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