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