Subject: Re: containers (was Re: -mm merge plans for 2.6.23)
Posted by Srivatsa Vaddagiri on Wed, 11 Jul 2007 11:39:53 GMT
View Forum Message <> Reply to Message

On Wed, Jul 11, 2007 at 12:19:58PM +0200, Ingo Molnar wrote:
> * Srivatsa Vaddagiri <vatsa@linux.vnet.ibm.com> wrote:
>
> > The other alternative is to hook up group scheduler with user-id's
> > (again only for 2.6.23).
>
> could you just try this and send an as simple patch as possible? This is
> actually something that non-container people would be interested in as
> well.

Note that interfacing with container infrastructure doesn't preclude the possibility of doing fair-user scheduling (that a normal university server or desktop user would want). All that is needed is a daemon which listens for uid change events from kernel (using process-event connector) and moves the task (whose uid is changing) to an appropriate container for that user.
Primitive source for such a daemon is attached.

> (btw., if this goes into 2.6.23 then we cannot possibly turn it off in 2.6.24,

The fact that we will have two interface for group scheduler in 2.6.24 is what worries me a bit (one user-id based and other container based). We would need some mechanism for admin to choose only one interface (and not both together, otherwise the group definitions may conflict), which doesn't sound very clean to me.

Ideally I would have liked to hook onto only container infrastructure and let user-space decide grouping policy (whether user-id based or something else).

Hmm ..would it help if I maintain a patch outside the mainline which turns on fair-user scheduling in 2.6.23-rcX? Folks will have to apply that patch on top of 2.6.23-rcX to use and test fair-user scheduling.

In 2.6.24, when container infrastructure goes in, people can get fair-user scheduling off-the-shelf by simply starting the daemon attached at bootup/initrd time.

Or would you rather prefer that I add user-id based interface permanently and in 2.6.24 introduce a compile/run-time switch for admin to select one of the two interfaces (user-id based or container-based)?

> so it must be sane - but per UID task groups are
> certainly sane, the only question is how to configure the per-UID weight
> after bootup.

Yeah ..the container based infrastructure allows for configuring such
things very easily using a fs-based interface. In the absence of that,
we either provide some /proc interface or settle for the non-configurable
default that you mention below.

> [the default after-bootup should be something along the
> lines of 'same weight for all users, a bit more for root'.]) This would
> make it possible for users to test that thing. (it would also help
> X-heavy workloads.)

--
Regards,
vatsa

```
/*
 * cpuctl_group_changer.c
 *
 * Used to change the group of running tasks to the correct
 * uid container.
 *
 * Copyright IBM Corporation, 2007
 * Author: Dhaval Giani <dhaval@linux.vnet.ibm.com>
 * Derived from test_cn_proc.c by Matt Helsley
 * Original copyright notice follows
 *
 * Copyright (C) Matt Helsley, IBM Corp. 2005
 * Derived from fcctl.c by Guillaume Thouvenin
 * Original copyright notice follows:
 *
 * Copyright (C) 2005 BULL SA.
 * Written by Guillaume Thouvenin <guillaume.thouvenin@bull.net>
 *
 * 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 02111-1307 USA
 */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include <string.h>

#include <sys/socket.h>
#include <sys/types.h>

#include <sys/stat.h>
#include <sys/param.h>

#include <linux/connector.h>
#include <linux/netlink.h>
#include "linux/cn_proc.h"

#include <errno.h>

#include <signal.h>
#include <setjmp.h>

#define SEND_MESSAGE_LEN (NLMSG_LENGTH(sizeof(struct cn_msg) + \
        sizeof(enum proc_cn_mcast_op)))
#define RECV_MESSAGE_LEN (NLMSG_LENGTH(sizeof(struct cn_msg) + \
        sizeof(struct proc_event)))

#define SEND_MESSAGE_SIZE    (NLMSG_SPACE(SEND_MESSAGE_LEN))
#define RECV_MESSAGE_SIZE    (NLMSG_SPACE(RECV_MESSAGE_LEN))

#define max(x,y) ((y)<(x)?(x):(y))
#define min(x,y) ((y)>(x)?(x):(y))

#define BUFF_SIZE (max(max(SEND_MESSAGE_SIZE, RECV_MESSAGE_SIZE), 1024))
#define MIN_RECV_SIZE (min(SEND_MESSAGE_SIZE, RECV_MESSAGE_SIZE))

#define PROC_CN_MCAST_LISTEN (1)
#define PROC_CN_MCAST_IGNORE (2)

/*
 * SIGINT causes the program to exit gracefully
 * this could happen any time after the LISTEN message has
 * been sent
 */
#define INTR_SIG SIGINT


sigjmp_buf g_jmp;
```

```
char cpuctl_fs_path[MAXPATHLEN];

void handle_intr (int signum)
{
 siglongjmp(g_jmp, signum);
}

static inline void itos(int i, char* str)
{
 sprintf(str, "%d", i);
}

int set_notify_release(int val)
{
 FILE *f;

 f = fopen("notify_on_release", "r+");
 fprintf(f, "%d\n", val);
 fclose(f);
 return 0;
}

int add_task_pid(int pid)
{
 FILE *f;

 f = fopen("tasks", "a");
 fprintf(f, "%d\n", pid);
 fclose(f);
 return 0;
}

int set_value(char* file, char *str)
{
 FILE *f;

 f=fopen(file, "w");
 fprintf(f, "%s", str);
 fclose(f);
 return 0;
}

int change_group(int pid, int uid)
{
 char str[100];
 int ret;

 ret = chdir(cpuctl_fs_path);
```

```
 itos(uid, str);
 ret = mkdir(str, 0777);
 if (ret == -1) {
  /*
   * If the folder already exists, then it is alright. anything
   * else should be killed
   */
  if (errno != EEXIST) {
   perror("mkdir");
   return -1;
  }
 }
 ret = chdir(str);
 if (ret == -1) {
  /*Again, i am just quitting the program!*/
  perror("chdir");
  return -1;
 }
 /*If using cpusets set cpus and mems*
  *
  * set_value("cpus","0");
  * set_value("mems","0");
  */
 set_notify_release(1);
 add_task_pid(pid);
 return 0;
}

int handle_msg (struct cn_msg *cn_hdr)
{
 struct proc_event *ev;
 int ret;

 ev = (struct proc_event*)cn_hdr->data;

 switch(ev->what){
 case PROC_EVENT_UID:
  printf("UID Change happening\n");
  printf("UID = %d\tPID=%d\n", ev->event_data.id.e.euid,
      ev->event_data.id.process_pid);
  ret = change_group(ev->event_data.id.process_pid,
     ev->event_data.id.r.ruid);
  break;
 case PROC_EVENT_FORK:
 case PROC_EVENT_EXEC:
 case PROC_EVENT_EXIT:
 default:
  break;
```

```
 }
 return ret;
}
int main(int argc, char **argv)
{
 int sk_nl;
 int err;
 struct sockaddr_nl my_nla, kern_nla, from_nla;
 socklen_t from_nla_len;
 char buff[BUFF_SIZE];
 int rc = -1;
 struct nlmsghdr *nl_hdr;
 struct cn_msg *cn_hdr;
 enum proc_cn_mcast_op *mcop_msg;
 size_t recv_len = 0;
 FILE *f;

 if (argc == 1)
  strcpy(cpuctl_fs_path, "/dev/cpuctl");
 else
  strcpy(cpuctl_fs_path, argv[1]);
 chdir(cpuctl_fs_path);
 f = fopen("tasks", "r");
 if (f == NULL) {
  printf("Container not mounted at %s\n", cpuctl_fs_path);
  return -1;
 }
 fclose(f);
 f = fopen("notify_on_release", "r");
 if (f == NULL) {
  printf("Container not mounted at %s\n", cpuctl_fs_path);
  return -1;
 }
 fclose(f);
 if (getuid() != 0) {
  printf("Only root can start/stop the fork connector\n");
  return 0;
 }
 /*
  * Create an endpoint for communication. Use the kernel user
  * interface device (PF_NETLINK) which is a datagram oriented
  * service (SOCK_DGRAM). The protocol used is the connector
  * protocol (NETLINK_CONNECTOR)
  */
 sk_nl = socket(PF_NETLINK, SOCK_DGRAM, NETLINK_CONNECTOR);
 if (sk_nl == -1) {
  printf("socket sk_nl error");
  return rc;
```

```
}
my_nla.nl_family = AF_NETLINK;
my_nla.nl_groups = CN_IDX_PROC;
my_nla.nl_pid = getpid();

kern_nla.nl_family = AF_NETLINK;
kern_nla.nl_groups = CN_IDX_PROC;
kern_nla.nl_pid = 1;

err = bind(sk_nl, (struct sockaddr *)&my_nla, sizeof(my_nla));
if (err == -1) {
 printf("binding sk_nl error");
 goto close_and_exit;
}
nl_hdr = (struct nlmsghdr *)buff;
cn_hdr = (struct cn_msg *)NLMSG_DATA(nl_hdr);
mcop_msg = (enum proc_cn_mcast_op*)&cn_hdr->data[0];
printf("sending proc connector: PROC_CN_MCAST_LISTEN... ");
memset(buff, 0, sizeof(buff));
*mcop_msg = PROC_CN_MCAST_LISTEN;
signal(INTR_SIG, handle_intr);
/* fill the netlink header */
nl_hdr->nlmsg_len = SEND_MESSAGE_LEN;
nl_hdr->nlmsg_type = NLMSG_DONE;
nl_hdr->nlmsg_flags = 0;
nl_hdr->nlmsg_seq = 0;
nl_hdr->nlmsg_pid = getpid();
/* fill the connector header */
cn_hdr->id.idx = CN_IDX_PROC;
cn_hdr->id.val = CN_VAL_PROC;
cn_hdr->seq = 0;
cn_hdr->ack = 0;
cn_hdr->len = sizeof(enum proc_cn_mcast_op);
if (send(sk_nl, nl_hdr, nl_hdr->nlmsg_len, 0) != nl_hdr->nlmsg_len) {
 printf("failed to send proc connector mcast ctl op!\n");
 goto close_and_exit;
}
printf("sent\n");
for(memset(buff, 0, sizeof(buff)), from_nla_len = sizeof(from_nla);
  ; memset(buff, 0, sizeof(buff)), from_nla_len = sizeof(from_nla)) {
 struct nlmsghdr *nlh = (struct nlmsghdr*)buff;
 memcpy(&from_nla, &kern_nla, sizeof(from_nla));
 recv_len = recvfrom(sk_nl, buff, BUFF_SIZE, 0,
   (struct sockaddr*)&from_nla, &from_nla_len);
 if (recv_len < 1)
  continue;
 while (NLMSG_OK(nlh, recv_len)) {
  cn_hdr = NLMSG_DATA(nlh);
```

```
   if (nlh->nlmsg_type == NLMSG_NOOP)
    continue;
   if ((nlh->nlmsg_type == NLMSG_ERROR) ||
       (nlh->nlmsg_type == NLMSG_OVERRUN))
    break;
   if(handle_msg(cn_hdr)<0) {
    goto close_and_exit;
   }
   if (nlh->nlmsg_type == NLMSG_DONE)
    break;
   nlh = NLMSG_NEXT(nlh, recv_len);
  }
 }
close_and_exit:
 close(sk_nl);
 exit(rc);

 return 0;
}
```

File Attachments

1) cpuctld.c, downloaded 337 times