Subject: [PATCH 1/2] dm-band: The I/O bandwidth controller: Source code patch
Posted by Ryo Tsuruta on Wed, 23 Jan 2008 12:56:32 GMT
View Forum Message <> Reply to Message

Here is the patch of dm-band.

Based on 2.6.23.14
Signed-off-by: Ryo Tsuruta <ryov@valinux.co.jp>
Signed-off-by: Hirokazu Takahashi <taka@valinux.co.jp>

```
diff -uprN linux-2.6.23.14.orig/drivers/md/Kconfig linux-2.6.23.14/drivers/md/Kconfig
--- linux-2.6.23.14.orig/drivers/md/Kconfig 2008-01-15 05:49:56.000000000 +0900
+++ linux-2.6.23.14/drivers/md/Kconfig 2008-01-21 16:09:41.000000000 +0900
@@ -276,4 +276,13 @@ config DM_DELAY

  If unsure, say N.

+config DM_BAND
+ tristate "I/O band width control "
+ depends on BLK_DEV_DM
+ ---help---
+ Any processes or cgroups can use the same storage
+ with its band-width fairly shared.
+
+ If unsure, say N.
+
 endif # MD
diff -uprN linux-2.6.23.14.orig/drivers/md/Makefile linux-2.6.23.14/drivers/md/Makefile
--- linux-2.6.23.14.orig/drivers/md/Makefile 2008-01-15 05:49:56.000000000 +0900
+++ linux-2.6.23.14/drivers/md/Makefile 2008-01-21 20:45:03.000000000 +0900
@@ -8,6 +8,7 @@ dm-multipath-objs := dm-hw-handler.o dm-
 dm-snapshot-objs := dm-snap.o dm-exception-store.o
 dm-mirror-objs := dm-log.o dm-raid1.o
 dm-rdac-objs := dm-mpath-rdac.o
+dm-band-objs := dm-bandctl.o dm-band-policy.o dm-band-type.o
 md-mod-objs     := md.o bitmap.o
 raid456-objs := raid5.o raid6algos.o raid6recov.o raid6tables.o \
      raid6int1.o raid6int2.o raid6int4.o \
@@ -39,6 +40,7 @@ obj-$(CONFIG_DM_MULTIPATH_RDAC) += dm-rd
 obj-$(CONFIG_DM_SNAPSHOT) += dm-snapshot.o
 obj-$(CONFIG_DM_MIRROR)  += dm-mirror.o
 obj-$(CONFIG_DM_ZERO)  += dm-zero.o
+obj-$(CONFIG_DM_BAND)  += dm-band.o

 quiet_cmd_unroll = UNROLL  $@
       cmd_unroll = $(PERL) $(srctree)/$(src)/unroll.pl $(UNROLL) \
diff -uprN linux-2.6.23.14.orig/drivers/md/dm-band-policy.c
linux-2.6.23.14/drivers/md/dm-band-policy.c
```

```
--- linux-2.6.23.14.orig/drivers/md/dm-band-policy.c 1970-01-01 09:00:00.000000000 +0900
+++ linux-2.6.23.14/drivers/md/dm-band-policy.c 2008-01-21 20:31:14.000000000 +0900
@@ -0,0 +1,185 @@
+/*
+ * Copyright (C) 2008 VA Linux Systems Japan K.K.
+ *
+ *  I/O bandwidth control
+ *
+ * This file is released under the GPL.
+ */
+#include <linux/bio.h>
+#include <linux/workqueue.h>
+#include "dm.h"
+#include "dm-bio-list.h"
+#include "dm-band.h"
+
+/*
+ * The following functiotons determine when and which BIOs should
+ * be submitted to control the I/O flow.
+ * It is possible to add a new I/O scheduling policy with it.
+ */
+
+
+/*
+ * Functions for weight balancing policy.
+ */
+#define DEFAULT_WEIGHT 100
+#define DEFAULT_TOKENBASE 2048
+#define BAND_IOPRIO_BASE 100
+
+static int proceed_global_epoch(struct banddevice *bs)
+{
+ bs->g_epoch++;
+#if 0 /* this will also work correct */
+ if (bs->g_blocked)
+  queue_work(bs->g_band_wq, &bs->g_conductor);
+ return 0;
+#endif
+ dprintk(KERN_ERR "proceed_epoch %d --> %d\n",
+       bs->g_epoch-1, bs->g_epoch);
+ return 1;
+}
+
+static inline int proceed_epoch(struct bandgroup *bw)
+{
+ struct banddevice *bs = bw->c_shared;
+
+ if (bw->c_my_epoch != bs->g_epoch) {
```

```
+  bw->c_my_epoch = bs->g_epoch;
+  return 1;
+ }
+ return 0;
+}
+
+static inline int iopriority(struct bandgroup *bw)
+{
+ struct banddevice *bs = bw->c_shared;
+ int iopri;
+
+ iopri = bw->c_token*BAND_IOPRIO_BASE/bw->c_token_init_value + 1;
+ if (bw->c_my_epoch != bs->g_epoch)
+  iopri += BAND_IOPRIO_BASE;
+ if (bw->c_going_down)
+  iopri += BAND_IOPRIO_BASE*2;
+
+ return iopri;
+}
+
+static int is_token_left(struct bandgroup *bw)
+{
+ if (bw->c_token > 0)
+  return iopriority(bw);
+
+ if (proceed_epoch(bw) || bw->c_going_down) {
+  bw->c_token = bw->c_token_init_value;
+  dprintk(KERN_ERR "refill token: bw:%p token:%d\n",
+       bw, bw->c_token);
+  return iopriority(bw);
+ }
+ return 0;
+}
+
+static void prepare_token(struct bandgroup *bw, struct bio *bio)
+{
+ bw->c_token--;
+}
+
+static void set_weight(struct bandgroup *bw, int new)
+{
+ struct banddevice *bs = bw->c_shared;
+ struct bandgroup *p;
+
+ bs->g_weight_total += (new - bw->c_weight);
+ bw->c_weight = new;
+
+ list_for_each_entry(p, &bs->g_brothers, c_list) {
```

```
+  /* Fixme: it might overflow */
+  p->c_token = p->c_token_init_value =
+      bs->g_token_base*p->c_weight/bs->g_weight_total + 1;
+ }
+}
+
+static int policy_weight_ctr(struct bandgroup *bw)
+{
+ struct banddevice *bs = bw->c_shared;
+
+ bw->c_my_epoch = bs->g_epoch;
+ bw->c_weight = 0;
+ set_weight(bw, DEFAULT_WEIGHT);
+ return 0;
+}
+
+static void policy_weight_dtr(struct bandgroup *bw)
+{
+ set_weight(bw, 0);
+}
+
+static int policy_weight_param(struct bandgroup *bw, char *cmd, char *value)
+{
+ struct banddevice *bs = bw->c_shared;
+ int val = simple_strtol(value, NULL, 0);
+ int r = 0;
+
+ if (!strcmp(cmd, "weight")) {
+  if (val > 0)
+   set_weight(bw, val);
+  else
+   r = -EINVAL;
+ } else if (!strcmp(cmd, "token")) {
+  if (val > 0) {
+   bs->g_token_base = val;
+   set_weight(bw, bw->c_weight);
+  } else
+   r = -EINVAL;
+ } else {
+  r = -EINVAL;
+ }
+ return r;
+}
+
+/*
+ *  <Method>      <description>
+ * g_can_submit   : To determine whether a given group has a right to
+ *                  submit BIOs.
```

```
+ *       The larger return value the higher priority to submit.
+ *       Zero means it has no right.
+ * g_prepare_bio  : Called right before submitting each BIO.
+ * g_restart_bios : Called when there exist some BIOs blocked but none of them
+ *       can't be submitted now.
+ *      This method have to do something to restart to submit BIOs.
+ *       Returns 0 if it has become able to submit them now.
+ *       Otherwise, returns 1 and this policy module has to restart
+ *      sumitting BIOs by itself later on.
+ * g_hold_bio     : To hold a given BIO until it is submitted.
+ *       The default function is used when this method is undefined.
+ * g_pop_bio      : To select and get the best BIO to submit.
+ * g_group_ctr    : To initalize the policy own members of struct bandgroup.
+ * g_group_dtr    : Called when struct bandgroup is removed.
+ * g_set_param    : To update the policy own date.
+ *       The parameters can be passed through "dmsetup message"
+ *                 command.
+ */
+static void policy_weight_init(struct banddevice *bs)
+{
+ bs->g_can_submit = is_token_left;
+ bs->g_prepare_bio = prepare_token;
+ bs->g_restart_bios = proceed_global_epoch;
+ bs->g_group_ctr = policy_weight_ctr;
+ bs->g_group_dtr = policy_weight_dtr;
+ bs->g_set_param = policy_weight_param;
+
+ bs->g_token_base = DEFAULT_TOKENBASE;
+ bs->g_epoch = 0;
+ bs->g_weight_total = 0;
+}
+/* weight balancing policy. --- End --- */
+
+
+static void policy_default_init(struct banddevice *bs) /*XXX*/
+{
+ policy_weight_init(bs); /* temp */
+}
+
+struct policy_type band_policy_type[] = {
+ {"default", policy_default_init},
+ {"weight", policy_weight_init},
+ {NULL,     policy_default_init}
+};
diff -uprN linux-2.6.23.14.orig/drivers/md/dm-band-type.c
linux-2.6.23.14/drivers/md/dm-band-type.c
--- linux-2.6.23.14.orig/drivers/md/dm-band-type.c 1970-01-01 09:00:00.000000000 +0900
+++ linux-2.6.23.14/drivers/md/dm-band-type.c 2008-01-21 20:27:15.000000000 +0900
```

```
@@ -0,0 +1,69 @@
+/*
+ * Copyright (C) 2008 VA Linux Systems Japan K.K.
+ *
+ *  I/O bandwidth control
+ *
+ * This file is released under the GPL.
+ */
+#include <linux/bio.h>
+#include "dm.h"
+#include "dm-bio-list.h"
+#include "dm-band.h"
+
+/*
+ * Any I/O bandwidth can be divided into several bandwidth groups, each of which
+ * has its own unique ID. The following functions are called to determine
+ * which group a given BIO belongs to and return the ID of the group.
+ */
+
+/* ToDo: unsigned long value would be better for group ID */
+
+static int band_process_id(struct bio *bio)
+{
+ /*
+  * This function will work for KVM and Xen.
+  */
+ return (int)current->tgid;
+}
+
+static int band_process_group(struct bio *bio)
+{
+ return (int)process_group(current);
+}
+
+static int band_uid(struct bio *bio)
+{
+ return (int)current->uid;
+}
+
+static int band_cpuset(struct bio *bio)
+{
+ return 0; /* not implemented yet */
+}
+
+static int band_node(struct bio *bio)
+{
+ return 0; /* not implemented yet */
+}
```

```
+
+static int band_cgroup(struct bio *bio)
+{
+  /*
+   * This function should return the ID of the cgroup which issued "bio".
+   * The ID of the cgroup which the current process belongs to won't be
+   * suitable ID for this purpose, since some BIOs will be handled by kernel
+   * threads like aio or pdflush on behalf of the process requesting the BIOs.
+   */
+ return 0; /* not implemented yet */
+}
+
+struct group_type band_group_type[] = {
+ {"none",   NULL},
+ {"pgrp",   band_process_group},
+ {"pid",    band_process_id},
+ {"node",   band_node},
+ {"cpuset", band_cpuset},
+ {"cgroup", band_cgroup},
+ {"user",   band_uid},
+ {NULL,     NULL}
+};
diff -uprN linux-2.6.23.14.orig/drivers/md/dm-band.h linux-2.6.23.14/drivers/md/dm-band.h
--- linux-2.6.23.14.orig/drivers/md/dm-band.h 1970-01-01 09:00:00.000000000 +0900
+++ linux-2.6.23.14/drivers/md/dm-band.h 2008-01-21 20:20:54.000000000 +0900
@@ -0,0 +1,99 @@
+/*
+ * Copyright (C) 2008 VA Linux Systems Japan K.K.
+ *
+ *  I/O bandwidth control
+ *
+ * This file is released under the GPL.
+ */
+
+#define DEFAULT_IO_THROTTLE 4
+#define DEFAULT_IO_LIMIT 128
+#define BAND_NAME_MAX 31
+#define BAND_ID_ANY (-1)
+
+struct bandgroup;
+
+struct banddevice {
+ struct list_head g_brothers;
+ struct work_struct g_conductor;
+ struct workqueue_struct *g_band_wq;
+ int g_io_throttle;
+ int g_io_limit;
+ int g_plug_bio;
```

```
+ int g_issued;
+ int g_blocked;
+ spinlock_t g_lock;
+
+ int g_devgroup;
+ int g_ref;  /* just for debugging */
+ struct list_head g_list;
+ int g_flags; /* rfu */
+ char g_name[BAND_NAME_MAX + 1]; /* rfu */
+
+ /* policy dependent */
+ int (*g_can_submit)(struct bandgroup *);
+ void (*g_prepare_bio)(struct bandgroup *, struct bio *);
+ int (*g_restart_bios)(struct banddevice *);
+ void (*g_hold_bio)(struct bandgroup *, struct bio *);
+ struct bio * (*g_pop_bio)(struct bandgroup *);
+ int (*g_group_ctr)(struct bandgroup *);
+ void (*g_group_dtr)(struct bandgroup *);
+ int (*g_set_param)(struct bandgroup *, char *cmd, char *value);
+
+ /* members for weight balancing policy */
+ int g_epoch;
+ int g_weight_total;
+ int g_token_base;
+
+};
+
+struct bandgroup_stat {
+ unsigned long sectors;
+ unsigned long immediate;
+ unsigned long deferred;
+};
+
+struct bandgroup {
+ struct list_head c_list;
+ struct banddevice *c_shared;
+ struct dm_dev *c_dev;
+ int c_going_down;
+ struct bio_list c_blocked_bios;
+ struct list_head c_children;
+ int c_id; /* should be unsigned long or unsigned long long */
+ char c_name[BAND_NAME_MAX + 1]; /* rfu */
+ int c_issued;
+ int c_blocked;
+ struct bandgroup_stat c_stat[2]; /* hold rd/wr status */
+
+ /* type dependent */
+ int (*c_getid)(struct bio *);
```

```
+
+ /* members for weight balancing policy */
+ int c_weight;
+ int c_my_epoch;
+ int c_token;
+ int c_token_init_value;
+
+ /* rfu */
+ /* struct bio_list c_ordered_tag_bios; */
+};
+
+struct policy_type {
+ const char *p_name;
+ void   (*p_policy_init)(struct banddevice *);
+};
+
+extern struct policy_type band_policy_type[];
+
+struct group_type {
+ const char *t_name;
+ int   (*t_getid)(struct bio *);
+};
+
+extern struct group_type band_group_type[];
+
+/* Just for debugging */
+extern int band_debug;
+#define dprintk(format, a...) \
+ if (band_debug > 0) band_debug--, printk(format, ##a)
diff -uprN linux-2.6.23.14.orig/drivers/md/dm-bandctl.c linux-2.6.23.14/drivers/md/dm-bandctl.c
--- linux-2.6.23.14.orig/drivers/md/dm-bandctl.c 1970-01-01 09:00:00.000000000 +0900
+++ linux-2.6.23.14/drivers/md/dm-bandctl.c 2008-01-23 20:48:19.000000000 +0900
@@ -0,0 +1,833 @@
+/*
+ * Copyright (C) 2008 VA Linux Systems Japan K.K.
+ * Authors: Hirokazu Takahashi <taka@valinux.co.jp>
+ *          Ryo Tsuruta <ryov@valinux.co.jp>
+ *
+ *  I/O bandwidth control
+ *
+ * This file is released under the GPL.
+ */
+#include <linux/module.h>
+#include <linux/init.h>
+#include <linux/bio.h>
+#include <linux/slab.h>
+#include <linux/workqueue.h>
+#include "dm.h"
```

```
+#include "dm-bio-list.h"
+#include "dm-band.h"
+
+#define DM_MSG_PREFIX "band"
+
+static LIST_HEAD(banddevice_list);
+static DEFINE_SPINLOCK(banddevicelist_lock);  /* to protect banddevice_list */
+
+static void band_conduct(struct work_struct *);
+static void band_hold_bio(struct bandgroup *, struct bio *);
+static struct bio *band_pop_bio(struct bandgroup *);
+
+int band_debug = 0; /* just for debugging */
+
+static void policy_init(struct banddevice *bs, char *name)
+{
+ struct policy_type *p;
+ for (p = band_policy_type; (p->p_name); p++) {
+  if (!strcmp(name, p->p_name))
+   break;
+ }
+
+ p->p_policy_init(bs);
+ if (!bs->g_hold_bio)
+  bs->g_hold_bio = band_hold_bio;
+ if (!bs->g_pop_bio)
+  bs->g_pop_bio = band_pop_bio;
+}
+
+static struct banddevice *alloc_banddevice(int devgroup_id, char *name,
+    int io_throttle, int io_limit)
+{
+ struct banddevice *bs = NULL;
+ struct banddevice *p;
+ struct banddevice *new;
+ unsigned long flags;
+
+ new = kzalloc(sizeof(struct banddevice), GFP_KERNEL);
+ if (!new)
+  goto try_to_find;
+
+ /*
+  * Prepare its own workqueue as generic_make_request() may potentially
+  * block the workqueue when submitting BIOs.
+  */
+ new->g_band_wq = create_workqueue("kband");
+ if (!new->g_band_wq) {
+  kfree(new);
```

```
+  new = NULL;
+  goto try_to_find;
+ }
+
+ INIT_WORK(&new->g_conductor, band_conduct);
+ INIT_LIST_HEAD(&new->g_brothers);
+ INIT_LIST_HEAD(&new->g_list);
+ spin_lock_init(&new->g_lock);
+ new->g_devgroup = devgroup_id;
+ new->g_io_throttle = io_throttle;
+ new->g_io_limit = io_limit;
+ new->g_plug_bio = 0;
+ new->g_issued = 0;
+ new->g_blocked = 0;
+ new->g_ref = 0;
+ new->g_flags = 0;
+ memset(new->g_name, 0, sizeof(new->g_name));
+ new->g_hold_bio = NULL;
+ new->g_pop_bio = NULL;
+
+try_to_find:
+ spin_lock_irqsave(&banddevicelist_lock, flags);
+ list_for_each_entry(p, &banddevice_list, g_list) {
+  if (p->g_devgroup == devgroup_id) {
+   bs = p;
+   break;
+  }
+ }
+ if (!bs && (new)) {
+  policy_init(new, name);
+  bs = new;
+  new = NULL;
+  list_add_tail(&bs->g_list, &banddevice_list);
+ }
+ spin_unlock_irqrestore(&banddevicelist_lock, flags);
+
+ if (new) {
+  destroy_workqueue(new->g_band_wq);
+  kfree(new);
+ }
+
+ return bs;
+}
+
+static inline void release_banddevice(struct banddevice *bs)
+{
+ unsigned long flags;
+
```

```
+ spin_lock_irqsave(&banddevicelist_lock, flags);
+ if (!list_empty(&bs->g_brothers)) {
+  spin_unlock_irqrestore(&banddevicelist_lock, flags);
+  return;
+ }
+ list_del(&bs->g_list);
+ spin_unlock_irqrestore(&banddevicelist_lock, flags);
+ destroy_workqueue(bs->g_band_wq);
+ kfree(bs);
+}
+
+static struct bandgroup *bandgroup_find(struct bandgroup *parent, int id)
+{
+ struct bandgroup *p;
+ struct bandgroup *bw = NULL;
+
+ list_for_each_entry(p, &parent->c_children, c_children) {
+  if (p->c_id == id || id == BAND_ID_ANY)
+   bw = p;
+ }
+ return bw;
+}
+
+static int bandgroup_init(struct bandgroup *bw,
+  struct bandgroup *parent, struct banddevice *bs, int id)
+{
+ unsigned long flags;
+
+ INIT_LIST_HEAD(&bw->c_list);
+ bio_list_init(&bw->c_blocked_bios);
+ bw->c_id = id; /* should be verified */
+ bw->c_going_down = 0;
+ bw->c_issued = 0;
+ bw->c_blocked = 0;
+ memset(bw->c_stat, 0, sizeof(bw->c_stat));
+
+ INIT_LIST_HEAD(&bw->c_children);
+
+ bw->c_shared = bs;
+
+ spin_lock_irqsave(&bs->g_lock, flags);
+ if (bandgroup_find(bw, id)) {
+  spin_unlock_irqrestore(&bs->g_lock, flags);
+  DMWARN("bandgroup: id=%d already exists.\n", id);
+  return -EEXIST;
+ }
+ bs->g_ref++;
+ list_add_tail(&bw->c_list, &bs->g_brothers);
```

```
+
+ bs->g_group_ctr(bw);
+
+ if (parent) {
+  list_add_tail(&bw->c_children, &parent->c_children);
+  bw->c_dev = parent->c_dev;
+ }
+
+ spin_unlock_irqrestore(&bs->g_lock, flags);
+
+ return 0;
+}
+
+static inline void bandgroup_release(struct bandgroup *bw)
+{
+ struct banddevice *bs = bw->c_shared;
+
+ list_del(&bw->c_list);
+ list_del(&bw->c_children);
+ bs->g_ref--;
+ bs->g_group_dtr(bw);
+ kfree(bw);
+}
+
+static void bandgroup_destroy_all(struct bandgroup *bw)
+{
+ struct banddevice *bs = bw->c_shared;
+ struct bandgroup *child;
+ unsigned long flags;
+
+ spin_lock_irqsave(&bs->g_lock, flags);
+ while ((child = bandgroup_find(bw, BAND_ID_ANY)))
+  bandgroup_release(child);
+ bandgroup_release(bw);
+ spin_unlock_irqrestore(&bs->g_lock, flags);
+}
+
+static void bandgroup_stop(struct bandgroup *bw)
+{
+ struct banddevice *bs = bw->c_shared;
+ unsigned long flags;
+
+ spin_lock_irqsave(&bs->g_lock, flags);
+ bw->c_going_down = 1;
+ spin_unlock_irqrestore(&bs->g_lock, flags);
+ queue_work(bs->g_band_wq, &bs->g_conductor);
+ flush_scheduled_work();
+}
```

```
+
+static void bandgroup_stop_all(struct bandgroup *parent)
+{
+ struct banddevice *bs = parent->c_shared;
+ struct bandgroup *p;
+ unsigned long flags;
+
+ spin_lock_irqsave(&bs->g_lock, flags);
+ list_for_each_entry(p, &parent->c_children, c_children)
+  p->c_going_down = 1;
+ parent->c_going_down = 1;
+ spin_unlock_irqrestore(&bs->g_lock, flags);
+ queue_work(bs->g_band_wq, &bs->g_conductor);
+ flush_scheduled_work();
+ /* should wait for bw->c_issued becoming zero? */
+}
+
+static void bandgroup_resume_all(struct bandgroup *parent)
+{
+ struct banddevice *bs = parent->c_shared;
+ struct bandgroup *p;
+ unsigned long flags;
+
+ spin_lock_irqsave(&bs->g_lock, flags);
+ list_for_each_entry(p, &parent->c_children, c_children)
+  p->c_going_down = 0;
+ parent->c_going_down = 0;
+ spin_unlock_irqrestore(&bs->g_lock, flags);
+}
+
+/*
+ * Create a new band device:
+ *   parameters:  <device> <device-group-id> [<io_throttle>] [<io_limit>]
+ */
+static int band_ctr(struct dm_target *ti, unsigned int argc, char **argv)
+{
+ struct bandgroup *bw;
+ struct banddevice *bs;
+ int io_throttle = DEFAULT_IO_THROTTLE;
+ int io_limit = DEFAULT_IO_LIMIT;
+ int devgroup_id;
+ int val;
+ int r = 0;
+
+ if (argc < 2) {
+  ti->error = "Requires 2 or more arguments";
+  return -EINVAL;
+ }
```

```
+
+ bw = kzalloc(sizeof(struct bandgroup), GFP_KERNEL);
+ if (!bw) {
+  ti->error = "Cannot allocate memory for bandgroup";
+  return -ENOMEM;
+ }
+
+ val = simple_strtol(argv[1], NULL, 0);
+ if (val < 0) {
+  ti->error = "Device Group ID # is too large";
+  r = -EINVAL;
+  goto error;
+ }
+ devgroup_id = val;
+ dprintk(KERN_ERR "band_ctr device group id:%d\n", val);
+
+ if (argc >= 3) {
+  val = simple_strtol(argv[2], NULL, 0);
+  if (val > 0) io_throttle = val;
+  dprintk(KERN_ERR "band_ctr ioqueue_low:%d\n", io_throttle);
+ }
+ if (argc >= 4) {
+  val = simple_strtol(argv[3], NULL, 0);
+  if (val > 0) io_limit = val;
+  dprintk(KERN_ERR "band_ctr ioqueue_high:%d\n", io_limit);
+ }
+ if (io_limit < io_throttle)
+  io_limit = io_throttle;
+
+ if (dm_get_device(ti, argv[0], 0, ti->len,
+     dm_table_get_mode(ti->table), &bw->c_dev)) {
+  ti->error = "band: device lookup failed";
+  r = -EINVAL;
+  goto error;
+ }
+
+ bs = alloc_banddevice(devgroup_id, "default", io_throttle, io_limit);
+ if (!bs) {
+  ti->error = "Cannot allocate memory for banddevice";
+  r = -ENOMEM;
+  goto error2;
+ }
+
+ bandgroup_init(bw, NULL, bs, BAND_ID_ANY);
+ bw->c_getid = band_group_type[0].t_getid;
+ strlcpy(bw->c_name, band_group_type[0].t_name, sizeof(bw->c_name));
+
+ ti->private = bw;
```

```
+
+ return 0;
+
+error2:
+ dm_put_device(ti, bw->c_dev);
+error:
+ kfree(bw);
+ return r;
+}
+
+static void band_dtr(struct dm_target *ti)
+{
+ struct bandgroup *bw = ti->private;
+ struct banddevice *bs = bw->c_shared;
+
+ bandgroup_stop_all(bw);
+ dm_put_device(ti, bw->c_dev);
+ bandgroup_destroy_all(bw);
+ release_banddevice(bs);
+}
+
+static void band_hold_bio(struct bandgroup *bw, struct bio *bio)
+{
+ /* Todo: The list should be split into a read list and a write list */
+ bio_list_add(&bw->c_blocked_bios, bio);
+}
+
+static struct bio *band_pop_bio(struct bandgroup *bw)
+{
+ return bio_list_pop(&bw->c_blocked_bios);
+}
+
+static inline void prepare_to_issue(struct bandgroup *bw, struct bio *bio)
+{
+ struct banddevice *bs = bw->c_shared;
+
+ bs->g_prepare_bio(bw, bio);
+ bs->g_issued++;
+ bw->c_issued++;
+ if (bs->g_issued >= bs->g_io_limit)
+  bs->g_plug_bio = 1;
+}
+
+static inline int room_for_bio(struct bandgroup *bw)
+{
+ struct banddevice *bs = bw->c_shared;
+
+ return !bs->g_plug_bio || bw->c_going_down;
```

```
+}
+
+static inline void hold_bio(struct bandgroup *bw, struct bio *bio)
+{
+ struct banddevice *bs = bw->c_shared;
+
+ bs->g_blocked++;
+ bw->c_blocked++;
+ bw->c_stat[bio_data_dir(bio)].deferred++;
+ bs->g_hold_bio(bw, bio);
+}
+
+static inline int release_bios(struct bandgroup *bw, struct bio_list *bl)
+{
+ struct banddevice *bs = bw->c_shared;
+ struct bio *bio;
+
+ while (bs->g_can_submit(bw) && bw->c_blocked) {
+  if (!room_for_bio(bw))
+   return 1;
+  bio = bs->g_pop_bio(bw);
+  if (!bio)
+   return 0;
+#if 0
+  if (bs->g_blocked == bs->g_io_limit*2)
+   wake_up(&bs->waitq_io);
+#endif
+  bs->g_blocked--;
+  bw->c_blocked--;
+  prepare_to_issue(bw, bio);
+  bio_list_add(bl, bio);
+ }
+
+ return 0;
+}
+
+static inline struct bandgroup *bandgroup_get(
+   struct bandgroup *parent, struct bio *bio)
+{
+ struct bandgroup *bw;
+
+ if (!parent->c_getid)
+  return parent;
+
+ bw = bandgroup_find(parent, parent->c_getid(bio));
+
+ if (!bw)
+  bw = parent;
```

```
+ return bw;
+}
+
+/*
+ * Start to control the bandwidth once the number of uncompleted BIOs
+ * exceeds the value of "io_throttle".
+ */
+static int band_map(struct dm_target *ti, struct bio *bio,
+       union map_info *map_context)
+{
+ struct bandgroup *bw = ti->private;
+ struct bandgroup_stat *bws;
+ struct banddevice *bs = bw->c_shared;
+ unsigned long flags;
+
+#if 0 /* not supported yet */
+ if (bio_barrier(bio))
+  return -EOPNOTSUPP;
+#endif
+
+ /*
+  * ToDo: Block to accept new requests when the number of the blocked
+  *       BIOs becomes extremely large.
+  */
+ bio->bi_bdev = bw->c_dev->bdev;
+ bio->bi_sector -= ti->begin;
+
+ spin_lock_irqsave(&bs->g_lock, flags);
+#if 0
+ /* ToDo: Should only stop processes requesting too many BIOs. */
+ wait_event_lock_irq(&bs->waitq_io, bs->g_blocked < bs->g_io_limit*2,
+      bs->g_lock, do_nothing);
+#endif
+ bw = bandgroup_get(bw, bio);
+ bws = &bw->c_stat[bio_data_dir(bio)];
+ bws->sectors += bio_sectors(bio);
+retry:
+ if (!bw->c_blocked && room_for_bio(bw)) {
+  if (bs->g_can_submit(bw)) {
+   bws->immediate++;
+   prepare_to_issue(bw, bio);
+   spin_unlock_irqrestore(&bs->g_lock, flags);
+   return 1;
+  } else if (bs->g_issued < bs->g_io_throttle) {
+   dprintk(KERN_ERR "band_map: token expired "
+     "bw:%p bio:%p\n", bw, bio);
+   if (bs->g_restart_bios(bs))
+    goto retry;
```

```
+  }
+ }
+ hold_bio(bw, bio);
+ spin_unlock_irqrestore(&bs->g_lock, flags);
+
+ return 0;
+}
+
+/*
+ * Select the best group to resubmit its BIOs.
+ */
+static inline struct bandgroup *choose_best_group(struct banddevice *bs)
+{
+ struct bandgroup *bw;
+ struct bandgroup *best = NULL;
+ int highest = 0;
+ int pri;
+
+ /* Todo: The algorithm should be optimized.
+  *       It would be better to use rbtree.
+  */
+ list_for_each_entry(bw, &bs->g_brothers, c_list) {
+  if (!bw->c_blocked || !room_for_bio(bw))
+   continue;
+  pri = bs->g_can_submit(bw);
+  if (pri > highest) {
+   highest = pri;
+   best = bw;
+  }
+ }
+
+ return best;
+}
+
+/*
+ * This function is called right after it becomes able to resubmit BIOs.
+ * It selects the best BIOs and passes them to the underlying layer.
+ */
+static void band_conduct(struct work_struct *work)
+{
+ struct banddevice *bs =
+  container_of(work, struct banddevice, g_conductor);
+ struct bandgroup *bw = NULL;
+ struct bio *bio;
+ unsigned long flags;
+ struct bio_list bl;
+
+ bio_list_init(&bl);
```

```
+
+ spin_lock_irqsave(&bs->g_lock, flags);
+retry:
+ while (bs->g_blocked && (bw = choose_best_group(bs)))
+  release_bios(bw, &bl);
+
+ if (bs->g_blocked && bs->g_issued < bs->g_io_throttle) {
+  dprintk(KERN_ERR "band_conduct: token expired bw:%p\n", bw);
+  if (bs->g_restart_bios(bs))
+   goto retry;
+ }
+
+ spin_unlock_irqrestore(&bs->g_lock, flags);
+
+ while ((bio = bio_list_pop(&bl)))
+  generic_make_request(bio);
+}
+
+static int band_end_io(struct dm_target *ti, struct bio *bio,
+       int error, union map_info *map_context)
+{
+ struct bandgroup *bw = ti->private;
+ struct banddevice *bs = bw->c_shared;
+ unsigned long flags;
+
+ /* Todo: The algorithm should be optimized to eliminate the spinlock. */
+ spin_lock_irqsave(&bs->g_lock, flags);
+ if (bs->g_issued <= bs->g_io_throttle)
+  bs->g_plug_bio = 0;
+ bs->g_issued--;
+ bw->c_issued--;
+
+ /*
+  * Todo: It would be better to introduce high/low water marks here
+  *   not to kick the workqueues so often.
+  */
+ if (bs->g_blocked && !bs->g_plug_bio)
+  queue_work(bs->g_band_wq, &bs->g_conductor);
+ spin_unlock_irqrestore(&bs->g_lock, flags);
+ return 0;
+}
+
+static void band_presuspend(struct dm_target *ti)
+{
+ struct bandgroup *bw = ti->private;
+ bandgroup_stop_all(bw);
+}
+
```

```
+static void band_resume(struct dm_target *ti)
+{
+ struct bandgroup *bw = ti->private;
+ bandgroup_resume_all(bw);
+}
+
+
+static void bandgroup_status(struct bandgroup *bw, int *szp, char *result,
+    unsigned int maxlen)
+{
+ struct bandgroup_stat *bws;
+ unsigned long reqs;
+ int i, sz = *szp; /* used in DMEMIT() */
+
+ if (bw->c_id == BAND_ID_ANY)
+  DMEMIT("\n default");
+ else
+  DMEMIT("\n %d", bw->c_id);
+ for (i = 0; i < 2; i++) {
+  bws = &bw->c_stat[i];
+  reqs = bws->immediate + bws->deferred;
+  DMEMIT(" %lu %lu %lu",
+   bws->immediate + bws->deferred, bws->deferred,
+   bws->sectors);
+ }
+ *szp = sz;
+}
+
+static int band_status(struct dm_target *ti, status_type_t type,
+   char *result, unsigned int maxlen)
+{
+ struct bandgroup *bw = ti->private, *p;
+ struct banddevice *bs = bw->c_shared;
+ int sz = 0; /* used in DMEMIT() */
+ unsigned long flags;
+
+ switch (type) {
+ case STATUSTYPE_INFO:
+  spin_lock_irqsave(&bs->g_lock, flags);
+  DMEMIT("devgrp=%u # read-req delay sect write-req delay sect",
+         bs->g_devgroup);
+  bandgroup_status(bw, &sz, result, maxlen);
+  list_for_each_entry(p, &bw->c_children, c_children)
+   bandgroup_status(p, &sz, result, maxlen);
+  spin_unlock_irqrestore(&bs->g_lock, flags);
+  break;
+
+ case STATUSTYPE_TABLE:
```

```
+  DMEMIT("%s devgrp=%u io_throttle=%u io_limit=%u",
+     bw->c_dev->name, bs->g_devgroup,
+     bs->g_io_throttle, bs->g_io_limit);
+  DMEMIT("\n id=default type=%s weight=%u token=%u",
+   bw->c_name, bw->c_weight, bw->c_token_init_value);
+  list_for_each_entry(p, &bw->c_children, c_children)
+   DMEMIT("\n id=%d weight=%u token=%u",
+    p->c_id, p->c_weight, p->c_token_init_value);
+  break;
+ }
+
+ return 0;
+}
+
+static int band_group_type_select(struct bandgroup *bw, char *name)
+{
+ struct banddevice *bs = bw->c_shared;
+ struct group_type *t;
+ unsigned long flags;
+
+ for (t = band_group_type; (t->t_name); t++) {
+  if (!strcmp(name, t->t_name))
+   break;
+ }
+ if (!t->t_name) {
+  DMWARN("band tyep select: %s isn't supported.\n", name);
+  return -EINVAL;
+ }
+ spin_lock_irqsave(&bs->g_lock, flags);
+ if (!list_empty(&bw->c_children)) {
+  spin_unlock_irqrestore(&bs->g_lock, flags);
+  return -EBUSY;
+ }
+ bw->c_getid = t->t_getid;
+ strlcpy(bw->c_name, t->t_name, sizeof(bw->c_name));
+ spin_unlock_irqrestore(&bs->g_lock, flags);
+
+ return 0;
+}
+
+static inline int split_string(char *s, char **v)
+{
+ int id = 0;
+ char *p, *q;
+
+ p = strsep(&s, "=:,");
+ q = strsep(&s, "=:,");
+ if (!q) {
```

```
+  *v = p;
+ } else {
+  id = simple_strtol(p, NULL, 0);
+  *v = q;
+ }
+ return id;
+}
+
+static int band_set_param(struct bandgroup *bw, char *cmd, char *value)
+{
+ struct banddevice *bs = bw->c_shared;
+ char *val_str;
+ int id;
+ unsigned long flags;
+ int r;
+
+ id = split_string(value, &val_str);
+
+ spin_lock_irqsave(&bs->g_lock, flags);
+ if (id) {
+  bw = bandgroup_find(bw, id);
+  if (!bw) {
+   spin_unlock_irqrestore(&bs->g_lock, flags);
+   DMWARN("band_set_param: id=%d not found.\n", id);
+   return -EINVAL;
+  }
+ }
+ r = bs->g_set_param(bw, cmd, val_str);
+ spin_unlock_irqrestore(&bs->g_lock, flags);
+ return r;
+}
+
+static int band_group_attach(struct bandgroup *bw, int id)
+{
+ struct banddevice *bs = bw->c_shared;
+ struct bandgroup *sub_bw;
+ int r;
+
+ if (id <= 0) {
+  DMWARN("band_group_attach: invalid id:%d\n", id);
+  return -EINVAL;
+ }
+ sub_bw = kzalloc(sizeof(struct bandgroup), GFP_KERNEL);
+ if (!sub_bw)
+  return -ENOMEM;
+
+ r = bandgroup_init(sub_bw, bw, bs, id);
+ if (r < 0) {
```

```
+  kfree(sub_bw);
+  return r;
+ }
+ return 0;
+}
+
+static int band_group_detach(struct bandgroup *bw, int id)
+{
+ struct banddevice *bs = bw->c_shared;
+ struct bandgroup *sub_bw;
+ unsigned long flags;
+
+ if (id <= 0) {
+  DMWARN("band_group_detach: invalid id:%d\n", id);
+  return -EINVAL;
+ }
+ spin_lock_irqsave(&bs->g_lock, flags);
+ sub_bw = bandgroup_find(bw, id);
+ spin_unlock_irqrestore(&bs->g_lock, flags);
+ if (!sub_bw) {
+  DMWARN("band_group_detach: invalid id:%d\n", id);
+  return -EINVAL;
+ }
+ bandgroup_stop(sub_bw);
+ spin_lock_irqsave(&bs->g_lock, flags);
+ bandgroup_release(sub_bw);
+ spin_unlock_irqrestore(&bs->g_lock, flags);
+ return 0;
+}
+
+/*
+ * Message parameters:
+ * "policy"      <name>
+ *       ex)
+ *  "policy" "weight"
+ * "type"        "none"|"pid"|"pgrp"|"node"|"cpuset"|"cgroup"|"user"
+ *  "io_throttle" <value>
+ *  "io_limit"    <value>
+ * "attach"      <group id>
+ * "detach"      <group id>
+ * "any-command" <group id>:<value>
+ *       ex)
+ *  "weight" 0:<value>
+ *  "token"  24:<value>
+ */
+static int band_message(struct dm_target *ti, unsigned int argc, char **argv)
+{
+ struct bandgroup *bw = ti->private, *p;
```

```
+ struct banddevice *bs = bw->c_shared;
+ int val = 0;
+ int r = 0;
+ unsigned long flags;
+
+ if (argc == 1 && !strcmp(argv[0], "reset")) {
+  spin_lock_irqsave(&bs->g_lock, flags);
+  memset(bw->c_stat, 0, sizeof(bw->c_stat));
+  list_for_each_entry(p, &bw->c_children, c_children)
+   memset(p->c_stat, 0, sizeof(p->c_stat));
+  spin_unlock_irqrestore(&bs->g_lock, flags);
+  return 0;
+ }
+
+ if (argc != 2) {
+  DMWARN("Unrecognised band message received.");
+  return -EINVAL;
+ }
+ if (!strcmp(argv[0], "debug")) {
+  band_debug = simple_strtol(argv[1], NULL, 0);
+  if (band_debug < 0) band_debug = 0;
+  return 0;
+ } else if (!strcmp(argv[0], "io_throttle")) {
+  val = simple_strtol(argv[1], NULL, 0);
+  spin_lock_irqsave(&bs->g_lock, flags);
+  if (val > 0 && val <= bs->g_io_limit)
+   bs->g_io_throttle = val;
+  spin_unlock_irqrestore(&bs->g_lock, flags);
+  return 0;
+ } else if (!strcmp(argv[0], "io_limit")) {
+  val = simple_strtol(argv[1], NULL, 0);
+  spin_lock_irqsave(&bs->g_lock, flags);
+  if (val > bs->g_io_throttle)
+   bs->g_io_limit = val;
+  spin_unlock_irqrestore(&bs->g_lock, flags);
+  return 0;
+ } else if (!strcmp(argv[0], "type")) {
+  return band_group_type_select(bw, argv[1]);
+ } else if (!strcmp(argv[0], "attach")) {
+  /* message attach <group-id> */
+  int id = simple_strtol(argv[1], NULL, 0);
+  return band_group_attach(bw, id);
+ } else if (!strcmp(argv[0], "detach")) {
+  /* message detach <group-id> */
+  int id = simple_strtol(argv[1], NULL, 0);
+  return band_group_detach(bw, id);
+ } else {
+  /* message anycommand <group-id>:<value> */
```

```
+  r = band_set_param(bw, argv[0], argv[1]);
+  if (r < 0)
+   DMWARN("Unrecognised band message received.");
+  return r;
+ }
+ return 0;
+}
+
+static struct target_type band_target = {
+ .name      = "band",
+ .module      = THIS_MODULE,
+ .version     = {0, 0, 2},
+ .ctr       = band_ctr,
+ .dtr       = band_dtr,
+ .map       = band_map,
+ .end_io      = band_end_io,
+ .presuspend  = band_presuspend,
+ .resume      = band_resume,
+ .status      = band_status,
+ .message     = band_message,
+};
+
+static int __init dm_band_init(void)
+{
+ int r;
+
+ r = dm_register_target(&band_target);
+ if (r < 0) {
+  DMERR("register failed %d", r);
+  return r;
+ }
+ return r;
+}
+
+static void __exit dm_band_exit(void)
+{
+ int r;
+
+ r = dm_unregister_target(&band_target);
+ if (r < 0)
+  DMERR("unregister failed %d", r);
+}
+
+module_init(dm_band_init);
+module_exit(dm_band_exit);
+
+MODULE_DESCRIPTION(DM_NAME " I/O bandwidth control");
+MODULE_AUTHOR("Hirokazu Takahashi <taka@valinux.co.jp>, "
```

+ "Ryo Tsuruta <ryov@valinux.co.jp");
+MODULE_LICENSE("GPL");

_____

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