Subject: [PATCH 0/2][cryo] Save/restore pipe state
Posted by Sukadev Bhattiprolu on Tue, 24 Jun 2008 03:23:43 GMT
View Forum Message <> Reply to Message

[PATCH 1/2] Save/restore state of unnamed pipes

Basic infrastructure to save/restore pipe state with assumptions
about order of fds.

[PATCH 2/2] Support Non-consecutive and dup pipe fds

Remove above assumptions about order of fds and support dups of
pipe fds.

_____

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

Subject: [PATCH 1/2][cryo] Save/restore state of unnamed pipes
Posted by Sukadev Bhattiprolu on Tue, 24 Jun 2008 03:25:36 GMT
View Forum Message <> Reply to Message

>From e513f8bc0fe808425264ad01210ac610f6453047 Mon Sep 17 00:00:00 2001
From: Sukadev Bhattiprolu <sukadev@linux.vnet.ibm.com>
Date: Mon, 16 Jun 2008 18:41:05 -0700
Subject: [PATCH] Save/restore state of unnamed pipes

Design:

Current Linux kernels provide ability to read/write contents of FIFOs
using /proc. i.e 'cat /proc/pid/fd/read-side-fd' prints the unread data
in the FIFO. Similarly, 'cat foo > /proc/pid/fd/read-sid-fd' appends
the contents of 'foo' to the unread contents of the FIFO.

So to save/restore the state of the pipe, a simple implementation is
to read the from the unnamed pipe's fd and save to the checkpoint-file.
When restoring, create a pipe (using PT_PIPE()) in the child process,
read the contents of the pipe from the checkpoint file and write it to
the newly created pipe.

Its fairly straightforward, except for couple of notes:

- when we read contents of '/proc/pid/fd/read-side-fd' we drain
  the pipe such that when the checkpointed application resumes,
  it will not find any data. To fix this, we read from the
  'read-side-fd' and write it back to the 'read-side-fd' in
  addition to writing to the checkpoint file.

- there does not seem to be a mechanism to determine the count
  of unread bytes in the file. Current implmentation assumes a
  maximum of 64K bytes (PIPE_BUFS * PAGE_SIZE on i386) and fails
  if the pipe is not fully drained.

Changelog:[v1]:

- [Serge Hallyn]: use || instead of && in ensure_fifo_has_drained

- [Serge Hallyn, Matt Helsley]: Use dup2() to restore fds and
  remove assumptions about order of read and write fds
  (addressed in PATCH 2/2).

Some unit-testing done at this point (using tests/pipe.c).

TODO:
- Additional testing (with multiple-processes and multiple-pipes)
- Named-pipes
---
 cr.c | 217 ++----
 1 files changed, 205 insertions(+), 12 deletions(-)

diff --git a/cr.c b/cr.c
index c7e3332..716cc86 100644
--- a/cr.c
+++ b/cr.c
@@ -88,6 +88,11 @@ typedef struct fdinfo_t {
  char name[128]; /* file name. NULL if anonymous (pipe, socketpair) */
 } fdinfo_t;

+typedef struct fifoinfo_t {
+ int fi_fd; /* fifo's read-side fd */
+ int fi_length; /* number of bytes in the fifo */
+} fifofdinfo_t;
+
 typedef struct memseg_t {
  unsigned long start; /* memory segment start address */
  unsigned long end; /* memory segment end address */
@@ -499,6 +504,129 @@ out:
  return rc;
 }

+static int estimate_fifo_unread_bytes(pinfo_t *pi, int fd)
+{
+ /*
+ * Is there a way to find the number of bytes remaining to be
+ * read in a fifo ? If not, can we print it in fdinfo ?

```
+  *
+  * Return 64K (PIPE_BUFS * PAGE_SIZE) for now.
+  */
+ return 65536;
+}
+
+static void ensure_fifo_has_drained(char *fname, int fifo_fd)
+{
+ int rc, c;
+
+ errno = 0;
+ rc = read(fifo_fd, &c, 1);
+ if (rc != -1 || errno != EAGAIN) {
+  ERROR("FIFO '%s' not drained fully. rc %d, c %d "
+   "errno %d\n", fname, rc, c, errno);
+ }
+
+}
+
+static int save_process_fifo_info(pinfo_t *pi, int fd)
+{
+ int i;
+ int rc;
+ int nbytes;
+ int fifo_fd;
+ int pbuf_size;
+ pid_t pid = pi->pid;
+ char fname[256];
+ fdinfo_t *fi = pi->fi;
+ char  *pbuf;
+ fifofdinfo_t fifofdinfo;
+
+ write_item(fd, "FIFO", NULL, 0);
+
+ for (i = 0; i < pi->nf; i++) {
+  if (! S_ISFIFO(fi[i].mode))
+   continue;
+
+  DEBUG("FIFO fd %d (%s), flag 0x%x\n", fi[i].fdnum, fi[i].name,
+    fi[i].flag);
+
+  if (!(fi[i].flag & O_WRONLY))
+   continue;
+
+  pbuf_size = estimate_fifo_unread_bytes(pi, fd);
+
+  pbuf = (char *)malloc(pbuf_size);
+  if (!pbuf) {
```

```
+   ERROR("Unable to allocate FIFO buffer of size %d\n",
+     pbuf_size);
+  }
+  memset(pbuf, 0, pbuf_size);
+
+  sprintf(fname, "/proc/%u/fd/%u", pid, fi[i].fdnum);
+
+  /*
+   * Open O_NONBLOCK so read does not block if fifo has fewer
+   * bytes than our estimate.
+   */
+  fifo_fd = open(fname, O_RDWR|O_NONBLOCK);
+  if (fifo_fd < 0)
+   ERROR("Error %d opening FIFO '%s'\n", errno, fname);
+
+  nbytes = read(fifo_fd, pbuf, pbuf_size);
+  if (nbytes < 0) {
+   if (errno != EAGAIN) {
+    ERROR("Error %d reading FIFO '%s'\n", errno,
+      fname);
+   }
+   nbytes = 0;  /* empty fifo */
+  }
+
+  /*
+   * Ensure FIFO has been drained.
+   *
+   * TODO: If FIFO has not fully drained, our estimate of
+   *   unread-bytes is wrong. We could:
+   *
+   *   - have kernel print exact number of unread-bytes
+   *     in /proc/pid/fdinfo/<fd>
+   *
+   *   - read in contents multiple times and write multiple
+   *     fifobufs or assemble them into a single, large
+   *     buffer.
+   */
+  ensure_fifo_has_drained(fname, fifo_fd);
+
+  /*
+   * Save FIFO data to checkpoint file
+   */
+  fifofdinfo.fi_fd = fi[i].fdnum;
+  fifofdinfo.fi_length = nbytes;
+  write_item(fd, "fifofdinfo", &fifofdinfo, sizeof(fifofdinfo));
+
+  if (nbytes) {
+   write_item(fd, "fifobufs", pbuf, nbytes);
```

```
+
+   /*
+    * Restore FIFO's contents so checkpointed application
+    * won't miss a thing.
+    */
+   errno = 0;
+   rc = write(fifo_fd, pbuf, nbytes);
+   if (rc != nbytes) {
+    ERROR("Wrote-back only %d of %d bytes to FIFO, "
+      "error %d\n", rc, nbytes, errno);
+   }
+  }
+
+  close(fifo_fd);
+  free(pbuf);
+ }
+
+ write_item(fd, "END FIFO", NULL, 0);
+
+ return 0;
+}
+
 static int save_process_data(pid_t pid, int fd, lh_list_t *ptree)
 {
  char fname[256], exe[256], cwd[256], *argv, *env, *buf;
@@ -618,6 +746,8 @@ static int save_process_data(pid_t pid, int fd, lh_list_t *ptree)
  }
  write_item(fd, "END FD", NULL, 0);

+ save_process_fifo_info(pi, fd);
+
  /* sockets */
  write_item(fd, "SOCK", NULL, 0);
  for (i = 0; i < pi->ns; i++)
@@ -870,6 +1000,29 @@ int restore_fd(int fd, pid_t pid)
     }
     if (pfd != fdinfo->fdnum) t_d(PT_CLOSE(pid, pfd));
    }
+  } else if (S_ISFIFO(fdinfo->mode)) {
+   int pipefds[2] = { 0, 0 };
+
+   /*
+    * We create the pipe when we see the pipe's read-fd.
+    * Just ignore the pipe's write-fd.
+    */
+   if (fdinfo->flag == O_WRONLY)
+    continue;
+
```

```
+   DEBUG("Creating pipe for fd %d\n", fdinfo->fdnum);
+
+   t_d(PT_PIPE(pid, pipefds));
+   t_d(pipefds[0]);
+   t_d(pipefds[1]);
+
+   if (pipefds[0] != fdinfo->fdnum) {
+    DEBUG("Hmm, new pipe has fds %d, %d "
+     "Old pipe had fd %d\n", pipefds[0],
+     pipefds[1], fdinfo->fdnum); getchar();
+    exit(1);
+   }
+   DEBUG("Done creating pipefds[0] %d\n", pipefds[0]);
   }

   /*
@@ -878,20 +1031,8 @@ int restore_fd(int fd, pid_t pid)
   ret = PT_FCNTL(pid, fdinfo->fdnum, F_SETFL, fdinfo->flag);
   DEBUG("---- restore_fd() fd %d setfl flag 0x%x, ret %d\n",
    fdinfo->fdnum, fdinfo->flag, ret);
-
-
   free(fdinfo);
  }
- if (1) {
-  /* test: force pipe creation */
-  static int first = 1;
-  int pipe[2] = { 0, 0 };
-  if (! first) return 0;
-  else first = 0;
-  t_d(PT_PIPE(pid, pipe));
-  t_d(pipe[0]);
-  t_d(pipe[1]);
- }
  return 0;
 error:
  free(fdinfo);
@@ -1286,6 +1427,56 @@ int restore_sig(pid_t pid, struct sigaction *sigact, sigset_t *sigmask,
sigset_t
  return 0;
 }

+int restore_fifo(int fd, pid_t pid)
+{
+ char item[64];
+ void *buf = NULL;
+ size_t bufsz;
+ int ret;
```

```
+ int fifo_fd;
+ char fname[64];
+ int nbytes;
+ fifofdinfo_t *fifofdinfo = NULL;
+
+ for(;;) {
+  ret = read_item(fd, item, sizeof(item), &buf, &bufsz);
+  DEBUG("restore_fifo() read item '%.12s'\n", item);
+  if ITEM_IS("END FIFO")
+   break;
+  else ITEM_SET(fifofdinfo, fifofdinfo_t);
+  else if ITEM_IS("fifobufs") {
+   DEBUG("restore_fifo() bufsz %d, fi_fd %d, length %d\n",
+     bufsz, fifofdinfo->fi_fd,
+     fifofdinfo->fi_length);
+
+   if (!fifofdinfo->fi_length)
+    continue;
+
+   sprintf(fname, "/proc/%u/fd/%d", pid,
+     fifofdinfo->fi_fd);
+
+   fifo_fd = open(fname, O_WRONLY|O_NONBLOCK);
+   if (fifo_fd < 0) {
+    ERROR("Error %d opening FIFO '%s'\n", errno,
+      fname);
+   }
+
+   errno = 0;
+   nbytes = write(fifo_fd, buf, bufsz);
+   if (nbytes != bufsz) {
+    ERROR("Wrote %d of %d bytes to FIFO '%s', "
+      "errno %d\n", nbytes, bufsz,
+      fname, errno);
+   }
+   close(fifo_fd);
+  } else
+   ERROR("Unexpected item, '%s'\n", item);
+ }
+ DEBUG("restore_fifo() fd %d, len %d, got 'END FIFO'\n",
+   fifofdinfo->fi_fd, fifofdinfo->fi_length);
+ return 0;
+}
+
 static int process_restart(int fd, int mode)
 {
  char item[64];
@@ -1369,6 +1560,8 @@ static int process_restart(int fd, int mode)
```

```
     ptrace_set_thread_area(npid, ldt);
    if (cwd) PT_CHDIR(npid, cwd);
    restore_fd(fd, npid);
+  } else if (ITEM_IS("FIFO")) {
+   restore_fifo(fd, npid);
   } else if (ITEM_IS("SOCK")) {
    restore_sock(fd, npid);
   } else if (ITEM_IS("SEMUNDO")) {
--
1.5.2.5
```

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

---

Subject: [PATCH 2/2] Support Non-consecutive and dup pipe fds
Posted by Sukadev Bhattiprolu on Tue, 24 Jun 2008 03:26:20 GMT
View Forum Message <> Reply to Message

>From a80c5215763f757840214465277e911e46e01219 Mon Sep 17 00:00:00 2001
From: Sukadev Bhattiprolu <sukadev@linux.vnet.ibm.com>
Date: Mon, 23 Jun 2008 20:13:57 -0700
Subject: [PATCH] Support Non-consecutive and dup pipe fds

PATCH 1/1 provides basic infrastructure to save/restore state of
pipes. This patch removes assumptions about order of the pipe-fds
and also supports existence of 'dups' of pipe-fds.

This logic has been separated from PATCH 1/1 for easier review and
the two patches could be combined into a single one.

Thanks to Matt Helsley for the optimized logic/code in match_pipe_ends().

TODO:
 There are few TODO's marked out in the patch. Hopefully these
 can be addressed without significant impact to the central-logic
 of saving/restoring pipes.

 - Temporarily using a regular-file's fd as 'trampoline-fd' when
   all fds are in use

 - Maybe read all fdinfo into memory during restart, so we can
   reduce the information we save into the checkpoint-file
   (see comments near 'struct fdinfo').

 - Check logic of detecting 'dup's of pipe fds (any hidden

```
   gotchas ?) See pair_pipe_fds()

 - Alloc ppi_list[] dynamically (see getfdinfo()).

 - Use getrlimit() to compute max-open-fds (see near caller of
   pair_pipe_fds()).

 - [Oleg Nesterov]: SIGIO/inotify() issues associated with writing-back
   to pipes (fixing this would require some assitance from kernel ?)

Ran several unit-test cases (see test-patches). Additional cases to be
developed/executed.

Signed-off-by: Sukadev Bhattiprolu <sukadev@us.ibm.com>
---
 cr.c |  262 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++------
 1 files changed, 240 insertions(+), 22 deletions(-)

diff --git a/cr.c b/cr.c
index 716cc86..f40a4fb 100644
--- a/cr.c
+++ b/cr.c
@@ -79,8 +79,25 @@ typedef struct isockinfo_t {
  char tcpstate[TPI_LEN];
 } isockinfo_t;

+/*
+ * TODO: restore_fd() processes each fd as it reads it of the checkpoint
+ * file. To avoid making a second-pass at the file, we store following
+ * fields during checkpoint (for now).
+ *
+ *  peer_fdnum, dup_fdnum, create_pipe, tramp_fd' fields can be
+ *
+ *  We could eliminate this fields by reading all fdinfo into memory
+ *  and then 'computing' the above fields before processing the fds.
+ *  But this would require a non-trivial rewrite of the restore_fd()
+ *  logic. Hopefully that can be done without significant impact to
+ *  rest of the logic associated with saving/restoring pipes.
+ */
 typedef struct fdinfo_t {
  int fdnum;  /* file descriptor number */
+ int peer_fdnum;  /* peer fd for pipes */
+ int dup_fdnum;  /* fd, if fd is dup of another pipe fd */
+ int create_pipe; /* TRUE if this is the create-end of the pipe */
+ int tramp_fd;  /* trampoline-fd for use in restoring pipes */
  mode_t mode;  /* mode as per stat(2) */
  off_t offset;  /* read/write pointer position for regular files */
  int flag;  /* open(2) flag */
```

```
@@ -117,6 +134,7 @@ typedef struct pinfo_t {
  int nt;   /* number of thread child (0 if no thread lib) */
  pid_t *tpid;  /* array of thread info */
  struct pinfo_t *pmt; /* multithread: pointer to main thread info */
+ int tramp_fd;  /* trampoline-fd for use in restoring pipes */
 } pinfo_t;

 /*
@@ -263,6 +281,89 @@ int getsockinfo(pid_t pid, pinfo_t *pi, int num)
  return ret;
 }

+typedef struct pipe_peer_info {
+ fdinfo_t *pipe_fdi;
+ //fdinfo_t *peer_fdi;
+ __ino_t pipe_ino;
+} pipe_peer_info_t;
+
+__ino_t get_fd_ino(char *fname)
+{
+ struct stat sbuf;
+
+ if (stat(fname, &sbuf) < 0)
+  ERROR("stat() on fd %s failed, errno %d\n", fname, errno);
+
+ return sbuf.st_ino;
+}
+
+static void pair_pipe_fds(pipe_peer_info_t *ppi_list, int npipe_fds)
+{
+ int i, j;
+ pipe_peer_info_t *xppi, *yppi;
+ fdinfo_t *xfdi, *yfdi;
+
+ /*
+  * TODO: This currently assumes pipefds have not been dup'd.
+  *   Of course, need to kill this assumption soon.
+  */
+ for (i = 0; i < npipe_fds; i++) {
+  xppi = &ppi_list[i];
+  xfdi = xppi->pipe_fdi;
+
+  j = i + 1;
+  for (j = i+1; j < npipe_fds; j++) {
+   yppi = &ppi_list[j];
+   yfdi = yppi->pipe_fdi;
+
+   if (yppi->pipe_ino != xppi->pipe_ino)
```

```
+    continue;
+
+   DEBUG("Checking flag i %d, j %d\n", i, j);
+   /*
+    * i and j refer to same pipe. Check if they are
+    * peers or aliases (dup'd fds). dup'd fds share
+    * file-status flags. Peer fds of unnamed pipes
+    * differ in O_WRONLY bit.
+    *
+    * TODO:
+    *  CHECK ABOVE ASSUMPTION
+    */
+   if (xfdi->flag == yfdi->flag) {
+    yfdi->dup_fdnum = xfdi->fdnum;
+    DEBUG("Pipe fds %d and %d are dups\n",
+      xfdi->fdnum, yfdi->fdnum);
+   } else {
+    DEBUG("Pipe fds %d and %d are peers\n",
+      xfdi->fdnum, yfdi->fdnum);
+
+    /*
+     * If we have already paired it or determined
+     * it is a dup, ignore
+     */
+    if (xfdi->peer_fdnum != -1 ||
+      xfdi->dup_fdnum != -1)
+     continue;
+
+    DEBUG("Pipe fd %d not paired yet\n",
+      xfdi->fdnum);
+    /*
+     * Create pipe on first end of the pipe we
+     * come across.
+     */
+    if (xfdi->fdnum < yfdi->fdnum)
+     xfdi->create_pipe = 1;
+
+    xfdi->peer_fdnum = yfdi->fdnum;
+    yfdi->peer_fdnum = xfdi->fdnum;
+   }
+   DEBUG("Clearing ino i %d, j %d\n", i, j);
+  }
+ }
+
+ DEBUG("Done building pipe list i %d, j %d\n", i, j);
+}
+
 /*
```

```
  * getfds() parse the process open file descriptors as found in /proc.
  */
@@ -275,6 +376,10 @@ int getfdinfo(pinfo_t *pi)
  int len, n = 0;
  pid_t syscallpid = pi->syscallpid ? pi->syscallpid : pi->pid;

+ int npipe_fds = 0;
+ pipe_peer_info_t ppi_list[256];// TODO: alloc dynamically
+ pipe_peer_info_t *ppi;
+
  snprintf(dname, sizeof(dname), "/proc/%u/fd", pi->pid);
  if (! (dir = opendir(dname))) return 0;
  while ((dent = readdir(dir))) {
@@ -288,14 +393,54 @@ int getfdinfo(pinfo_t *pi)
   stat(dname, &st);
   pi->fi[n].mode = st.st_mode;
   pi->fi[n].flag = PT_FCNTL(syscallpid, pi->fi[n].fdnum, F_GETFL, 0);
+  pi->fi[n].create_pipe = 0;
+  pi->fi[n].tramp_fd = -1;
+  pi->fi[n].dup_fdnum = -1;
+  pi->fi[n].peer_fdnum = -1;
   if (S_ISREG(st.st_mode))
    pi->fi[n].offset = (off_t)PT_LSEEK(syscallpid, pi->fi[n].fdnum, 0, SEEK_CUR);
-  else if (S_ISFIFO(st.st_mode))
+  else if (S_ISFIFO(st.st_mode)) {
    t_s("fifo");
+   ppi = &ppi_list[npipe_fds];
+   ppi->pipe_fdi = &pi->fi[n];
+   //ppi->peer_fdi = NULL;
+   ppi->pipe_ino = get_fd_ino(dname);
+
+   DEBUG("Found a pipe: fd %d, flag 0x%x ino %d\n",
+     ppi->pipe_fdi->fdnum,
+     ppi->pipe_fdi->flag, ppi->pipe_ino);
+   npipe_fds++;
+
+  }
   else if (S_ISSOCK(st.st_mode))
    getsockinfo(syscallpid, pi, pi->fi[n].fdnum);
   n++;
  }
+
+ if (n) {
+  pi->tramp_fd = pi->fi[n-1].fdnum + 1;
+  DEBUG("Using %d as trampoline-fd\n", pi->tramp_fd);
+ }
+
+ /*
```

```
+  * TODO: replace 1024 with rlim.rlim_cur (but need to execute
+  *   getrlimit() in checkpointed-process)
+  */
+ if (npipe_fds && pi->tramp_fd >= 1024) {
+  /*
+   * TODO:
+   *  This restriction can be relaxed a bit. We only
+   *  need to give-up here if all fds are in use as
+   *  pipe-fds.
+   */
+  ERROR("Cannot allocate a 'trampoline_fd' for pipes");
+ }
+
+ /*
+  * Now that we found all fds,  pair up any pipe_fds
+  */
+ pair_pipe_fds(ppi_list, npipe_fds);
+
 end:
  closedir(dir);
  return n;
@@ -550,7 +695,11 @@ static int save_process_fifo_info(pinfo_t *pi, int fd)
   DEBUG("FIFO fd %d (%s), flag 0x%x\n", fi[i].fdnum, fi[i].name,
    fi[i].flag);

-  if (!(fi[i].flag & O_WRONLY))
+  /*
+   * If its read-side fd or a dup of another write-side-fd,
+   * don't need the data.
+   */
+  if (!(fi[i].flag & O_WRONLY) || fi[i].dup_fdnum != -1)
    continue;

   pbuf_size = estimate_fifo_unread_bytes(pi, fd);
@@ -742,6 +891,12 @@ static int save_process_data(pid_t pid, int fd, lh_list_t *ptree)
  write_item(fd, "FD", NULL, 0);
  t_d(pi->nf);
  for (i = 0; i < pi->nf; i++) {
+  /*
+   * trampoline-fd is common to all fds, so we could write it
+   * once, as a separate item by itself. Stick it in each
+   * fdinfo for now.
+   */
+  pi->fi[i].tramp_fd = pi->tramp_fd;
   write_item(fd, "fdinfo", &pi->fi[i], sizeof(fdinfo_t));
  }
  write_item(fd, "END FD", NULL, 0);
@@ -949,6 +1104,74 @@ pid_t restart_thread(pid_t ppid, int exitsig, int addr)
```

```
 return pid;
}

+static void match_pipe_ends(int pid, int tramp_fd, int expected_fds[],
+  int actual_fds[])
+{
+ int ret;
+ int expected_read_fd = expected_fds[0];
+ int expected_write_fd = expected_fds[1];
+
+ DEBUG("tramp_fd %d\n", tramp_fd);
+       /*
+        * pipe() may have returned one (or both) of the restarted fds
+        * at the wrong end of the pipe. This could cause dup2() to
+        * accidentaly close the pipe. Avoid that with an extra dup().
+        */
+       if (actual_fds[1] == expected_read_fd) {
+               t_d(ret = PT_DUP2(pid, actual_fds[1], tramp_fd + 1));
+               actual_fds[1] = tramp_fd + 1;
+       }
+
+       if (actual_fds[0] != expected_read_fd) {
+               t_d(ret = PT_DUP2(pid, actual_fds[0], expected_read_fd));
+               t_d(PT_CLOSE(pid, actual_fds[0]));
+       }
+
+       if (actual_fds[1] != expected_write_fd) {
+               t_d(ret = PT_DUP2(pid, actual_fds[1], expected_write_fd));
+               t_d(PT_CLOSE(pid, actual_fds[1]));
+       }
+}
+
+static void recreate_pipe(int pid, int tramp_fd, fdinfo_t *fdinfo)
+{
+ int actual_fds[2] = { 0, 0 };
+ int expected_fds[2];
+
+ DEBUG("Creating pipe for fd %d\n", fdinfo->fdnum);
+
+ t_d(PT_PIPE(pid, actual_fds));
+ t_d(actual_fds[0]);
+ t_d(actual_fds[1]);
+
+ /*
+  * Find read-end and write-end of the checkpointed pipe
+  * (i.e don't assume that read-side fd is smaller than
+  * write-side fd)
+  */
```

```
+ if (fdinfo->flag & O_WRONLY) {
+  expected_fds[0] = fdinfo->peer_fdnum;
+  expected_fds[1] = fdinfo->fdnum;
+ } else {
+  expected_fds[0] = fdinfo->fdnum;
+  expected_fds[1] = fdinfo->peer_fdnum;
+ }
+
+ /*
+  * Match the ends of newly created pipe with the ends of the
+  * checkpointed pipe.
+  */
+ match_pipe_ends(pid, tramp_fd, expected_fds, actual_fds);
+
+ /*
+  * for debug, use fcntl() on fdinfo->fdnum and fdinfo->peer_fdnum
+  * to ensure ends match
+  */
+
+ DEBUG("Done creating pipe '{%d, %d}'\n", fdinfo->fdnum,
+   fdinfo->peer_fdnum);
+}
+
 int restore_fd(int fd, pid_t pid)
 {
  char item[64];
@@ -1001,34 +1224,29 @@ int restore_fd(int fd, pid_t pid)
     if (pfd != fdinfo->fdnum) t_d(PT_CLOSE(pid, pfd));
    }
   } else if (S_ISFIFO(fdinfo->mode)) {
-   int pipefds[2] = { 0, 0 };
-
+   if (fdinfo->dup_fdnum != -1) {
+    t_d(ret = PT_DUP2(pid, fdinfo->dup_fdnum,
+       fdinfo->fdnum));
+   }
    /*
-    * We create the pipe when we see the pipe's read-fd.
-    * Just ignore the pipe's write-fd.
+    * When checkpointing, we arbitrarily mark one end
+    * of the pipe as the 'create-end'.  Create a pipe
+    * if this fd is the 'create-end' and then restore
+    * the fcntl-flags. For the other end of the pipe,
+    * just restore its fcntl-flags.
     */
-   if (fdinfo->flag == O_WRONLY)
-    continue;
-
```

```
-   DEBUG("Creating pipe for fd %d\n", fdinfo->fdnum);
-
-   t_d(PT_PIPE(pid, pipefds));
-   t_d(pipefds[0]);
-   t_d(pipefds[1]);
-
-   if (pipefds[0] != fdinfo->fdnum) {
-    DEBUG("Hmm, new pipe has fds %d, %d "
-     "Old pipe had fd %d\n", pipefds[0],
-     pipefds[1], fdinfo->fdnum); getchar();
-    exit(1);
-   }
-   DEBUG("Done creating pipefds[0] %d\n", pipefds[0]);
+   else if (fdinfo->create_pipe)
+    recreate_pipe(pid, fdinfo->tramp_fd, fdinfo);
   }

   /*
    * Restore any special flags this fd had
    */
   ret = PT_FCNTL(pid, fdinfo->fdnum, F_SETFL, fdinfo->flag);
+  if (ret < 0) {
+   ERROR("restore_fd() fd %d setfl flag 0x%x, ret %d\n",
+     fdinfo->fdnum, fdinfo->flag, ret);
+  }
   DEBUG("---- restore_fd() fd %d setfl flag 0x%x, ret %d\n",
     fdinfo->fdnum, fdinfo->flag, ret);
   free(fdinfo);
--
1.5.2.5
```

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

---

Subject: Re: [PATCH 1/2][cryo] Save/restore state of unnamed pipes
Posted by Sukadev Bhattiprolu on Tue, 24 Jun 2008 16:15:31 GMT
View Forum Message <> Reply to Message

```
| +  } else
| +   ERROR("Unexpected item, '%s'\n", item);
| + }
| + DEBUG("restore_fifo() fd %d, len %d, got 'END FIFO'\n",
| +   fifofdinfo->fi_fd, fifofdinfo->fi_length);
| + return 0;
| +}
```

fifofdinfo can be NULL if application does not have any fifos and
the above DEBUG() will SIGSEGV. Following updated patch fixes this.
Thanks Serge for reporting the bug.

>From bdb9d8f20abd412a46a4e3951ed084ca5690e7d2 Mon Sep 17 00:00:00 2001
From: Sukadev Bhattiprolu <sukadev@linux.vnet.ibm.com>
Date: Mon, 16 Jun 2008 18:41:05 -0700
Subject: [PATCH] Save/restore state of unnamed pipes

Design:

Current Linux kernels provide ability to read/write contents of FIFOs
using /proc. i.e 'cat /proc/pid/fd/read-side-fd' prints the unread data
in the FIFO. Similarly, 'cat foo > /proc/pid/fd/read-sid-fd' appends
the contents of 'foo' to the unread contents of the FIFO.

So to save/restore the state of the pipe, a simple implementation is
to read the from the unnamed pipe's fd and save to the checkpoint-file.
When restoring, create a pipe (using PT_PIPE()) in the child process,
read the contents of the pipe from the checkpoint file and write it to
the newly created pipe.

Its fairly straightforward, except for couple of notes:

 - when we read contents of '/proc/pid/fd/read-side-fd' we drain
   the pipe such that when the checkpointed application resumes,
   it will not find any data. To fix this, we read from the
   'read-side-fd' and write it back to the 'read-side-fd' in
   addition to writing to the checkpoint file.

 - there does not seem to be a mechanism to determine the count
   of unread bytes in the file. Current implmentation assumes a
   maximum of 64K bytes (PIPE_BUFS * PAGE_SIZE on i386) and fails
   if the pipe is not fully drained.

Changelog:[v1]:

 - [Serge Hallyn]: use || instead of && in ensure_fifo_has_drained

 - [Serge Hallyn, Matt Helsley]: Use dup2() to restore fds and
  remove assumptions about order of read and write fds
  (addressed in PATCH 2/2).

Basic unit-testing done at this point (using tests/pipe.c).

TODO:
 - Additional testing (with multiple-processes and multiple-pipes)

```
 - Named-pipes
---
 cr.c |  218 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++----
 1 files changed, 206 insertions(+), 12 deletions(-)

diff --git a/cr.c b/cr.c
index c7e3332..fda1111 100644
--- a/cr.c
+++ b/cr.c
@@ -88,6 +88,11 @@ typedef struct fdinfo_t {
  char name[128];  /* file name. NULL if anonymous (pipe, socketpair) */
 } fdinfo_t;

+typedef struct fifoinfo_t {
+ int fi_fd;  /* fifo's read-side fd */
+ int fi_length;  /* number of bytes in the fifo */
+} fifofdinfo_t;
+
 typedef struct memseg_t {
  unsigned long start; /* memory segment start address */
  unsigned long end; /* memory segment end address */
@@ -499,6 +504,129 @@ out:
  return rc;
 }

+static int estimate_fifo_unread_bytes(pinfo_t *pi, int fd)
+{
+ /*
+  * Is there a way to find the number of bytes remaining to be
+  * read in a fifo ? If not, can we print it in fdinfo ?
+  *
+  * Return 64K (PIPE_BUFS * PAGE_SIZE) for now.
+  */
+ return 65536;
+}
+
+static void ensure_fifo_has_drained(char *fname, int fifo_fd)
+{
+ int rc, c;
+
+ errno = 0;
+ rc = read(fifo_fd, &c, 1);
+ if (rc != -1 || errno != EAGAIN) {
+  ERROR("FIFO '%s' not drained fully. rc %d, c %d "
+   "errno %d\n", fname, rc, c, errno);
+ }
+
+}
```

```
+
+static int save_process_fifo_info(pinfo_t *pi, int fd)
+{
+ int i;
+ int rc;
+ int nbytes;
+ int fifo_fd;
+ int pbuf_size;
+ pid_t pid = pi->pid;
+ char fname[256];
+ fdinfo_t *fi = pi->fi;
+ char  *pbuf;
+ fifofdinfo_t fifofdinfo;
+
+ write_item(fd, "FIFO", NULL, 0);
+
+ for (i = 0; i < pi->nf; i++) {
+  if (! S_ISFIFO(fi[i].mode))
+   continue;
+
+  DEBUG("FIFO fd %d (%s), flag 0x%x\n", fi[i].fdnum, fi[i].name,
+    fi[i].flag);
+
+  if (!(fi[i].flag & O_WRONLY))
+   continue;
+
+  pbuf_size = estimate_fifo_unread_bytes(pi, fd);
+
+  pbuf = (char *)malloc(pbuf_size);
+  if (!pbuf) {
+   ERROR("Unable to allocate FIFO buffer of size %d\n",
+     pbuf_size);
+  }
+  memset(pbuf, 0, pbuf_size);
+
+  sprintf(fname, "/proc/%u/fd/%u", pid, fi[i].fdnum);
+
+  /*
+   * Open O_NONBLOCK so read does not block if fifo has fewer
+   * bytes than our estimate.
+   */
+  fifo_fd = open(fname, O_RDWR|O_NONBLOCK);
+  if (fifo_fd < 0)
+   ERROR("Error %d opening FIFO '%s'\n", errno, fname);
+
+  nbytes = read(fifo_fd, pbuf, pbuf_size);
+  if (nbytes < 0) {
+   if (errno != EAGAIN) {
```

```
+    ERROR("Error %d reading FIFO '%s'\n", errno,
+      fname);
+   }
+   nbytes = 0;  /* empty fifo */
+  }
+
+  /*
+   * Ensure FIFO has been drained.
+   *
+   * TODO: If FIFO has not fully drained, our estimate of
+   *   unread-bytes is wrong. We could:
+   *
+   *   - have kernel print exact number of unread-bytes
+   *     in /proc/pid/fdinfo/<fd>
+   *
+   *   - read in contents multiple times and write multiple
+   *     fifobufs or assemble them into a single, large
+   *     buffer.
+   */
+  ensure_fifo_has_drained(fname, fifo_fd);
+
+  /*
+   * Save FIFO data to checkpoint file
+   */
+  fifofdinfo.fi_fd = fi[i].fdnum;
+  fifofdinfo.fi_length = nbytes;
+  write_item(fd, "fifofdinfo", &fifofdinfo, sizeof(fifofdinfo));
+
+  if (nbytes) {
+   write_item(fd, "fifobufs", pbuf, nbytes);
+
+   /*
+    * Restore FIFO's contents so checkpointed application
+    * won't miss a thing.
+    */
+   errno = 0;
+   rc = write(fifo_fd, pbuf, nbytes);
+   if (rc != nbytes) {
+    ERROR("Wrote-back only %d of %d bytes to FIFO, "
+      "error %d\n", rc, nbytes, errno);
+   }
+  }
+
+  close(fifo_fd);
+  free(pbuf);
+ }
+
+ write_item(fd, "END FIFO", NULL, 0);
```

```
+
+ return 0;
+}
+
 static int save_process_data(pid_t pid, int fd, lh_list_t *ptree)
 {
  char fname[256], exe[256], cwd[256], *argv, *env, *buf;
@@ -618,6 +746,8 @@ static int save_process_data(pid_t pid, int fd, lh_list_t *ptree)
  }
  write_item(fd, "END FD", NULL, 0);

+ save_process_fifo_info(pi, fd);
+
  /* sockets */
  write_item(fd, "SOCK", NULL, 0);
  for (i = 0; i < pi->ns; i++)
@@ -870,6 +1000,29 @@ int restore_fd(int fd, pid_t pid)
     }
     if (pfd != fdinfo->fdnum) t_d(PT_CLOSE(pid, pfd));
    }
+  } else if (S_ISFIFO(fdinfo->mode)) {
+   int pipefds[2] = { 0, 0 };
+
+   /*
+    * We create the pipe when we see the pipe's read-fd.
+    * Just ignore the pipe's write-fd.
+    */
+   if (fdinfo->flag == O_WRONLY)
+    continue;
+
+   DEBUG("Creating pipe for fd %d\n", fdinfo->fdnum);
+
+   t_d(PT_PIPE(pid, pipefds));
+   t_d(pipefds[0]);
+   t_d(pipefds[1]);
+
+   if (pipefds[0] != fdinfo->fdnum) {
+    DEBUG("Hmm, new pipe has fds %d, %d "
+     "Old pipe had fd %d\n", pipefds[0],
+     pipefds[1], fdinfo->fdnum); getchar();
+    exit(1);
+   }
+   DEBUG("Done creating pipefds[0] %d\n", pipefds[0]);
   }

   /*
@@ -878,20 +1031,8 @@ int restore_fd(int fd, pid_t pid)
   ret = PT_FCNTL(pid, fdinfo->fdnum, F_SETFL, fdinfo->flag);
```

```
   DEBUG("---- restore_fd() fd %d setfl flag 0x%x, ret %d\n",
     fdinfo->fdnum, fdinfo->flag, ret);
-
-
   free(fdinfo);
  }
- if (1) {
-  /* test: force pipe creation */
-  static int first = 1;
-  int pipe[2] = { 0, 0 };
-  if (! first) return 0;
-  else first = 0;
-  t_d(PT_PIPE(pid, pipe));
-  t_d(pipe[0]);
-  t_d(pipe[1]);
- }
  return 0;
 error:
  free(fdinfo);
@@ -1286,6 +1427,57 @@ int restore_sig(pid_t pid, struct sigaction *sigact, sigset_t *sigmask,
sigset_t
  return 0;
 }

+int restore_fifo(int fd, pid_t pid)
+{
+ char item[64];
+ void *buf = NULL;
+ size_t bufsz;
+ int ret;
+ int fifo_fd;
+ char fname[64];
+ int nbytes;
+ fifofdinfo_t *fifofdinfo = NULL;
+
+ for(;;) {
+  ret = read_item(fd, item, sizeof(item), &buf, &bufsz);
+  DEBUG("restore_fifo() read item '%.12s'\n", item);
+  if ITEM_IS("END FIFO")
+   break;
+  else ITEM_SET(fifofdinfo, fifofdinfo_t);
+  else if ITEM_IS("fifobufs") {
+   DEBUG("restore_fifo() bufsz %d, fi_fd %d, length %d\n",
+     bufsz, fifofdinfo->fi_fd,
+     fifofdinfo->fi_length);
+
+   if (!fifofdinfo->fi_length)
+    continue;
```

```
+
+   sprintf(fname, "/proc/%u/fd/%d", pid,
+     fifofdinfo->fi_fd);
+
+   fifo_fd = open(fname, O_WRONLY|O_NONBLOCK);
+   if (fifo_fd < 0) {
+    ERROR("Error %d opening FIFO '%s'\n", errno,
+      fname);
+   }
+
+   errno = 0;
+   nbytes = write(fifo_fd, buf, bufsz);
+   if (nbytes != bufsz) {
+    ERROR("Wrote %d of %d bytes to FIFO '%s', "
+      "errno %d\n", nbytes, bufsz,
+      fname, errno);
+   }
+   close(fifo_fd);
+  } else
+   ERROR("Unexpected item, '%s'\n", item);
+ }
+
+ DEBUG("restore_fifo() pid %d 'END FIFO'\n", pid);
+
+ return 0;
+}
+
 static int process_restart(int fd, int mode)
 {
  char item[64];
@@ -1369,6 +1561,8 @@ static int process_restart(int fd, int mode)
     ptrace_set_thread_area(npid, ldt);
    if (cwd) PT_CHDIR(npid, cwd);
    restore_fd(fd, npid);
+  } else if (ITEM_IS("FIFO")) {
+   restore_fifo(fd, npid);
   } else if (ITEM_IS("SOCK")) {
    restore_sock(fd, npid);
   } else if (ITEM_IS("SEMUNDO")) {
--
1.5.2.5
```

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

Subject: Re: [PATCH 1/2][cryo] Save/restore state of unnamed pipes
Posted by serue on Tue, 24 Jun 2008 17:38:04 GMT
View Forum Message <> Reply to Message

Quoting sukadev@us.ibm.com (sukadev@us.ibm.com):
> | + } else
> | +   ERROR("Unexpected item, '%s'\n", item);
> | + }
> | + DEBUG("restore_fifo() fd %d, len %d, got 'END FIFO'\n",
> | +   fifofdinfo->fi_fd, fifofdinfo->fi_length);
> | + return 0;
> | +}
>
> fifofdinfo can be NULL if application does not have any fifos and
> the above DEBUG() will SIGSEGV. Following updated patch fixes this.
> Thanks Serge for reporting the bug.

Thanks, Suka. This version works for both pipe and sleep testcases.

Though, interestingly, it segfaults on the -lxc kernel. It succeeds on
a recent -git from Linus' tree, though.

So I'm about to apply these on git://git.sr71.net/~hallyn/cryodev.git

thanks,
-serge

> >From bdb9d8f20abd412a46a4e3951ed084ca5690e7d2 Mon Sep 17 00:00:00 2001
> From: Sukadev Bhattiprolu <sukadev@linux.vnet.ibm.com>
> Date: Mon, 16 Jun 2008 18:41:05 -0700
> Subject: [PATCH] Save/restore state of unnamed pipes
>
> Design:
>
> Current Linux kernels provide ability to read/write contents of FIFOs
> using /proc. i.e 'cat /proc/pid/fd/read-side-fd' prints the unread data
> in the FIFO. Similarly, 'cat foo > /proc/pid/fd/read-sid-fd' appends
> the contents of 'foo' to the unread contents of the FIFO.
>
> So to save/restore the state of the pipe, a simple implementation is
> to read the from the unnamed pipe's fd and save to the checkpoint-file.
> When restoring, create a pipe (using PT_PIPE()) in the child process,
> read the contents of the pipe from the checkpoint file and write it to
> the newly created pipe.
>
> Its fairly straightforward, except for couple of notes:
>
> - when we read contents of '/proc/pid/fd/read-side-fd' we drain
>   the pipe such that when the checkpointed application resumes,

>   it will not find any data. To fix this, we read from the
>   'read-side-fd' and write it back to the 'read-side-fd' in
>   addition to writing to the checkpoint file.
>
> - there does not seem to be a mechanism to determine the count
>   of unread bytes in the file. Current implmentation assumes a
>   maximum of 64K bytes (PIPE_BUFS * PAGE_SIZE on i386) and fails
>   if the pipe is not fully drained.
>
> Changelog:[v1]:
>
> - [Serge Hallyn]: use || instead of && in ensure_fifo_has_drained
>
> - [Serge Hallyn, Matt Helsley]: Use dup2() to restore fds and
>   remove assumptions about order of read and write fds
>   (addressed in PATCH 2/2).
>
> Basic unit-testing done at this point (using tests/pipe.c).
>
> TODO:
> - Additional testing (with multiple-processes and multiple-pipes)
> - Named-pipes
> ---
> cr.c | 218 ++----
> 1 files changed, 206 insertions(+), 12 deletions(-)
>
> diff --git a/cr.c b/cr.c
> index c7e3332..fda1111 100644
> --- a/cr.c
> +++ b/cr.c
> @@ -88,6 +88,11 @@ typedef struct fdinfo_t {
>   char name[128]; /* file name. NULL if anonymous (pipe, socketpair) */
> } fdinfo_t;
>
> +typedef struct fifoinfo_t {
> + int fi_fd; /* fifo's read-side fd */
> + int fi_length; /* number of bytes in the fifo */
> +} fifofdinfo_t;
> +
> typedef struct memseg_t {
>   unsigned long start; /* memory segment start address */
>   unsigned long end; /* memory segment end address */
> @@ -499,6 +504,129 @@ out:
>   return rc;
> }
>
> +static int estimate_fifo_unread_bytes(pinfo_t *pi, int fd)
> +{

```
> + /*
> +  * Is there a way to find the number of bytes remaining to be
> +  * read in a fifo ? If not, can we print it in fdinfo ?
> +  *
> +  * Return 64K (PIPE_BUFS * PAGE_SIZE) for now.
> +  */
> + return 65536;
> +}
> +
> +static void ensure_fifo_has_drained(char *fname, int fifo_fd)
> +{
> + int rc, c;
> +
> + errno = 0;
> + rc = read(fifo_fd, &c, 1);
> + if (rc != -1 || errno != EAGAIN) {
> +  ERROR("FIFO '%s' not drained fully. rc %d, c %d "
> +   "errno %d\n", fname, rc, c, errno);
> + }
> +
> +}
> +
> +static int save_process_fifo_info(pinfo_t *pi, int fd)
> +{
> + int i;
> + int rc;
> + int nbytes;
> + int fifo_fd;
> + int pbuf_size;
> + pid_t pid = pi->pid;
> + char fname[256];
> + fdinfo_t *fi = pi->fi;
> + char  *pbuf;
> + fifofdinfo_t fifofdinfo;
> +
> + write_item(fd, "FIFO", NULL, 0);
> +
> + for (i = 0; i < pi->nf; i++) {
> +  if (! S_ISFIFO(fi[i].mode))
> +   continue;
> +
> +  DEBUG("FIFO fd %d (%s), flag 0x%x\n", fi[i].fdnum, fi[i].name,
> +    fi[i].flag);
> +
> +  if (!(fi[i].flag & O_WRONLY))
> +   continue;
> +
> +  pbuf_size = estimate_fifo_unread_bytes(pi, fd);
```

```
> +
> +  pbuf = (char *)malloc(pbuf_size);
> +  if (!pbuf) {
> +   ERROR("Unable to allocate FIFO buffer of size %d\n",
> +     pbuf_size);
> +  }
> +  memset(pbuf, 0, pbuf_size);
> +
> +  sprintf(fname, "/proc/%u/fd/%u", pid, fi[i].fdnum);
> +
> +  /*
> +   * Open O_NONBLOCK so read does not block if fifo has fewer
> +   * bytes than our estimate.
> +   */
> +  fifo_fd = open(fname, O_RDWR|O_NONBLOCK);
> +  if (fifo_fd < 0)
> +   ERROR("Error %d opening FIFO '%s'\n", errno, fname);
> +
> +  nbytes = read(fifo_fd, pbuf, pbuf_size);
> +  if (nbytes < 0) {
> +   if (errno != EAGAIN) {
> +    ERROR("Error %d reading FIFO '%s'\n", errno,
> +      fname);
> +   }
> +   nbytes = 0;  /* empty fifo */
> +  }
> +
> +  /*
> +   * Ensure FIFO has been drained.
> +   *
> +   * TODO: If FIFO has not fully drained, our estimate of
> +   *   unread-bytes is wrong. We could:
> +   *
> +   *   - have kernel print exact number of unread-bytes
> +   *     in /proc/pid/fdinfo/<fd>
> +   *
> +   *   - read in contents multiple times and write multiple
> +   *     fifobufs or assemble them into a single, large
> +   *     buffer.
> +   */
> +  ensure_fifo_has_drained(fname, fifo_fd);
> +
> +  /*
> +   * Save FIFO data to checkpoint file
> +   */
> +  fifofdinfo.fi_fd = fi[i].fdnum;
> +  fifofdinfo.fi_length = nbytes;
> +  write_item(fd, "fifofdinfo", &fifofdinfo, sizeof(fifofdinfo));
```

```
> +
> +  if (nbytes) {
> +   write_item(fd, "fifobufs", pbuf, nbytes);
> +
> +   /*
> +    * Restore FIFO's contents so checkpointed application
> +    * won't miss a thing.
> +    */
> +   errno = 0;
> +   rc = write(fifo_fd, pbuf, nbytes);
> +   if (rc != nbytes) {
> +    ERROR("Wrote-back only %d of %d bytes to FIFO, "
> +      "error %d\n", rc, nbytes, errno);
> +   }
> +  }
> +
> +  close(fifo_fd);
> +  free(pbuf);
> + }
> +
> + write_item(fd, "END FIFO", NULL, 0);
> +
> + return 0;
> +}
> +
>  static int save_process_data(pid_t pid, int fd, lh_list_t *ptree)
>  {
>   char fname[256], exe[256], cwd[256], *argv, *env, *buf;
> @@ -618,6 +746,8 @@ static int save_process_data(pid_t pid, int fd, lh_list_t *ptree)
>   }
>   write_item(fd, "END FD", NULL, 0);
>
> + save_process_fifo_info(pi, fd);
> +
>   /* sockets */
>   write_item(fd, "SOCK", NULL, 0);
>   for (i = 0; i < pi->ns; i++)
> @@ -870,6 +1000,29 @@ int restore_fd(int fd, pid_t pid)
>      }
>      if (pfd != fdinfo->fdnum) t_d(PT_CLOSE(pid, pfd));
>     }
> +  } else if (S_ISFIFO(fdinfo->mode)) {
> +   int pipefds[2] = { 0, 0 };
> +
> +   /*
> +    * We create the pipe when we see the pipe's read-fd.
> +    * Just ignore the pipe's write-fd.
> +    */
```

```
> +   if (fdinfo->flag == O_WRONLY)
> +    continue;
> +
> +   DEBUG("Creating pipe for fd %d\n", fdinfo->fdnum);
> +
> +   t_d(PT_PIPE(pid, pipefds));
> +   t_d(pipefds[0]);
> +   t_d(pipefds[1]);
> +
> +   if (pipefds[0] != fdinfo->fdnum) {
> +    DEBUG("Hmm, new pipe has fds %d, %d "
> +     "Old pipe had fd %d\n", pipefds[0],
> +     pipefds[1], fdinfo->fdnum); getchar();
> +    exit(1);
> +   }
> +   DEBUG("Done creating pipefds[0] %d\n", pipefds[0]);
>    }
>
>    /*
> @@ -878,20 +1031,8 @@ int restore_fd(int fd, pid_t pid)
>    ret = PT_FCNTL(pid, fdinfo->fdnum, F_SETFL, fdinfo->flag);
>    DEBUG("---- restore_fd() fd %d setfl flag 0x%x, ret %d\n",
>      fdinfo->fdnum, fdinfo->flag, ret);
> -
> -
>    free(fdinfo);
>   }
> - if (1) {
> -  /* test: force pipe creation */
> -  static int first = 1;
> -  int pipe[2] = { 0, 0 };
> -  if (! first) return 0;
> -  else first = 0;
> -  t_d(PT_PIPE(pid, pipe));
> -  t_d(pipe[0]);
> -  t_d(pipe[1]);
> - }
>   return 0;
>  error:
>   free(fdinfo);
> @@ -1286,6 +1427,57 @@ int restore_sig(pid_t pid, struct sigaction *sigact, sigset_t *sigmask,
sigset_t
>   return 0;
>  }
>
> +int restore_fifo(int fd, pid_t pid)
> +{
> + char item[64];
```

```
> + void *buf = NULL;
> + size_t bufsz;
> + int ret;
> + int fifo_fd;
> + char fname[64];
> + int nbytes;
> + fifofdinfo_t *fifofdinfo = NULL;
> +
> + for(;;) {
> +  ret = read_item(fd, item, sizeof(item), &buf, &bufsz);
> +  DEBUG("restore_fifo() read item '%.12s'\n", item);
> +  if ITEM_IS("END FIFO")
> +   break;
> +  else ITEM_SET(fifofdinfo, fifofdinfo_t);
> +  else if ITEM_IS("fifobufs") {
> +   DEBUG("restore_fifo() bufsz %d, fi_fd %d, length %d\n",
> +     bufsz, fifofdinfo->fi_fd,
> +     fifofdinfo->fi_length);
> +
> +   if (!fifofdinfo->fi_length)
> +    continue;
> +
> +   sprintf(fname, "/proc/%u/fd/%d", pid,
> +     fifofdinfo->fi_fd);
> +
> +   fifo_fd = open(fname, O_WRONLY|O_NONBLOCK);
> +   if (fifo_fd < 0) {
> +    ERROR("Error %d opening FIFO '%s'\n", errno,
> +      fname);
> +   }
> +
> +   errno = 0;
> +   nbytes = write(fifo_fd, buf, bufsz);
> +   if (nbytes != bufsz) {
> +    ERROR("Wrote %d of %d bytes to FIFO '%s', "
> +      "errno %d\n", nbytes, bufsz,
> +      fname, errno);
> +   }
> +   close(fifo_fd);
> +  } else
> +   ERROR("Unexpected item, '%s'\n", item);
> + }
> +
> + DEBUG("restore_fifo() pid %d 'END FIFO'\n", pid);
> +
> + return 0;
> +}
> +
```

> static int process_restart(int fd, int mode)
> {
> char item[64];
> @@ -1369,6 +1561,8 @@ static int process_restart(int fd, int mode)
> ptrace_set_thread_area(npid, ldt);
> if (cwd) PT_CHDIR(npid, cwd);
> restore_fd(fd, npid);
> + } else if (ITEM_IS("FIFO")) {
> + restore_fifo(fd, npid);
> } else if (ITEM_IS("SOCK")) {
> restore_sock(fd, npid);
> } else if (ITEM_IS("SEMUNDO")) {
> --
> 1.5.2.5

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

---

Subject: Re: [PATCH 1/2][cryo] Save/restore state of unnamed pipes
Posted by Sukadev Bhattiprolu on Tue, 24 Jun 2008 17:58:53 GMT
View Forum Message <> Reply to Message

Serge E. Hallyn [serue@us.ibm.com] wrote:
| Quoting sukadev@us.ibm.com (sukadev@us.ibm.com):
| > | + } else
| > | + ERROR("Unexpected item, '%s'\n", item);
| > | + }
| > | + DEBUG("restore_fifo() fd %d, len %d, got 'END FIFO'\n",
| > | + fifofdinfo->fi_fd, fifofdinfo->fi_length);
| > | + return 0;
| > | +}
| >
| > fifofdinfo can be NULL if application does not have any fifos and
| > the above DEBUG() will SIGSEGV. Following updated patch fixes this.
| > Thanks Serge for reporting the bug.
|
| Thanks, Suka. This version works for both pipe and sleep testcases.
|
| Though, interestingly, it segfaults on the -lxc kernel. It succeeds on
| a recent -git from Linus' tree, though.

I have been testing 2.6.25-mm1.

I run into the semun_undo underflow bug with -lxc. Is that the same
bug you are running into ?

|
| So I'm about to apply these on git://git.sr71.net/~hallyn/cryodev.git
|

Thanks,

Suka

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

---

Subject: Re: [PATCH 1/2][cryo] Save/restore state of unnamed pipes
Posted by serue on Tue, 24 Jun 2008 19:36:59 GMT
View Forum Message <> Reply to Message

Quoting sukadev@us.ibm.com (sukadev@us.ibm.com):
> Serge E. Hallyn [serue@us.ibm.com] wrote:
> | Quoting sukadev@us.ibm.com (sukadev@us.ibm.com):
> | > | + } else
> | > | +   ERROR("Unexpected item, '%s'\n", item);
> | > | + }
> | > | + DEBUG("restore_fifo() fd %d, len %d, got 'END FIFO'\n",
> | > | +   fifofdinfo->fi_fd, fifofdinfo->fi_length);
> | > | + return 0;
> | > | +}
> | >
> | > fifofdinfo can be NULL if application does not have any fifos and
> | > the above DEBUG() will SIGSEGV. Following updated patch fixes this.
> | > Thanks Serge for reporting the bug.
> |
> | Thanks, Suka. This version works for both pipe and sleep testcases.
> |
> | Though, interestingly, it segfaults on the -lxc kernel. It succeeds on
> | a recent -git from Linus' tree, though.
>
> I have been testing 2.6.25-mm1.
>
> I run into the semun_undo underflow bug with -lxc. Is that the same
> bug you are running into ?

I'm not sure... Not looking into it too deeply. Hopefully Kathy is
coming out with a new -lxc in the next few days, and if it still causes
me problems then I'll really look into it.

-serge

_____

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