Subject: [PATCH 2/3] i/o bandwidth controller infrastructure
Posted by Andrea Righi on Fri, 04 Jul 2008 13:58:47 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 | 529 ++
 include/linux/blk-io-throttle.h | 14 +
 include/linux/cgroup_subsys.h | 6 +
 init/Kconfig | 10 +
 5 files changed, 561 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..caf740a
--- /dev/null
+++ b/block/blk-io-throttle.c
@@ -0,0 +1,529 @@
+/*
+ * 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/spinlock.h>
+#include <linux/uaccess.h>
+#include <linux/blk-io-throttle.h>
+
+/**
+ * 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 (0 = leaky bucket, 1 = token bucket)
+ * @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;
+ long 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 *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);
+}
+
+/*
+ * Note: called with rcu_read_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 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;
+}
+
+/*
+ * 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_rcu(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;
+
+  BUG_ON(!n->dev);
+  delta = jiffies_to_msecs((long)jiffies - (long)n->timestamp);
+  s += scnprintf(buffer + s, nbytes - s,
```

```
+    "%u %u %llu %li %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();
+ ret = simple_read_from_buffer(userbuf, nbytes, ppos, buffer, s);
+out:
+ cgroup_unlock();
+ kfree(buffer);
+ return ret;
+}
+
+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 the following syntax:
+ *
+ * device:bw-limit:strategy:bucket-size
+ */
+static int iothrottle_parse_args(char *buf, size_t nbytes,
+     dev_t *dev, u64 *iorate,
+     long *strategy, s64 *bucket_size)
+{
+ char *ioratep, *strategyp, *bucket_sizep;
+ int ret;
```

```
+
+ ioratep = memchr(buf, ':', nbytes);
+ if (!ioratep)
+  return -EINVAL;
+ *ioratep++ = '\0';
+
+ strategyp = memchr(ioratep, ':', buf + nbytes - ioratep);
+ if (!strategyp)
+  return -EINVAL;
+ *strategyp++ = '\0';
+
+ bucket_sizep = memchr(strategyp, ':', buf + nbytes - strategyp);
+ if (!bucket_sizep)
+  return -EINVAL;
+ *bucket_sizep++ = '\0';
+
+ /* i/o bandwidth limit (0 to delete a limiting rule) */
+ ret = strict_strtoull(ioratep, 10, iorate);
+ if (ret < 0)
+  return ret;
+ *iorate = ALIGN(*iorate, 1024);
+
+ /* throttling strategy */
+ ret = strict_strtol(strategyp, 10, strategy);
+ if (ret < 0)
+  return ret;
+
+ /* bucket size */
+ ret = strict_strtoll(bucket_sizep, 10, bucket_size);
+ if (ret < 0)
+  return ret;
+ if (*bucket_size < 0)
+  return -EINVAL;
+ *bucket_size = ALIGN(*bucket_size, 1024);
+
+ /* block device number */
+ *dev = devname2dev_t(buf);
+ if (!*dev)
+  return -EINVAL;
+
+ 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, *newn = NULL;
+ char *buffer, *s;
+ dev_t dev;
+ u64 iorate;
+ long strategy;
+ s64 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;
+
+ ret = strncpy_from_user(buffer, userbuf, nbytes);
+ if (ret < 0)
+  goto out1;
+ buffer[nbytes] = '\0';
+ s = strstrip(buffer);
+
+ ret = iothrottle_parse_args(s, 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);
+ }
+
+ 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, newn);
+  goto out3;
+ }
+ /* Add a new block device limiting rule */
+ iothrottle_insert_node(iot, newn);
+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,
+};
+
+/*
+ * Note: called with rcu_read_lock() held.
+ */
+static unsigned 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;
+
+ /*
+  * 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.
+  */
+ t = usecs_to_jiffies(atomic_long_read(&n->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, size_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.
+ **/
+void cgroup_io_throttle(struct block_device *bdev, size_t bytes, int can_sleep)
+{
+ struct iothrottle *iot;
+ struct iothrottle_node *n;
+ dev_t dev;
+ unsigned long sleep;
+
```

```
+ if (unlikely(!bdev))
+  return;
+
+ iot = task_to_iothrottle(current);
+ if (unlikely(!iot))
+  return;
+
+ BUG_ON(!bdev->bd_inode || !bdev->bd_disk);
+ 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;
+ }
+ switch (n->strategy) {
+ case 0:
+  sleep = leaky_bucket(n, bytes);
+  break;
+ case 1:
+  sleep = token_bucket(n, bytes);
+  break;
+ default:
+  sleep = 0;
+ }
+ if (unlikely(can_sleep && sleep)) {
+  rcu_read_unlock();
+  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..0fe7430
--- /dev/null
+++ b/include/linux/blk-io-throttle.h
@@ -0,0 +1,14 @@
+#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, int can_sleep);
```

```
+#else
+static inline void
+cgroup_io_throttle(struct block_device *bdev, size_t bytes, int can_sleep)
+{
+}
+#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

Subject: Re: [PATCH 2/3] i/o bandwidth controller infrastructure
Posted by Li Zefan on Sat, 05 Jul 2008 02:07:35 GMT
View Forum Message <> Reply to Message

> +/**
> + * 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 (0 = leaky bucket, 1 = token bucket)

use enum or define

> + * @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;
> + long 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 *cont)
> +{
```

cgrp is a preferable name to 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);
> +}
> +
> +/*
> + * Note: called with rcu_read_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 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;
```

you just return back 'old', so the return value is useless

```
> +}
> +
> +/*
> + * 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_rcu(n, &iot->list, node)
```

list_for_each_entry()

```
> +  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)
```

use read_seq_string can simplify this function:
 .read_seq_string = iothrottle_read

```
> +{
> + 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;
> +
> +  BUG_ON(!n->dev);
> +  delta = jiffies_to_msecs((long)jiffies - (long)n->timestamp);
> +  s += scnprintf(buffer + s, nbytes - s,
```

```
> +    "%u %u %llu %li %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();
> + ret = simple_read_from_buffer(userbuf, nbytes, ppos, buffer, s);
> +out:
> + cgroup_unlock();
> + kfree(buffer);
> + return ret;
> +}
> +
> +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 the following syntax:
> + *
> + * device:bw-limit:strategy:bucket-size
> + */
```

why not support these syntax:
device:0
device:bw-limit:0

```
> +static int iothrottle_parse_args(char *buf, size_t nbytes,
```

```
> +     dev_t *dev, u64 *iorate,
> +     long *strategy, s64 *bucket_size)
> +{
> + char *ioratep, *strategyp, *bucket_sizep;
> + int ret;
> +
> + ioratep = memchr(buf, ':', nbytes);
> + if (!ioratep)
> +  return -EINVAL;
> + *ioratep++ = '\0';
> +
> + strategyp = memchr(ioratep, ':', buf + nbytes - ioratep);
> + if (!strategyp)
> +  return -EINVAL;
> + *strategyp++ = '\0';
> +
> + bucket_sizep = memchr(strategyp, ':', buf + nbytes - strategyp);
> + if (!bucket_sizep)
> +  return -EINVAL;
> + *bucket_sizep++ = '\0';
> +
> + /* i/o bandwidth limit (0 to delete a limiting rule) */
> + ret = strict_strtoull(ioratep, 10, iorate);
> + if (ret < 0)
> +  return ret;
> + *iorate = ALIGN(*iorate, 1024);
> +
> + /* throttling strategy */
> + ret = strict_strtol(strategyp, 10, strategy);
> + if (ret < 0)
> +  return ret;
> +
> + /* bucket size */
> + ret = strict_strtoll(bucket_sizep, 10, bucket_size);
> + if (ret < 0)
> +  return ret;
> + if (*bucket_size < 0)
> +  return -EINVAL;
> + *bucket_size = ALIGN(*bucket_size, 1024);
> +
> + /* block device number */
> + *dev = devname2dev_t(buf);
> + if (!*dev)
> +  return -EINVAL;
> +
> + 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)
> +{
```

you can use write_string (currently in -mm) to simplify this funciton:
 .write_string = iothrottle_write

```
> + struct iothrottle *iot;
> + struct iothrottle_node *n, *newn = NULL;
> + char *buffer, *s;
> + dev_t dev;
> + u64 iorate;
> + long strategy;
> + s64 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;
> +
> + ret = strncpy_from_user(buffer, userbuf, nbytes);
> + if (ret < 0)
> +  goto out1;
> + buffer[nbytes] = '\0';
> + s = strstrip(buffer);
> +
> + ret = iothrottle_parse_args(s, 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);
> + }
> +
> + 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, newn);
> +  goto out3;
> + }
> + /* Add a new block device limiting rule */
> + iothrottle_insert_node(iot, newn);
> +out3:
> + ret = nbytes;
> + spin_unlock(&iot->lock);
> + if (n) {
> +  synchronize_rcu();
> +  kfree(n);
> + }
> +out2:
> + cgroup_unlock();
> +out1:
> + kfree(buffer);
> + return ret;
> +}
```

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

Li Zefan wrote:
>> +/**
>> + * 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 (0 = leaky bucket, 1 = token bucket)
>
> use enum or define
>
>> + * @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;
>> + long 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 *cont)
>> +{
>
> cgrp is a preferable name to 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);
>> +}
>> +
>> +/*
>> + * Note: called with rcu_read_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 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;
```

>
> you just return back 'old', so the return value is useless
>
>> +}
>> +
>> +/*
>> + * 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_rcu(n, &iot->list, node)
>
> list_for_each_entry()
>
>> + 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)
>
> use read_seq_string can simplify this function:
>  .read_seq_string = iothrottle_read
>
>> +{
>> + 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;
>> +
>> +  BUG_ON(!n->dev);
>> +  delta = jiffies_to_msecs((long)jiffies - (long)n->timestamp);
>> +  s += scnprintf(buffer + s, nbytes - s,
>> +    "%u %u %llu %li %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();
>> + ret = simple_read_from_buffer(userbuf, nbytes, ppos, buffer, s);
>> +out:
>> + cgroup_unlock();
>> + kfree(buffer);
>> + return ret;
>> +}
>> +
>> +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 the following syntax:
>> + *
>> + * device:bw-limit:strategy:bucket-size
>> + */
>
> why not support these syntax:
> device:0
> device:bw-limit:0
>
>> +static int iothrottle_parse_args(char *buf, size_t nbytes,
>> +     dev_t *dev, u64 *iorate,
>> +     long *strategy, s64 *bucket_size)
>> +{
>> + char *ioratep, *strategyp, *bucket_sizep;
>> + int ret;
```

```
>> +
>> + ioratep = memchr(buf, ':', nbytes);
>> + if (!ioratep)
>> +  return -EINVAL;
>> + *ioratep++ = '\0';
>> +
>> + strategyp = memchr(ioratep, ':', buf + nbytes - ioratep);
>> + if (!strategyp)
>> +  return -EINVAL;
>> + *strategyp++ = '\0';
>> +
>> + bucket_sizep = memchr(strategyp, ':', buf + nbytes - strategyp);
>> + if (!bucket_sizep)
>> +  return -EINVAL;
>> + *bucket_sizep++ = '\0';
>> +
>> + /* i/o bandwidth limit (0 to delete a limiting rule) */
>> + ret = strict_strtoull(ioratep, 10, iorate);
>> + if (ret < 0)
>> +  return ret;
>> + *iorate = ALIGN(*iorate, 1024);
>> +
>> + /* throttling strategy */
>> + ret = strict_strtol(strategyp, 10, strategy);
>> + if (ret < 0)
>> +  return ret;
>> +
>> + /* bucket size */
>> + ret = strict_strtoll(bucket_sizep, 10, bucket_size);
>> + if (ret < 0)
>> +  return ret;
>> + if (*bucket_size < 0)
>> +  return -EINVAL;
>> + *bucket_size = ALIGN(*bucket_size, 1024);
>> +
>> + /* block device number */
>> + *dev = devname2dev_t(buf);
>> + if (!*dev)
>> +  return -EINVAL;
>> +
>> + 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)
>> +{
>
```

> you can use write_string (currently in -mm) to simplify this funciton:
> .write_string = iothrottle_write
>
>> + struct iothrottle *iot;
>> + struct iothrottle_node *n, *newn = NULL;
>> + char *buffer, *s;
>> + dev_t dev;
>> + u64 iorate;
>> + long strategy;
>> + s64 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;
>> +
>> + ret = strncpy_from_user(buffer, userbuf, nbytes);
>> + if (ret < 0)
>> + goto out1;
>> + buffer[nbytes] = '\0';
>> + s = strstrip(buffer);
>> +
>> + ret = iothrottle_parse_args(s, 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);
>> + }

```
>> +
>> + 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, newn);
>> +  goto out3;
>> + }
>> + /* Add a new block device limiting rule */
>> + iothrottle_insert_node(iot, newn);
>> +out3:
>> + ret = nbytes;
>> + spin_unlock(&iot->lock);
>> + if (n) {
>> +  synchronize_rcu();
>> +  kfree(n);
>> + }
>> +out2:
>> + cgroup_unlock();
>> +out1:
>> + kfree(buffer);
>> + return ret;
>> +}
```

Li, thanks for reviewing. I agree on everything and I'll apply all your suggestions in the next patchset version.

-Andrea