic_long_add(bytes, &n->stat);
>> +
>> + /* Evaluate if we need to throttle the current process */
>> + delta = (long)jiffies - (long)n->timestamp;
>> + if (!delta)
>> + goto out;
>> +
>> + t = usecs_to_jiffies(KBS(atomic_long_read(&n->stat) / n->iorate));
>
> Are you sure that n->iorate cannot be set to zero between the above

> test and this division? Taking a copy into a local variable would fix
> that small race.

n->iorate can only change via userspace->kernel interface, that just
replaces the node in the list using the rcu way. AFAIK this shouldn't be
racy, but better to do it using the local variable to avoid future bugs.

>> + if (!t)
>> + goto out;
>> +
>> + sleep = t - delta;
>> + if (unlikely(sleep > 0)) {
>> + rcu_read_unlock();
>> + if (__cant_sleep())
>> +   return;
>> + pr_debug("io-throttle: task %p (%s) must sleep %lu jiffies\n",
>> +   current, current->comm, delta);
>> + schedule_timeout_killable(sleep);
>> + return;
>> + }
>> + /* Reset i/o statistics */
>> + atomic_long_set(&n->stat, 0);
>> + /*
>> + * NOTE: be sure i/o statistics have been resetted before updating the
>> + * timestamp, otherwise a very small time delta may possibly be read by
>> + * another CPU w.r.t. accounted i/o statistics, generating unnecessary
>> + * long sleeps.
>> + */
>> + smp_wmb();
>> + n->timestamp = jiffies;
>> +out:
>> + rcu_read_unlock();
>> +}
>> +EXPORT_SYMBOL(cgroup_io_throttle);
>
> I'm confused. This code is using jiffies but the string "HZ" doesn't
> appears anywhere in the diff. Where are we converting from the
> kernel-internal HZ rate into suitable-for-exposing-to-userspace units?
>
> HZ can vary from 100 to 1000 (approx). What are the implications of
> this for the accuracy of this code?

The code uses jiffies_to_usecs() and usecs_to_jiffies(), that should be
ok, isn't it?

-Andrea

_____
Containers mailing list

Subject: Re: [PATCH 2/3] i/o bandwidth controller infrastructure
Posted by akpm on Thu, 26 Jun 2008 22:59:48 GMT
View Forum Message <> Reply to Message

On Fri, 27 Jun 2008 00:36:46 +0200
Andrea Righi <righi.andrea@gmail.com> wrote:

> > Does all this code treat /dev/sda1 as a separate device from /dev/sda2?
> > If so, that would be broken.
>
> Yes, all the partitions are treated as separate devices with
> (potentially) different limiting rules, but I don't understand why it
> would be broken... dev_t has both minor and major numbers, so it would
> be possible to select single partitions as well.

Well it's functionally broken, isn't it? A physical disk has a fixed
IO bandwidth and when the administrator wants to partition that
bandwidth amongst control groups he will need to consider the entire
device when doing so?

I mean, the whole point of this feature and of control groups as a
whole is isolation. But /dev/sda1 and /dev/sda2 are very much _not_
isolated. Whereas /dev/sda and /dev/sdb are (to a large degree)
isolated.

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

Subject: Re: [PATCH 2/3] i/o bandwidth controller infrastructure
Posted by Andrea Righi on Fri, 27 Jun 2008 10:53:28 GMT
View Forum Message <> Reply to Message

Andrew Morton wrote:
> On Fri, 27 Jun 2008 00:36:46 +0200
> Andrea Righi <righi.andrea@gmail.com> wrote:
>
>>> Does all this code treat /dev/sda1 as a separate device from /dev/sda2?
>>> If so, that would be broken.
>> Yes, all the partitions are treated as separate devices with
>> (potentially) different limiting rules, but I don't understand why it

>> would be broken... dev_t has both minor and major numbers, so it would
>> be possible to select single partitions as well.
>
> Well it's functionally broken, isn't it? A physical disk has a fixed
> IO bandwidth and when the administrator wants to partition that
> bandwidth amongst control groups he will need to consider the entire
> device when doing so?
>
> I mean, the whole point of this feature and of control groups as a
> whole is isolation. But /dev/sda1 and /dev/sda2 are very much _not_
> isolated. Whereas /dev/sda and /dev/sdb are (to a large degree)
> isolated.

well... yes, sounds reasonable. In this case we could just ignore the
minor number and consider only major number as the key to identify a
specific block device (both for userspace<->kernel interface and when
accounting/throttling i/o requests).

-Andrea

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [PATCH 2/3] i/o bandwidth controller infrastructure
Posted by Andrea Righi on Mon, 30 Jun 2008 16:10:54 GMT
View Forum Message <> Reply to Message

Andrea Righi wrote:
> Andrew Morton wrote:
>> On Fri, 27 Jun 2008 00:36:46 +0200
>> Andrea Righi <righi.andrea@gmail.com> wrote:
>>
>>>> Does all this code treat /dev/sda1 as a separate device from /dev/sda2?
>>>> If so, that would be broken.
>>> Yes, all the partitions are treated as separate devices with
>>> (potentially) different limiting rules, but I don't understand why it
>>> would be broken... dev_t has both minor and major numbers, so it would
>>> be possible to select single partitions as well.
>> Well it's functionally broken, isn't it? A physical disk has a fixed
>> IO bandwidth and when the administrator wants to partition that
>> bandwidth amongst control groups he will need to consider the entire
>> device when doing so?
>>
>> I mean, the whole point of this feature and of control groups as a
>> whole is isolation. But /dev/sda1 and /dev/sda2 are very much _not_
>> isolated. Whereas /dev/sda and /dev/sdb are (to a large degree)

>> isolated.
>
> well... yes, sounds reasonable. In this case we could just ignore the
> minor number and consider only major number as the key to identify a
> specific block device (both for userspace<->kernel interface and when
> accounting/throttling i/o requests).

oops.. no, this is obviously wrong. So, I dunno if it would be better to add complexity in cgroup_io_throttle() to identify the disk a partition belongs or to just use the struct block_device as key, instead of dev_t, as you intially suggested. I'll investigate.

-Andrea

_____

Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers