
Subject: [PATCH -mm 2/3] i/o bandwidth controller infrastructure

Posted by [Andrea Righi](#) on Sat, 12 Jul 2008 11:31:30 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 | 549 +++++
include/linux/blk-io-throttle.h | 41 +++
include/linux/cgroup_subsys.h | 6 +
init/Kconfig            | 10 +
5 files changed, 608 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 208000b..b3afc86 100644
```

```
--- a/block/Makefile
```

```
+++ b/block/Makefile
```

```
@@ -13,6 +13,8 @@ obj-$(CONFIG_IOSCHED_AS) += as-iosched.o
```

```
obj-$(CONFIG_IOSCHED_DEADLINE) += deadline-iosched.o
```

```
obj-$(CONFIG_IOSCHED_CFQ) += cfq-iosched.o
```

```
+obj-$(CONFIG_CGROUP_IO_THROTTLE) += blk-io-throttle.o
```

```
+
```

```
obj-$(CONFIG_BLK_DEV_IO_TRACE) += blktrace.o
```

```
obj-$(CONFIG_BLOCK_COMPAT) += compat_ioctl.o
```

```
obj-$(CONFIG_BLK_DEV_INTEGRITY) += blk-integrity.o
```

```
diff --git a/block/blk-io-throttle.c b/block/blk-io-throttle.c
```

```
new file mode 100644
```

```
index 0000000..82100a6
```

```
--- /dev/null
```

```
+++ b/block/blk-io-throttle.c
```

```
@@ -0,0 +1,549 @@
```

```
+/*
```

```
+ * 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/genhd.h>
+#include <linux/fs.h>
+#include <linux/jiffies.h>
+#include <linux/hardirq.h>
+#include <linux/list.h>
+#include <linux/seq_file.h>
+#include <linux/spinlock.h>
+#include <linux/uaccess.h>
+#include <linux/blk-io-throttle.h>
+
+/* The various types of throttling algorithms */
+enum iothrottle_strategy {
+ IOTHROTTLE_LEAKY_BUCKET,
+ IOTHROTTLE_TOKEN_BUCKET,
+};
+
+/**
+ * struct iothrottle_node - throttling rule of a single block device
+ * @node: list of per block device throttling rules
+ * @dev: block device number, used as key in the list
+ * @iorate: max i/o bandwidth (in bytes/s)
+ * @strategy: throttling strategy
+ * @timestamp: timestamp of the last I/O request (in jiffies)
+ * @stat: i/o activity counter (leaky bucket only)
+ * @bucket_size: bucket size in bytes (token bucket only)
+ * @token: token counter (token bucket only)
+ *
+ * Define a i/o throttling rule for a single block device.
+ *
+ * NOTE: limiting rules always refer to dev_t; if a block device is unplugged

```

```

+ * the limiting rules defined for that device persist and they are still valid
+ * if a new device is plugged and it uses the same dev_t number.
+ */
+struct iothrottle_node {
+ struct list_head node;
+ dev_t dev;
+ u64 iorate;
+ enum iothrottle_strategy strategy;
+ unsigned long timestamp;
+ atomic_long_t stat;
+ s64 bucket_size;
+ atomic_long_t token;
+};
+
+/**
+ * struct iothrottle - throttling rules for a cgroup
+ * @css: pointer to the cgroup state
+ * @lock: spinlock used to protect write operations in the list
+ * @list: list of iothrottle_node elements
+ *
+ * Define multiple per-block device i/o throttling rules.
+ * Note: the list of the throttling rules is protected by RCU locking.
+ */
+struct iothrottle {
+ struct cgroup_subsys_state css;
+ spinlock_t lock;
+ struct list_head list;
+};
+
+static inline struct iothrottle *cgroup_to_iothrottle(struct cgroup *cgrp)
+{
+ return container_of(cgroup_subsys_state(cgrp, 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);
+}
+
+/**
+ * Note: called with rcu_read_lock() or iot->lock held.
+ */
+static 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;
+}
+
+/*
+ * Note: called with iot->lock held.
+ */
+static inline void iothrottle_insert_node(struct iothrottle *iot,
+ struct iothrottle_node *n)
+{
+ list_add_rcu(&n->node, &iot->list);
+}
+
+/*
+ * Note: called with iot->lock held.
+ */
+static inline void
+iothrottle_replace_node(struct iothrottle *iot, struct iothrottle_node *old,
+ struct iothrottle_node *new)
+{
+ list_replace_rcu(&old->node, &new->node);
+}
+
+/*
+ * Note: called with iot->lock held.
+ */
+static 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 *cgrp)
+{

```

```

+ 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 *cgrp)
+{
+ struct iothrottle_node *n, *p;
+ struct iothrottle *iot = cgroup_to_iothrottle(cgrp);
+
+ /*
+ * 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 int iothrottle_read(struct cgroup *cgrp, struct cftype *cft,
+ struct seq_file *m)
+{
+ struct iothrottle *iot = cgroup_to_iothrottle(cgrp);
+ struct iothrottle_node *n;
+
+ rcu_read_lock();
+ list_for_each_entry_rcu(n, &iot->list, node) {
+ unsigned long delta;
+
+ BUG_ON(!n->dev);
+ delta = jiffies_to_msecs((long)jiffies - (long)n->timestamp);
+ seq_printf(m, "%u %u %llu %u %li %lli %li %lu\n",
+ MAJOR(n->dev), MINOR(n->dev), n->iorate,
+ n->strategy, atomic_long_read(&n->stat),
+ n->bucket_size, atomic_long_read(&n->token),
+ delta);
+ }
+ rcu_read_unlock();

```

```

+ return 0;
+}
+
+static dev_t devname2dev_t(const char *buf)
+{
+ struct block_device *bdev;
+ dev_t dev = 0;
+ struct gendisk *disk;
+ int part;
+
+ /* use a lookup to validate the block device */
+ bdev = lookup_bdev(buf);
+ if (IS_ERR(bdev))
+ return 0;
+
+ /* only entire devices are allowed, not single partitions */
+ disk = get_gendisk(bdev->bd_dev, &part);
+ if (disk && !part) {
+ BUG_ON(!bdev->bd_inode);
+ dev = bdev->bd_inode->i_rdev;
+ }
+ bdput(bdev);
+
+ return dev;
+}
+
+/*
+ * The userspace input string must use one of the following syntax:
+ *
+ * dev:0 <- delete a limiting rule
+ * dev:bw-limit:0 <- leaky bucket throttling rule
+ * dev:bw-limit:1:bucket-size <- token bucket throttling rule
+ */
+static int iothrottle_parse_args(char *buf, size_t nbytes, dev_t *dev,
+ u64 *iorate,
+ enum iothrottle_strategy *strategy,
+ s64 *bucket_size)
+{
+ char *p = buf;
+ int count = 0;
+ char *s[3];
+ unsigned long strategy_val;
+ int ret;
+
+ /* split the colon-delimited input string into its elements */
+ memset(s, 0, sizeof(s));
+ while (count < ARRAY_SIZE(s)) {
+ p = memchr(p, ':', buf + nbytes - p);

```

```

+ if (!p)
+ break;
+ *p++ = '\0';
+ if (p >= buf + nbytes)
+ break;
+ s[count++] = p;
+ }
+
+ /* i/o bandwidth limit */
+ if (!s[0])
+ return -EINVAL;
+ ret = strict_strtoull(s[0], 10, iorate);
+ if (ret < 0)
+ return ret;
+ if (!*iorate) {
+ /*
+  * we're deleting a limiting rule, so just ignore the other
+  * parameters
+  */
+ *strategy = 0;
+ *bucket_size = 0;
+ goto out;
+ }
+ *iorate = ALIGN(*iorate, 1024);
+
+ /* throttling strategy */
+ if (!s[1])
+ return -EINVAL;
+ ret = strict_strtoul(s[1], 10, &strategy_val);
+ if (ret < 0)
+ return ret;
+ *strategy = (enum iothrottle_strategy)strategy_val;
+ switch (*strategy) {
+ case IOTHROTTLLE_LEAKY_BUCKET:
+ /* leaky bucket ignores bucket size */
+ *bucket_size = 0;
+ goto out;
+ case IOTHROTTLLE_TOKEN_BUCKET:
+ break;
+ default:
+ return -EINVAL;
+ }
+
+ /* bucket size */
+ if (!s[2])
+ return -EINVAL;
+ ret = strict_strtoll(s[2], 10, bucket_size);
+ if (ret < 0)

```

```

+ return ret;
+ if (*bucket_size < 0)
+ return -EINVAL;
+ *bucket_size = ALIGN(*bucket_size, 1024);
+out:
+
+ /* block device number */
+ *dev = devname2dev_t(buf);
+ return *dev ? 0 : -EINVAL;
+}
+
+static int iothrottle_write(struct cgroup *cgrp, struct cftype *cft,
+ const char *buffer)
+{
+ struct iothrottle *iot;
+ struct iothrottle_node *n, *newn = NULL;
+ dev_t dev;
+ u64 iorate;
+ enum iothrottle_strategy strategy;
+ s64 bucket_size;
+ char *buf;
+ size_t nbytes = strlen(buffer);
+ int ret = 0;
+
+ buf = kmalloc(nbytes + 1, GFP_KERNEL);
+ if (!buf)
+ return -ENOMEM;
+ memcpy(buf, buffer, nbytes + 1);
+
+ ret = iothrottle_parse_args(buf, nbytes, &dev, &iorate,
+ &strategy, &bucket_size);
+ if (ret)
+ goto out1;
+ if (iorate) {
+ newn = kmalloc(sizeof(*newn), GFP_KERNEL);
+ if (!newn) {
+ ret = -ENOMEM;
+ goto out1;
+ }
+ newn->dev = dev;
+ newn->iorate = iorate;
+ newn->strategy = strategy;
+ newn->bucket_size = bucket_size;
+ newn->timestamp = jiffies;
+ atomic_long_set(&newn->stat, 0);
+ atomic_long_set(&newn->token, 0);
+ }
+ if (!cgroup_lock_live_group(cgrp)) {

```

```

+ kfree(newn);
+ ret = -ENODEV;
+ goto out1;
+ }
+ iot = cgroup_to_iothrottle(cgrp);
+
+ spin_lock(&iot->lock);
+ if (!iorate) {
+ /* Delete a block device limiting rule */
+ n = iothrottle_delete_node(iot, dev);
+ goto out2;
+ }
+ n = iothrottle_search_node(iot, dev);
+ if (n) {
+ /* Update a block device limiting rule */
+ iothrottle_replace_node(iot, n, newn);
+ goto out2;
+ }
+ /* Add a new block device limiting rule */
+ iothrottle_insert_node(iot, newn);
+out2:
+ spin_unlock(&iot->lock);
+ cgroup_unlock();
+ if (n) {
+ synchronize_rcu();
+ kfree(n);
+ }
+out1:
+ kfree(buf);
+ return ret;
+}
+
+static struct cftype files[] = {
+ {
+ .name = "bandwidth",
+ .read_seq_string = iothrottle_read,
+ .write_string = iothrottle_write,
+ },
+};
+
+static int iothrottle_populate(struct cgroup_subsys *ss, struct cgroup *cgrp)
+{
+ return cgroup_add_files(cgrp, 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,
+};
+
+/*
+ * Note: called with rcu_read_lock() held.
+ */
+static unsigned long leaky_bucket(struct iothrottle_node *n, ssize_t bytes)
+{
+ unsigned long delta, t;
+ long sleep, stat;
+
+ /* 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;
+
+ /*
+ * NOTE: n->iorate cannot be set to zero here, iorate can only change
+ * via the userspace->kernel interface that in case of update fully
+ * replaces the iothrottle_node pointer in the list, using the RCU way.
+ */
+ stat = atomic_long_read(&n->stat);
+ if (stat > 0) {
+ t = usecs_to_jiffies(stat * USEC_PER_SEC / 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;
+}
+

```

```

+/*
+ * Note: called with rcu_read_lock() held.
+ * XXX: need locking in order to evaluate a consistent sleep???
+ */
+static unsigned long token_bucket(struct iothrottle_node *n, ssize_t bytes)
+{
+ unsigned long iorate = n->iorate / MSEC_PER_SEC;
+ unsigned long delta;
+ long tok;
+
+ BUG_ON(!iorate);
+
+ atomic_long_sub(bytes, &n->token);
+ delta = jiffies_to_msecs((long)jiffies - (long)n->timestamp);
+ n->timestamp = jiffies;
+ tok = atomic_long_read(&n->token);
+ if (delta && tok < n->bucket_size) {
+ tok += delta * iorate;
+ pr_debug("io-throttle: adding %lu tokens\n", delta * iorate);
+ if (tok > n->bucket_size)
+ tok = n->bucket_size;
+ atomic_long_set(&n->token, tok);
+ }
+ atomic_long_set(&n->token, tok);
+
+ return (tok < 0) ? msecs_to_jiffies(-tok / iorate) : 0;
+}
+
+/**
+ * cgroup_io_throttle() - account and throttle i/o activity
+ * @bdev: block device involved for the i/o.
+ * @bytes: size in bytes of the i/o operation.
+ * @can_sleep: used to set to 1 if we're in a sleep()able context, 0
+ * otherwise; into a non-sleep()able context we only account the
+ * i/o activity without applying any throttling sleep.
+ *
+ * This is the core of the block device i/o bandwidth controller. This function
+ * must be called by any function that generates i/o activity (directly or
+ * indirectly). It provides both i/o accounting and throttling functionalities;
+ * throttling is disabled if @can_sleep is set to 0.
+ *
+ * Returns the value of sleep in jiffies if it was not possible to schedule the
+ * timeout.
+ */
+unsigned long
+cgroup_io_throttle(struct block_device *bdev, ssize_t bytes, int can_sleep)
+{
+ struct iothrottle *iot;

```

```

+ struct iothrottle_node *n;
+ dev_t dev;
+ unsigned long sleep;
+
+ if (unlikely(!bdev))
+ return 0;
+ /*
+ * WARNING: in_atomic() do not know about held spinlocks in
+ * non-preemptible kernels, but we want to check it here to raise
+ * potential bugs by preemptible kernels.
+ */
+ WARN_ON_ONCE(can_sleep &&
+ (irqs_disabled() || in_interrupt() || in_atomic()));
+
+ /*
+ * AIO is accounted in io_submit_one(); instead of making the current
+ * task to sleep, AIO throttling is performed returning -EAGAIN from
+ * sys_io_submit().
+ */
+ if (is_in_aio() && (bytes >= 0))
+ return 0;
+
+
+ iot = task_to_iothrottle(current);
+ if (unlikely(!iot))
+ return 0;
+
+
+ BUG_ON(!bdev->bd_inode || !bdev->bd_disk);
+
+ /* accounting and throttling is done only on entire block devices */
+ dev = MKDEV(MAJOR(bdev->bd_inode->i_rdev), bdev->bd_disk->first_minor);
+
+ rcu_read_lock();
+ n = iothrottle_search_node(iot, dev);
+ if (!n || !n->iorate) {
+ rcu_read_unlock();
+ return 0;
+ }
+ switch (n->strategy) {
+ case IOTHROTTLLE_LEAKY_BUCKET:
+ sleep = leaky_bucket(n, bytes);
+ break;
+ case IOTHROTTLLE_TOKEN_BUCKET:
+ sleep = token_bucket(n, bytes);
+ break;
+ default:
+ sleep = 0;
+ }
+ if (unlikely(can_sleep && sleep && (bytes >= 0))) {

```

```

+ rcu_read_unlock();
+ pr_debug("io-throttle: task %p (%s) must sleep %lu jiffies\n",
+ current, current->comm, sleep);
+ schedule_timeout_killable(sleep);
+ return 0;
+ }
+ rcu_read_unlock();
+
+ return sleep;
+}
+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..d2d8b04
--- /dev/null
+++ b/include/linux/blk-io-throttle.h
@@ -0,0 +1,41 @@
+#ifndef BLK_IO_THROTTLE_H
+#define BLK_IO_THROTTLE_H
+
+#include <linux/sched.h>
+
+#ifdef CONFIG_CGROUP_IO_THROTTLE
+extern unsigned long
+cgroup_io_throttle(struct block_device *bdev, ssize_t bytes, int can_sleep);
+
+static inline void set_in_aio(void)
+{
+ atomic_set(&current->in_aio, 1);
+}
+
+static inline void unset_in_aio(void)
+{
+ atomic_set(&current->in_aio, 0);
+}
+
+static inline int is_in_aio(void)
+{
+ return atomic_read(&current->in_aio);
+}
+#else
+static inline unsigned long
+cgroup_io_throttle(struct block_device *bdev, ssize_t bytes, int can_sleep)
+{
+ return 0;
+}
+
+static inline void set_in_aio(void) { }

```

```

+
+static inline void unset_in_aio(void) { }
+
+static inline int is_in_aio(void)
+{
+ return 0;
+}
+#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 23c02e2..198ee52 100644
--- a/include/linux/cgroup_subsys.h
+++ b/include/linux/cgroup_subsys.h
@@ -52,3 +52,9 @@ SUBSYS(memrlimit_cgroup)
 #endif

 /* */
+
+#ifdef CONFIG_CGROUP_IO_THROTTLE
+SUBSYS(iothrottle)
+#endif
+
+/* */
diff --git a/init/Kconfig b/init/Kconfig
index 162d462..a87189d 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

```

Subject: Re: [PATCH -mm 2/3] i/o bandwidth controller infrastructure
Posted by [Li Zefan](#) on Mon, 14 Jul 2008 02:25:22 GMT
[View Forum Message](#) <> [Reply to Message](#)

```
> +/* The various types of throttling algorithms */
> +enum iothrottle_strategy {
> + IOTHROTTLLE_LEAKY_BUCKET,
```

It's better to explicitly assigned 0 to IOTHROTTLLE_LEAKY_BUCKET.

```
> + IOTHROTTLLE_TOKEN_BUCKET,
> +};
```

```
> +static int iothrottle_parse_args(char *buf, size_t nbytes, dev_t *dev,
> + u64 *iorate,
> + enum iothrottle_strategy *strategy,
> + s64 *bucket_size)
> +{
> + char *p = buf;
> + int count = 0;
> + char *s[3];
> + unsigned long strategy_val;
> + int ret;
> +
> + /* split the colon-delimited input string into its elements */
> + memset(s, 0, sizeof(s));
> + while (count < ARRAY_SIZE(s)) {
> + p = memchr(p, ':', buf + nbytes - p);
> + if (!p)
> + break;
> + *p++ = '\0';
> + if (p >= buf + nbytes)
> + break;
> + s[count++] = p;
> + }
```

use strsep()

```
> +
> + /* i/o bandwidth limit */
> + if (!s[0])
> + return -EINVAL;
```

```

> + ret = strict_strtoll(s[0], 10, iorate);
> + if (ret < 0)
> + return ret;
> + if (!*iorate) {
> + /*
> +  * we're deleting a limiting rule, so just ignore the other
> +  * parameters
> +  */
> + *strategy = 0;
> + *bucket_size = 0;
> + goto out;
> + }
> + *iorate = ALIGN(*iorate, 1024);
> +
> + /* throttling strategy */
> + if (!s[1])
> + return -EINVAL;
> + ret = strict_strtoul(s[1], 10, &strategy_val);
> + if (ret < 0)
> + return ret;
> + *strategy = (enum iothrottle_strategy)strategy_val;
> + switch (*strategy) {
> + case IOTHROTTLE_LEAKY_BUCKET:
> + /* leaky bucket ignores bucket size */
> + *bucket_size = 0;
> + goto out;
> + case IOTHROTTLE_TOKEN_BUCKET:
> + break;
> + default:
> + return -EINVAL;
> + }
> +
> + /* bucket size */
> + if (!s[2])
> + return -EINVAL;
> + ret = strict_strtoll(s[2], 10, bucket_size);
> + if (ret < 0)
> + return ret;
> + if (*bucket_size < 0)
> + return -EINVAL;
> + *bucket_size = ALIGN(*bucket_size, 1024);
> +out:
> +
> + /* block device number */
> + *dev = devname2dev_t(buf);

```

why not parse dev before parse bandwidth limit ?

```

> + return *dev ? 0 : -EINVAL;
> +}
> +
> +static int iothrottle_write(struct cgroup *cgrp, struct cftype *cft,
> + const char *buffer)
> +{
> + struct iothrottle *iot;
> + struct iothrottle_node *n, *newn = NULL;
> + dev_t dev;
> + u64 iorate;
> + enum iothrottle_strategy strategy;
> + s64 bucket_size;
> + char *buf;
> + size_t nbytes = strlen(buffer);
> + int ret = 0;
> +
> + buf = kmalloc(nbytes + 1, GFP_KERNEL);
> + if (!buf)
> + return -ENOMEM;
> + memcpy(buf, buffer, nbytes + 1);
> +

```

redundant kmalloc, just use buffer, and ...

```

> + ret = iothrottle_parse_args(buf, nbytes, &dev, &iorate,
> + &strategy, &bucket_size);
> + if (ret)
> + goto out1;
> + if (iorate) {
> + newn = kmalloc(sizeof(*newn), GFP_KERNEL);
> + if (!newn) {
> + ret = -ENOMEM;
> + goto out1;
> + }
> + newn->dev = dev;
> + newn->iorate = iorate;
> + newn->strategy = strategy;
> + newn->bucket_size = bucket_size;
> + newn->timestamp = jiffies;
> + atomic_long_set(&newn->stat, 0);
> + atomic_long_set(&newn->token, 0);
> + }
> + if (!cgroup_lock_live_group(cgrp)) {
> + kfree(newn);
> + ret = -ENODEV;
> + goto out1;
> + }
> + iot = cgroup_to_iothrottle(cgrp);

```

```

> +
> + spin_lock(&iot->lock);
> + if (!iorate) {
> + /* Delete a block device limiting rule */
> + n = iothrottle_delete_node(iot, dev);
> + goto out2;
> + }
> + n = iothrottle_search_node(iot, dev);
> + if (n) {
> + /* Update a block device limiting rule */
> + iothrottle_replace_node(iot, n, newn);
> + goto out2;
> + }
> + /* Add a new block device limiting rule */
> + iothrottle_insert_node(iot, newn);
> +out2:
> + spin_unlock(&iot->lock);
> + cgroup_unlock();
> + if (n) {
> + synchronize_rcu();
> + kfree(n);
> + }
> +out1:
> + kfree(buf);
> + return ret;
> +}
> +
> +static struct cftype files[] = {
> + {
> + .name = "bandwidth",
> + .read_seq_string = iothrottle_read,
> + .write_string = iothrottle_write,

```

and you should specify `.max_write_len = XXX` unless `XXX <= 64`.
You use 1024 in v4.

```

> + },
> +};
> +

```

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

Subject: Re: [PATCH -mm 2/3] i/o bandwidth controller infrastructure
Posted by [Andrea Righi](#) on Mon, 14 Jul 2008 21:26:51 GMT

Li Zefan wrote:

```
>> +/* The various types of throttling algorithms */
>> +enum iothrottle_strategy {
>> + IOTHROTTLE_LEAKY_BUCKET,
>
> It's better to explicitly assigned 0 to IOTHROTTLE_LEAKY_BUCKET.
```

OK.

```
>
>> + IOTHROTTLE_TOKEN_BUCKET,
>> +};
>
>> +static int iothrottle_parse_args(char *buf, size_t nbytes, dev_t *dev,
>> +   u64 *iorate,
>> +   enum iothrottle_strategy *strategy,
>> +   s64 *bucket_size)
>> +{
>> + char *p = buf;
>> + int count = 0;
>> + char *s[3];
>> + unsigned long strategy_val;
>> + int ret;
>> +
>> + /* split the colon-delimited input string into its elements */
>> + memset(s, 0, sizeof(s));
>> + while (count < ARRAY_SIZE(s)) {
>> + p = memchr(p, ':', buf + nbytes - p);
>> + if (!p)
>> + break;
>> + *p++ = '\0';
>> + if (p >= buf + nbytes)
>> + break;
>> + s[count++] = p;
>> + }
>
> use strsep()
```

OK.

```
>
>> +
>> + /* i/o bandwidth limit */
>> + if (!s[0])
>> + return -EINVAL;
>> + ret = strict_strtoull(s[0], 10, iorate);
>> + if (ret < 0)
```

```

>> + return ret;
>> + if (!*iorate) {
>> + /*
>> +  * we're deleting a limiting rule, so just ignore the other
>> +  * parameters
>> +  */
>> + *strategy = 0;
>> + *bucket_size = 0;
>> + goto out;
>> + }
>> + *iorate = ALIGN(*iorate, 1024);
>> +
>> + /* throttling strategy */
>> + if (!s[1])
>> + return -EINVAL;
>> + ret = strict_strtoul(s[1], 10, &strategy_val);
>> + if (ret < 0)
>> + return ret;
>> + *strategy = (enum iothrottle_strategy)strategy_val;
>> + switch (*strategy) {
>> + case IOTHROTTLE_LEAKY_BUCKET:
>> + /* leaky bucket ignores bucket size */
>> + *bucket_size = 0;
>> + goto out;
>> + case IOTHROTTLE_TOKEN_BUCKET:
>> + break;
>> + default:
>> + return -EINVAL;
>> + }
>> +
>> + /* bucket size */
>> + if (!s[2])
>> + return -EINVAL;
>> + ret = strict_strtoll(s[2], 10, bucket_size);
>> + if (ret < 0)
>> + return ret;
>> + if (*bucket_size < 0)
>> + return -EINVAL;
>> + *bucket_size = ALIGN(*bucket_size, 1024);
>> +out:
>> +
>> + /* block device number */
>> + *dev = devname2dev_t(buf);
>
> why not parse dev before parse bandwidth limit ?

```

Well... due to the filesystem lookup in devname2dev_t() this option is the most expensive to be parsed. For this reason I put it at the end, so

if any other parameter is wrong we can skip the lookup_bdev(). Does it make sense?

```
>
>> + return *dev ? 0 : -EINVAL;
>> +}
>> +
>> +static int iothrottle_write(struct cgroup *cgrp, struct cftype *cft,
>> + const char *buffer)
>> +{
>> + struct iothrottle *iot;
>> + struct iothrottle_node *n, *newn = NULL;
>> + dev_t dev;
>> + u64 iorate;
>> + enum iothrottle_strategy strategy;
>> + s64 bucket_size;
>> + char *buf;
>> + size_t nbytes = strlen(buffer);
>> + int ret = 0;
>> +
>> + buf = kmalloc(nbytes + 1, GFP_KERNEL);
>> + if (!buf)
>> + return -ENOMEM;
>> + memcpy(buf, buffer, nbytes + 1);
>> +
>
> redundant kmalloc, just use buffer, and ...
```

uhmm... I would apply strsep() directly to buffer in this way, that is a const char *.

```
>
>> + ret = iothrottle_parse_args(buf, nbytes, &dev, &iorate,
>> + &strategy, &bucket_size);
>> + if (ret)
>> + goto out1;
>> + if (iorate) {
>> + newn = kmalloc(sizeof(*newn), GFP_KERNEL);
>> + if (!newn) {
>> + ret = -ENOMEM;
>> + goto out1;
>> + }
>> + newn->dev = dev;
>> + newn->iorate = iorate;
>> + newn->strategy = strategy;
>> + newn->bucket_size = bucket_size;
>> + newn->timestamp = jiffies;
```

```

>> + atomic_long_set(&newn->stat, 0);
>> + atomic_long_set(&newn->token, 0);
>> + }
>> + if (!cgroup_lock_live_group(cgrp)) {
>> + kfree(newn);
>> + ret = -ENODEV;
>> + goto out1;
>> + }
>> + iot = cgroup_to_iothrottle(cgrp);
>> +
>> + spin_lock(&iot->lock);
>> + if (!iorate) {
>> + /* Delete a block device limiting rule */
>> + n = iothrottle_delete_node(iot, dev);
>> + goto out2;
>> + }
>> + n = iothrottle_search_node(iot, dev);
>> + if (n) {
>> + /* Update a block device limiting rule */
>> + iothrottle_replace_node(iot, n, newn);
>> + goto out2;
>> + }
>> + /* Add a new block device limiting rule */
>> + iothrottle_insert_node(iot, newn);
>> +out2:
>> + spin_unlock(&iot->lock);
>> + cgroup_unlock();
>> + if (n) {
>> + synchronize_rcu();
>> + kfree(n);
>> + }
>> +out1:
>> + kfree(buf);
>> + return ret;
>> +}
>> +
>> +static struct cftype files[] = {
>> + {
>> + .name = "bandwidth",
>> + .read_seq_string = iothrottle_read,
>> + .write_string = iothrottle_write,
>> +
>
> and you should specify .max_write_len = XXX unless XXX <= 64.
> You use 1024 in v4.

```

OK. Anyway, probably something like 256 would be enough.

Thanks again for the detailed revision Li! I'll include your fixes in

a new patchset version.

-Andrea

Containers mailing list

Containers@lists.linux-foundation.org

<https://lists.linux-foundation.org/mailman/listinfo/containers>
