Subject: [RFC][PATCH][cryo] Save/restore state of unnamed pipes
Posted by Sukadev Bhattiprolu on Tue, 17 Jun 2008 21:29:50 GMT
View Forum Message <> Reply to Message

>From fd13986de32af31621b1badbcf7bfb5626648e0e 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.

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

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

Signed-off-by: Sukadev Bhattiprolu <sukadev@us.ibm.com>
---
 cr.c | 215 ++----
 1 files changed, 203 insertions(+), 12 deletions(-)

diff --git a/cr.c b/cr.c
index 5163a3d..0cb9774 100644

```
--- a/cr.c
+++ b/cr.c
@@ -84,6 +84,11 @@ 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 */
@@ -468,6 +473,128 @@ 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;
+
+ 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;
@@ -587,6 +714,8 @@ 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++)
@@ -839,6 +968,29 @@ 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]);
   }

   /*
@@ -847,20 +999,8 @@ 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);
@@ -1231,6 +1371,55 @@ 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("Error %d writing to FIFO '%s'\n",
+      errno, fname);
+   }
+   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];
@@ -1314,6 +1503,8 @@ 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: [RFC][PATCH][cryo] Save/restore state of unnamed pipes
Posted by serue on Tue, 17 Jun 2008 22:30:39 GMT
View Forum Message <> Reply to Message

Quoting sukadev@us.ibm.com (sukadev@us.ibm.com):
>
> >From fd13986de32af31621b1badbcf7bfb5626648e0e 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.
>
> Basic unit-testing done at this point (using tests/pipe.c).
>
> TODO:
> - Additional testing (with multiple-processes and multiple-pipes)
> - Named-pipes
>
> Signed-off-by: Sukadev Bhattiprolu <sukadev@us.ibm.com>
> ---
> cr.c | 215 ++----
> 1 files changed, 203 insertions(+), 12 deletions(-)
>
> diff --git a/cr.c b/cr.c
> index 5163a3d..0cb9774 100644
> --- a/cr.c
> +++ b/cr.c
> @@ -84,6 +84,11 @@ 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 */
> @@ -468,6 +473,128 @@ 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;
> +
> + rc = read(fifo_fd, &c, 1);
> + if (rc != -1 && errno != EAGAIN) {

 Won't errno only be set if rc == -1? Did you mean || here?

> + 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;
> @@ -587,6 +714,8 @@ 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++)
> @@ -839,6 +968,29 @@ 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();
```

Can you explain what you're doing here? I would have expected you to dup2() to get back the correct fd, so maybe I'm missing something...

```
> +    exit(1);
> +   }
> +   DEBUG("Done creating pipefds[0] %d\n", pipefds[0]);
>    }
>
>    /*
> @@ -847,20 +999,8 @@ 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);
> @@ -1231,6 +1371,55 @@ 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("Error %d writing to FIFO '%s'\n",
> +      errno, fname);
> +   }
> +   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];
> @@ -1314,6 +1503,8 @@ 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: [RFC][PATCH][cryo] Save/restore state of unnamed pipes
Posted by Sukadev Bhattiprolu on Tue, 17 Jun 2008 22:55:57 GMT
View Forum Message <> Reply to Message

Serge E. Hallyn [serue@us.ibm.com] wrote:

<snip>

```
| > +
| > + rc = read(fifo_fd, &c, 1);
| > + if (rc != -1 && errno != EAGAIN) {
|
|  Won't errno only be set if rc == -1?  Did you mean || here?
```

Yes I meant ||. I also had 'errno = 0' before the read, but seem
to have deleted it when I moved code around.

<snip>

```
| > +  } 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();
|
| Can you explain what you're doing here?  I would have expected you to
| dup2() to get back the correct fd, so maybe I'm missing something...
```

You are right, I should use dup2() here.

Will send an updated patch.

Thanks,

Suka

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

---

Subject: Re: [RFC][PATCH][cryo] Save/restore state of unnamed pipes
Posted by Matt Helsley on Tue, 17 Jun 2008 23:31:12 GMT
View Forum Message <> Reply to Message

On Tue, 2008-06-17 at 17:30 -0500, Serge E. Hallyn wrote:

> Quoting sukadev@us.ibm.com (sukadev@us.ibm.com):
> >
> > >From fd13986de32af31621b1badbcf7bfb5626648e0e 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.
> >
> > Basic unit-testing done at this point (using tests/pipe.c).
> >
> > TODO:
> > - Additional testing (with multiple-processes and multiple-pipes)
> > - Named-pipes
> >
> > Signed-off-by: Sukadev Bhattiprolu <sukadev@us.ibm.com>
> > ---
> > cr.c | 215
++++++

> > +++ b/cr.c
> > @@ -84,6 +84,11 @@ 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 */
> > @@ -468,6 +473,128 @@ 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;
> > +
> > + rc = read(fifo_fd, &c, 1);
> > + if (rc != -1 && errno != EAGAIN) {
>
> Won't errno only be set if rc == -1? Did you mean || here?
>
> > + 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;
> > @@ -587,6 +714,8 @@ 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++)
> > @@ -839,6 +968,29 @@ 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();
>
> Can you explain what you're doing here?  I would have expected you to
> dup2() to get back the correct fd, so maybe I'm missing something...
```

Yes, I agree.

Though I wonder if it's possible that the two fds returned could be swapped during restart. Does anyone know if POSIX makes any guarantees about the numeric relationship between pipefds[0] and pipefds[1] (like "pipefds[0] < pipefds[1]")? If there are no guarantees then it may be possible for a simple dup2() to break the new pipe. Suppose, for example, that the original pipe used fds 4 and 5 in elements 0 and 1 of the fd array respectively and then we restart:

```
t_d(PT_PIPE(pid, pipefds)); /* returns 5 and 4 in elements 0 and 1 */
if (pipefds[0] != fdinfo->fdnum)
 PT_DUP2(pid, pipefds[0], fdinfo->fdnum); /* accidentally closes
            pipefds[1] */
```

I don't see anything in the pipe man page, at least, that suggests we can safely assume pipefds[0] < pipefds[1].

The solution could be to use "trampoline" fds. Suppose last_fd is the largest fd that exists in the final checkpointed/restarting application. We could do (Skipping the PT_FUNC "notation" for clarity):

```
pipe(pipefds); /* returns 5 and 4 in elements 0 and 1 */
/* use fds after last_fd as trampolines for fds we want to create */
dup2(pipefds[0], last_fd + 1);
dup2(pipefds[1], last_fd + 2);
close(pipefds[0]);
close(pipefds[1]);
dup2(last_fd + 1, <orig pipefd[0]>);
dup2(last_fd + 2, <orig pipefd[1]>);
close(last_fd + 1);
close(last_fd + 2);
```

Which is alot more code but should work no matter which fds we get back from pipe(). Of course this assumes the checkpointed application hasn't used all of its fds. :(

Cheers,
-Matt

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

---

Subject: Re: [RFC][PATCH][cryo] Save/restore state of unnamed pipes
Posted by Sukadev Bhattiprolu on Wed, 18 Jun 2008 00:32:14 GMT
View Forum Message <> Reply to Message

Matt Helsley [matthltc@us.ibm.com] wrote:
|
| On Tue, 2008-06-17 at 17:30 -0500, Serge E. Hallyn wrote:
| > Quoting sukadev@us.ibm.com (sukadev@us.ibm.com):

| > >
| > > >From fd13986de32af31621b1badbcf7bfb5626648e0e 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.
| > >
| > > Basic unit-testing done at this point (using tests/pipe.c).
| > >
| > > TODO:
| > > - Additional testing (with multiple-processes and multiple-pipes)
| > > - Named-pipes
| > >
| > > Signed-off-by: Sukadev Bhattiprolu <sukadev@us.ibm.com>
| > > ---
| > > cr.c | 215
++++++

```
| > > @@ -84,6 +84,11 @@ 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 */
| > > @@ -468,6 +473,128 @@ 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;
| > > +
| > > + rc = read(fifo_fd, &c, 1);
| > > + if (rc != -1 && errno != EAGAIN) {
| >
| >  Won't errno only be set if rc == -1?  Did you mean || here?
| >
| > > +  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;
```

```
| > > @@ -587,6 +714,8 @@ 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++)
| > > @@ -839,6 +968,29 @@ 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();
| >
| > Can you explain what you're doing here?  I would have expected you to
| > dup2() to get back the correct fd, so maybe I'm missing something...
|
| Yes, I agree.
|
|  Though I wonder if it's possible that the two fds returned could be
| swapped during restart. Does anyone know if POSIX makes any guarantees
| about the numeric relationship between pipefds[0] and pipefds[1] (like
| "pipefds[0] < pipefds[1]")? If there are no guarantees then it may be
| possible for a simple dup2() to break the new pipe. Suppose, for
| example, that the original pipe used fds 4 and 5 in elements 0 and 1 of
| the fd array respectively and then we restart:
```

Yes, I was just thinking about this assumption and was wondering if

I could find the peer fd by walking the list of fds in /proc/pid/fd
and doing an lstat() and comparing the inode numbers.

Then save the peer fd in fdinf. On restore, we could create the
pipe and dup2() both read and write-side fds.

```
|
|
|       t_d(PT_PIPE(pid, pipefds)); /* returns 5 and 4 in elements 0 and 1 */
|       if (pipefds[0] != fdinfo->fdnum)
|        PT_DUP2(pid, pipefds[0], fdinfo->fdnum); /* accidentally closes
|                  pipefds[1] */
|
|
| I don't see anything in the pipe man page, at least, that suggests we
| can safely assume pipefds[0] < pipefds[1].
|
|  The solution could be to use "trampoline" fds. Suppose last_fd is the
| largest fd that exists in the final checkpointed/restarting application.
| We could do (Skipping the PT_FUNC "notation" for clarity):

|
|
|       pipe(pipefds); /* returns 5 and 4 in elements 0 and 1 */
|       /* use fds after last_fd as trampolines for fds we want to create */
|       dup2(pipefds[0], last_fd + 1);
|       dup2(pipefds[1], last_fd + 2);
|       close(pipefds[0]);
|       close(pipefds[1]);
|       dup2(last_fd + 1, <orig pipefd[0]>);
|       dup2(last_fd + 2, <orig pipefd[1]>);
|       close(last_fd + 1);
|       close(last_fd + 2);
|
|
| Which is alot more code but should work no matter which fds we get back
| from pipe(). Of course this assumes the checkpointed application hasn't
| used all of its fds. :(
|
```

This sounds like a good idea too, but we could use any fd that has not
yet been used in the restart-process right ? It would break if all fds
are used AND one of the pipe fds is the very last one :-)

In that case, we could maybe create all pipe fds first and then go
back to creating the rest ?

_____
Containers mailing list

Subject: Re: [RFC][PATCH][cryo] Save/restore state of unnamed pipes
Posted by Matt Helsley on Wed, 18 Jun 2008 02:04:06 GMT
View Forum Message <> Reply to Message

On Tue, 2008-06-17 at 17:32 -0700, sukadev@us.ibm.com wrote:
> Matt Helsley [matthltc@us.ibm.com] wrote:
> |
> | On Tue, 2008-06-17 at 17:30 -0500, Serge E. Hallyn wrote:
> | > Quoting sukadev@us.ibm.com (sukadev@us.ibm.com):
> | > >
> | > > >From fd13986de32af31621b1badbcf7bfb5626648e0e 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.
> | > >
> | > > Basic unit-testing done at this point (using tests/pipe.c).
> | > >
> | > > TODO:

> | > > - Additional testing (with multiple-processes and multiple-pipes)
> | > > - Named-pipes
> | > >
> | > > Signed-off-by: Sukadev Bhattiprolu <sukadev@us.ibm.com>
> | > > ---
> | > > cr.c | 215
++++++

```
> | >
> | > > +  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;
> | > > @@ -587,6 +714,8 @@ 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++)
> | > > @@ -839,6 +968,29 @@ 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();
> | >
> | > Can you explain what you're doing here?  I would have expected you to
```

> | > dup2() to get back the correct fd, so maybe I'm missing something...
> |
> | Yes, I agree.
> |
> | Though I wonder if it's possible that the two fds returned could be
> | swapped during restart. Does anyone know if POSIX makes any guarantees
> | about the numeric relationship between pipefds[0] and pipefds[1] (like
> | "pipefds[0] < pipefds[1]")? If there are no guarantees then it may be
> | possible for a simple dup2() to break the new pipe. Suppose, for
> | example, that the original pipe used fds 4 and 5 in elements 0 and 1 of
> | the fd array respectively and then we restart:
>
> Yes, I was just thinking about this assumption and was wondering if
> I could find the peer fd by walking the list of fds in /proc/pid/fd
> and doing an lstat() and comparing the inode numbers.
>
> Then save the peer fd in fdinf. On restore, we could create the
> pipe and dup2() both read and write-side fds.
>
> |
> |
> | t_d(PT_PIPE(pid, pipefds)); /* returns 5 and 4 in elements 0 and 1 */
> | if (pipefds[0] != fdinfo->fdnum)
> | PT_DUP2(pid, pipefds[0], fdinfo->fdnum); /* accidentally closes
> | pipefds[1] */
> |
> |
> | I don't see anything in the pipe man page, at least, that suggests we
> | can safely assume pipefds[0] < pipefds[1].
> |
> | The solution could be to use "trampoline" fds. Suppose last_fd is the
> | largest fd that exists in the final checkpointed/restarting application.
> | We could do (Skipping the PT_FUNC "notation" for clarity):
>
> |
> |
> | pipe(pipefds); /* returns 5 and 4 in elements 0 and 1 */
> | /* use fds after last_fd as trampolines for fds we want to create */
> | dup2(pipefds[0], last_fd + 1);
> | dup2(pipefds[1], last_fd + 2);
> | close(pipefds[0]);
> | close(pipefds[1]);
> | dup2(last_fd + 1, <orig pipefd[0]>);
> | dup2(last_fd + 2, <orig pipefd[1]>);
> | close(last_fd + 1);
> | close(last_fd + 2);
> |
> |

> | Which is alot more code but should work no matter which fds we get back
> | from pipe(). Of course this assumes the checkpointed application hasn't
> | used all of its fds. :(
> |
>
> This sounds like a good idea too, but we could use any fd that has not
> yet been used in the restart-process right ? It would break if all fds

Yes, but we don't know which fd is available unless we allocate it
without dup2(). Here's how it could be done without last_fd (again,
dropping PT_FUNC notation):

```
/*
 * Move fds from src to dest. Useful for correctly "moving" pipe fds and
 * other cases where we have a small number of fds to move to their
 * original fd.
 *
 * Assumes dest_fds and src_fds are of the same, small length since
 * this is O(num_fds^2).
 *
 * If num_fds == 1 then use plain dup2().
 *
 * Use this in place of multiple dup2() calls (num_fds > 1) unless you are
 * absolutely certain the set of dest fds do not intersect the set of src fds.
 * Does NOT magically prevent you from accidentally clobbering fds outside the
 * src_fds array.
 */
void move_fds(int *dest_fds, int *src_fds, const unsigned int num_fds)
{
 int i;
 unsigned int num_moved = 0;

 for (i = 0; i < num_fds; i++) {
  int j;

  if (src_fds[i] == dest_fds[i])
   continue; /* nothing to be done */

  /* src fd != dest fd so we need to perform:
   dup2(src fd, dest fd);
     but dup2() closes dest fd if it already exists.
     This means we could accidentally close one of
     the src fds. Avoid this by searching for any
     src fd == dest fd and dup()'ing src fd to
     a different fd so we can use the dest fd.
   */
  for (j = i + 1; j < num_fds; j++) /* This makes us O(N^2) */
   if (dest_fds[i] == src_fds[j])
```

```
    /*
     * we're using an fd for something
     * else already -- we need a trampoline
     */
    break;

  if (j >= num_fds)
   /* dup2() is safe: dest fd is unused by all src fds */
   dup2(src_fds[i], dest_fds[i]);
  else {
   int new_fd;

   /* The dest fd is in use by src_fds[j]. Use a
      new fd for the src fd */
   new_fd = dup(src_fds[j]);
   close(src_fds[j]);
   src_fds[j] = new_fd;
   dup2(src_fds[i], dest_fds[i]);
  }
  close(src_fds[i]);
 }
}

move_fds(oldpipefds, pipefds, 2);
```

This means we need at least (max(num_fds) + 1) unused fds to be able to restart (likely: 3).

One thing I liked about last_fd is it would show us when we've accidentaly leaked an fd into the restarted task -- just look for any fd greater than last_fd before restarting.

> are used AND one of the pipe fds is the very last one :-)
>
> In that case, we could maybe create all pipe fds first and then go
> back to creating the rest ?

Seems reasonable to me.

Cheers,
 -Matt

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

Subject: Re: [RFC][PATCH][cryo] Save/restore state of unnamed pipes
Posted by Sukadev Bhattiprolu on Wed, 18 Jun 2008 18:00:25 GMT
View Forum Message <> Reply to Message

Matt Helsley [matthltc@us.ibm.com] wrote:

| > |
| > |
| > |	pipe(pipefds); /* returns 5 and 4 in elements 0 and 1 */
| > |	/* use fds after last_fd as trampolines for fds we want to create */
| > |	dup2(pipefds[0], last_fd + 1);
| > |	dup2(pipefds[1], last_fd + 2);
| > |	close(pipefds[0]);
| > |	close(pipefds[1]);
| > |	dup2(last_fd + 1, <orig pipefd[0]>);
| > |	dup2(last_fd + 2, <orig pipefd[1]>);
| > |	close(last_fd + 1);
| > |	close(last_fd + 2);
| > |
| > |
| > | Which is alot more code but should work no matter which fds we get back
| > | from pipe(). Of course this assumes the checkpointed application hasn't
| > | used all of its fds. :(
| > |
| >
| > This sounds like a good idea too, but we could use any fd that has not
| > yet been used in the restart-process right ? It would break if all fds
|
| Yes, but we don't know which fd is available unless we allocate it
| without dup2(). Here's how it could be done without last_fd (again,
| dropping PT_FUNC notation):
|
| /*
| * Move fds from src to dest. Useful for correctly "moving" pipe fds and
| * other cases where we have a small number of fds to move to their
| * original fd.
| *
| * Assumes dest_fds and src_fds are of the same, small length since
| * this is O(num_fds^2).
| *
| * If num_fds == 1 then use plain dup2().
| *
| * Use this in place of multiple dup2() calls (num_fds > 1) unless you are
| * absolutely certain the set of dest fds do not intersect the set of src fds.
| * Does NOT magically prevent you from accidentally clobbering fds outside the
| * src_fds array.
| */
| void move_fds(int *dest_fds, int *src_fds, const unsigned int num_fds)
| {

```
|  int i;
|  unsigned int num_moved = 0;
|
|  for (i = 0; i < num_fds; i++) {
|   int j;
|
|   if (src_fds[i] == dest_fds[i])
|    continue; /* nothing to be done */
|
|   /* src fd != dest fd so we need to perform:
|    dup2(src fd, dest fd);
|      but dup2() closes dest fd if it already exists.
|      This means we could accidentally close one of
|      the src fds. Avoid this by searching for any
|      src fd == dest fd and dup()'ing src fd to
|      a different fd so we can use the dest fd.
|    */
|   for (j = i + 1; j < num_fds; j++) /* This makes us O(N^2) */
|    if (dest_fds[i] == src_fds[j])
|     /*
|      * we're using an fd for something
|      * else already -- we need a trampoline
|      */
```

So let me rephrase the problem.

Suppose the checkpointed application was using fds in following
"orig-fd-set"

{ [0..10], 18, 27 }

where 18 and 27 are part of a pipe. For simplicity lets assume that
18 is the read-side-fd.

We checkpointed this application and are now trying to restart it.

In the restarted application, we would call

dup2(fd1, fd2),

where 'fd1' is some new, random fd and 'fd2' is an fd in 'orig-fd-set'
(say fd2 = 18).

IIUC, there is a risk here of 'fd2' being closed accidentally while
it is in use.

But, the only way I can see 'fd2' being in use in the restarted process
is if _cryo_ opened some file _during_ restart and did not close. I ran

into this early on with the randomize_va_space file (which was easily
fixed).

Would cryo need to keep one or more temporary/debug files open in the
restarted process (i.e files that are not in the 'orig-fd-set').

If cryo does, then maybe it could open such files:

 - after clone() (so files are not open in restarted process), or

 - find the last_fd used and dup2() to that fd, leaving the
   'orig-fd-set' all open/available for restarted process

For debug, before each 'dup2(fd1, fd2)' we could 'fstat(fd2, &buf)'
to ensure 'fd2' is not in use and error out if it is.

Thanks for your comments. I will look at your code in more detail.

Suka

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

---

Subject: Re: [RFC][PATCH][cryo] Save/restore state of unnamed pipes
Posted by Matt Helsley on Wed, 18 Jun 2008 19:57:35 GMT
View Forum Message <> Reply to Message

On Wed, 2008-06-18 at 11:00 -0700, sukadev@us.ibm.com wrote:
> Matt Helsley [matthltc@us.ibm.com] wrote:
>
> | > |
> | > |
> | > | pipe(pipefds); /* returns 5 and 4 in elements 0 and 1 */
> | > | /* use fds after last_fd as trampolines for fds we want to create */
> | > | dup2(pipefds[0], last_fd + 1);
> | > | dup2(pipefds[1], last_fd + 2);
> | > | close(pipefds[0]);
> | > | close(pipefds[1]);
> | > | dup2(last_fd + 1, <orig pipefd[0]>);
> | > | dup2(last_fd + 2, <orig pipefd[1]>);
> | > | close(last_fd + 1);
> | > | close(last_fd + 2);
> | > |
> | > |
> | > | Which is alot more code but should work no matter which fds we get back
> | > | from pipe(). Of course this assumes the checkpointed application hasn't

> | > | used all of its fds. :(
> | > |
> | >
> | > This sounds like a good idea too, but we could use any fd that has not
> | > yet been used in the restart-process right ? It would break if all fds
> |
> | Yes, but we don't know which fd is available unless we allocate it
> | without dup2(). Here's how it could be done without last_fd (again,
> | dropping PT_FUNC notation):
> |
> | /*
> | * Move fds from src to dest. Useful for correctly "moving" pipe fds and
> | * other cases where we have a small number of fds to move to their
> | * original fd.
> | *
> | * Assumes dest_fds and src_fds are of the same, small length since
> | * this is O(num_fds^2).
> | *
> | * If num_fds == 1 then use plain dup2().
> | *
> | * Use this in place of multiple dup2() calls (num_fds > 1) unless you are
> | * absolutely certain the set of dest fds do not intersect the set of src fds.
> | * Does NOT magically prevent you from accidentally clobbering fds outside the
> | * src_fds array.
> | */
> | void move_fds(int *dest_fds, int *src_fds, const unsigned int num_fds)
> | {
> | int i;
> | unsigned int num_moved = 0;
> |
> | for (i = 0; i < num_fds; i++) {
> | int j;
> |
> | if (src_fds[i] == dest_fds[i])
> | continue; /* nothing to be done */
> |
> | /* src fd != dest fd so we need to perform:
> | dup2(src fd, dest fd);
> | but dup2() closes dest fd if it already exists.
> | This means we could accidentally close one of
> | the src fds. Avoid this by searching for any
> | src fd == dest fd and dup()'ing src fd to
> | a different fd so we can use the dest fd.
> | */
> | for (j = i + 1; j < num_fds; j++) /* This makes us O(N^2) */
> | if (dest_fds[i] == src_fds[j])
> | /*
> | * we're using an fd for something

> | * else already -- we need a trampoline
> | */
>
> So let me rephrase the problem.
>
> Suppose the checkpointed application was using fds in following
> "orig-fd-set"
>
> { [0..10], 18, 27 }
>
> where 18 and 27 are part of a pipe. For simplicity lets assume that
> 18 is the read-side-fd.

so orig_pipefd[0] == 18
and orig_pipefd[1] == 27

> We checkpointed this application and are now trying to restart it.
>
> In the restarted application, we would call
>
> dup2(fd1, fd2),
>
> where 'fd1' is some new, random fd and 'fd2' is an fd in 'orig-fd-set'
                        ^^^^^^ Even if they were truly random, this
does not preclude fd1 from having the same value as an fd in the
remaining orig-fd-set -- such as one of the two we're about to try and
restart with pipe().

> (say fd2 = 18).

fd1 = restarted_pipefd[0]
fd2 = restarted_pipefd[1]

In my example fd1 == 27 and fd2 == 18

> IIUC, there is a risk here of 'fd2' being closed accidentally while
> it is in use.

 Yes, that's the risk.

> But, the only way I can see 'fd2' being in use in the restarted process
> is if _cryo_ opened some file _during_ restart and did not close. I ran

 Both file descriptors returned from pipe() are in use during restart
and closing one of them would not be proper. Cryo hasn't "forgotten" to
close one of them -- cryo needs to dup2() both of them to their
"destination" fds. But if they have been swapped or if just one is the
"destination" of the other then you could end up with a broken pipe.

> into this early on with the randomize_va_space file (which was easily
> fixed).

This logic only works if cryo only has one new fd at a time. However that's not possible with pipe(). Or socketpair(). In those cases one of the two new fds could be the "destination" fd for the other. In that case dup2() will kindly close it for you and break your new pipe/socketpair! :)

That's why I asked if POSIX guarantees the read side file descriptor is always less than the write side. If it does then the numbers can't be swapped and maybe using your assumption that we don't have any other fds accidentally left open ensures dup2() will be safe.

> Would cryo need to keep one or more temporary/debug files open in the
> restarted process (i.e files that are not in the 'orig-fd-set').

There's no need to keep temporary/debug files open that I can see. Just a need to be careful when more than one new file descriptor has been created before doing a dup2().

> If cryo does, then maybe it could open such files:
>
> - after clone() (so files are not open in restarted process), or
>
> - find the last_fd used and dup2() to that fd, leaving the
> 'orig-fd-set' all open/available for restarted process
>
> For debug, before each 'dup2(fd1, fd2)' we could 'fstat(fd2, &buf)'
> to ensure 'fd2' is not in use and error out if it is.

fstat() could certainly be useful for debugging dup2(). However it still doesn't nicely show us whether there are any fds we've leaked that we forgot about unless we fstat() all possible fds and then compare the set of existing fds to the orig-fd-set.

Cheers,
-Matt

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

Subject: Re: [RFC][PATCH][cryo] Save/restore state of unnamed pipes

Matt Helsley [matthltc@us.ibm.com] wrote:
|
| > So let me rephrase the problem.
| >
| > Suppose the checkpointed application was using fds in following
| > "orig-fd-set"
| >
| > { [0..10], 18, 27 }
| >
| > where 18 and 27 are part of a pipe. For simplicity lets assume that
| > 18 is the read-side-fd.
|
| so orig_pipefd[0] == 18
| and orig_pipefd[1] == 27
|
| > We checkpointed this application and are now trying to restart it.
| >
| > In the restarted application, we would call
| >
| > dup2(fd1, fd2),
| >
| > where 'fd1' is some new, random fd and 'fd2' is an fd in 'orig-fd-set'
| ^^^^^^ Even if they were truly random, this
| does not preclude fd1 from having the same value as an fd in the
| remaining orig-fd-set -- such as one of the two we're about to try and
| restart with pipe().

I agree. fd1 could be an hither-to-unseen fd from the 'orig-fd-set'.

|
| > (say fd2 = 18).
|
| fd1 = restarted_pipefd[0]
| fd2 = restarted_pipefd[1]
|
| In my example fd1 == 27 and fd2 == 18
|
| > IIUC, there is a risk here of 'fd2' being closed accidentally while
| > it is in use.
|
| Yes, that's the risk.
|
| > But, the only way I can see 'fd2' being in use in the restarted process
| > is if _cryo_ opened some file _during_ restart and did not close. I ran
|
| Both file descriptors returned from pipe() are in use during restart

| and closing one of them would not be proper. Cryo hasn't "forgotten" to
| close one of them -- cryo needs to dup2() both of them to their
| "destination" fds. But if they have been swapped or if just one is the
| "destination" of the other then you could end up with a broken pipe.

Ok I see what you are saying.

The assumption I have is that we would process the fds from 'orig-fd-set'
in ascending order. Its good to confirm that assumption now :-)

proc_readfd_common() seems to return the fds in ascending order (so
readdir() of "/proc/pid/fd/" would get them in ascending order - no ?)

If we process 'orig-fd-set' in order and suppose we create the pipe for
the smaller of the two fds (could be the write-side). Then the other side
of the pipe would either not collide with an existing fd or that
fd would not be in the 'orig-fd-set' (in the latter case it would
be safe for dup2() to close).

|
| > into this early on with the randomize_va_space file (which was easily
| > fixed).
|
| This logic only works if cryo only has one new fd at a time. However
| that's not possible with pipe(). Or socketpair(). In those cases one of
| the two new fds could be the "destination" fd for the other. In that
| case dup2() will kindly close it for you and break your new
| pipe/socketpair! :)
|
| That's why I asked if POSIX guarantees the read side file descriptor is
| always less than the write side. If it does then the numbers can't be
| swapped and maybe using your assumption that we don't have any other fds
| accidentally left open ensures dup2() will be safe.

I don't think POSIX guarantees, but will double check.

|
| > Would cryo need to keep one or more temporary/debug files open in the
| > restarted process (i.e files that are not in the 'orig-fd-set').
|
| There's no need to keep temporary/debug files open that I can see. Just
| a need to be careful when more than one new file descriptor has been
| created before doing a dup2().
|
| > If cryo does, then maybe it could open such files:
| >
| > - after clone() (so files are not open in restarted process), or
| >

| > - find the last_fd used and dup2() to that fd, leaving the
| >   'orig-fd-set' all open/available for restarted process
| >
| > For debug, before each 'dup2(fd1, fd2)' we could 'fstat(fd2, &buf)'
| > to ensure 'fd2' is not in use and error out if it is.
|
| fstat() could certainly be useful for debugging dup2(). However it still
| doesn't nicely show us whether there are any fds we've leaked that we
| forgot about unless we fstat() all possible fds and then compare the set
| of existing fds to the orig-fd-set.

Yes, was suggesting fstat() only to detect collisions, but yes, to
detect leaks, we have to do more.

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

---

Subject: Re: [RFC][PATCH][cryo] Save/restore state of unnamed pipes
Posted by Matt Helsley on Wed, 18 Jun 2008 22:38:09 GMT
View Forum Message <> Reply to Message

On Wed, 2008-06-18 at 14:56 -0700, sukadev@us.ibm.com wrote:
> Matt Helsley [matthltc@us.ibm.com] wrote:
> |
> | > So let me rephrase the problem.
> | >
> | > Suppose the checkpointed application was using fds in following
> | > "orig-fd-set"
> | >
> | > { [0..10], 18, 27 }
> | >
> | > where 18 and 27 are part of a pipe. For simplicity lets assume that
> | > 18 is the read-side-fd.
> |
> | so orig_pipefd[0] == 18
> | and orig_pipefd[1] == 27
> |
> | > We checkpointed this application and are now trying to restart it.
> | >
> | > In the restarted application, we would call
> | >
> | > dup2(fd1, fd2),
> | >
> | > where 'fd1' is some new, random fd and 'fd2' is an fd in 'orig-fd-set'
> | ^^^^^^ Even if they were truly random, this
> | does not preclude fd1 from having the same value as an fd in the

> | remaining orig-fd-set -- such as one of the two we're about to try and
> | restart with pipe().
>
> I agree. fd1 could be an hither-to-unseen fd from the 'orig-fd-set'.
>
> |
> | > (say fd2 = 18).
> |
> | fd1 = restarted_pipefd[0]
> | fd2 = restarted_pipefd[1]
> |
> | In my example fd1 == 27 and fd2 == 18
> |
> | > IIUC, there is a risk here of 'fd2' being closed accidentally while
> | > it is in use.
> |
> | Yes, that's the risk.
> |
> | > But, the only way I can see 'fd2' being in use in the restarted process
> | > is if _cryo_ opened some file _during_ restart and did not close. I ran
> |
> | Both file descriptors returned from pipe() are in use during restart
> | and closing one of them would not be proper. Cryo hasn't "forgotten" to
> | close one of them -- cryo needs to dup2() both of them to their
> | "destination" fds. But if they have been swapped or if just one is the
> | "destination" of the other then you could end up with a broken pipe.
>
> Ok I see what you are saying.
>
> The assumption I have is that we would process the fds from 'orig-fd-set'
> in ascending order. Its good to confirm that assumption now :-)

OK, but wouldn't that violate the "pipes first" ordering we discussed?

> proc_readfd_common() seems to return the fds in ascending order (so
> readdir() of "/proc/pid/fd/" would get them in ascending order - no ?)
>
> If we process 'orig-fd-set' in order and suppose we create the pipe for
> the smaller of the two fds (could be the write-side). Then the other side
> of the pipe would either not collide with an existing fd or that
> fd would not be in the 'orig-fd-set' (in the latter case it would
> be safe for dup2() to close).

Hmm, I don't see how that solves the problem.
Example:

Suppose fds 0-10 are already "restarted", 11 and 12 are the read and
write ends of a pipe we need to restart.

Now restart does :

```
int pipefds[2];

pipe(pipefds); /*
  * kernel is allowed to return pipefds[0] == 12 and
   * pipefds[1] == 11
  */

dup2(pipefds[0], 11); /* closes pipefds[1]! */
dup2(pipefds[1], 12);
```

So even though we're processing the fds in the orig-fd-set in order, and regardless of how we completed restarting the earlier fds, we'd still have this problem. Note that if the write end of the pipe should be fd 200 you'd still have this problem. The swap case is the nastiest though because no matter whicg order we do the dup2() calls we'd break our shiny new pipe().

move_fds() solves the problem by brute force checking and dup()'ing (not dup2) any problematic fds to new, guaranteed-safe-for-the-moment, fds.

Cheers,
 -Matt

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

---

Subject: Re: [RFC][PATCH][cryo] Save/restore state of unnamed pipes
Posted by Sukadev Bhattiprolu on Wed, 18 Jun 2008 22:55:55 GMT
View Forum Message <> Reply to Message

| Now restart does :
|
| int pipefds[2];
|
| pipe(pipefds); /*
| * kernel is allowed to return pipefds[0] == 12 and
| * pipefds[1] == 11
| */
|
| dup2(pipefds[0], 11); /* closes pipefds[1]! */
| dup2(pipefds[1], 12);

Aah. I see it now (finally). Thanks,

Suka

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

---

Subject: Re: [RFC][PATCH][cryo] Save/restore state of unnamed pipes
Posted by Sukadev Bhattiprolu on Thu, 19 Jun 2008 07:59:53 GMT
View Forum Message <> Reply to Message

Matt Helsley [matthltc@us.ibm.com] wrote:

<snip>
|
| > | I don't see anything in the pipe man page, at least, that suggests we
| > | can safely assume pipefds[0] < pipefds[1].
| > |
| > | The solution could be to use "trampoline" fds. Suppose last_fd is the
| > | largest fd that exists in the final checkpointed/restarting application.
| > | We could do (Skipping the PT_FUNC "notation" for clarity):
| >
| > |
| > |
| > | pipe(pipefds); /* returns 5 and 4 in elements 0 and 1 */
| > | /* use fds after last_fd as trampolines for fds we want to create */
| > | dup2(pipefds[0], last_fd + 1);
| > | dup2(pipefds[1], last_fd + 2);
| > | close(pipefds[0]);
| > | close(pipefds[1]);
| > | dup2(last_fd + 1, <orig pipefd[0]>);
| > | dup2(last_fd + 2, <orig pipefd[1]>);
| > | close(last_fd + 1);
| > | close(last_fd + 2);
| > |
| > |
| > | Which is alot more code but should work no matter which fds we get back
| > | from pipe(). Of course this assumes the checkpointed application hasn't
| > | used all of its fds. :(
| > |

It appears that this last_fd approach will fit in easier with current
design of cryo (where we process one or two fds at a time and don't have
the src_fds and dest_fds handy).

BTW, we should be able to accomplish the above with a single-unused fd
right (i.e no need for last_fd+2) ?

| >
| > This sounds like a good idea too, but we could use any fd that has not
| > yet been used in the restart-process right ? It would break if all fds
|
| Yes, but we don't know which fd is available unless we allocate it
| without dup2().

Right. I was thinking we could find that out at the time of checkpoint
(a brute-force fstat(i, &statbuf) for i = 0..n or something more
efficient).

Well just thought of another approach.

Basically, we have a temporary need for an unused fd for use as a
trampoline. So, why not 'set-aside' an fd for that purpose and after
all other fds have been created, go back and create this fd ?

i.e lets say the first regular file we open is associated with 'fd = 3'.
We save away the 'fdinfo' for 3 say in a global variable and close(3).
Now use 'fd = 3' in place of last_fd+1 above.

Once all fds have been setup correctly, go back and set up fd = 3
using the saved fdinfo (this would be a simple open of the file
followed by seek and maybe an fcntl).

This would work even if the application was using all its fds ?

If we do need both last_fd+1 and last_fd+2, we would have to set
aside two regular files.

Hmm, would it work even if an application uses all (1024) its fds
for pipes :-), but just a thought at this point.

Suka

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

---

Subject: Re: [RFC][PATCH][cryo] Save/restore state of unnamed pipes
Posted by Matt Helsley on Thu, 19 Jun 2008 23:46:00 GMT
View Forum Message <> Reply to Message

On Thu, 2008-06-19 at 00:59 -0700, sukadev@us.ibm.com wrote:

> Matt Helsley [matthltc@us.ibm.com] wrote:
>
> <snip>
> |
> | > | I don't see anything in the pipe man page, at least, that suggests we
> | > | can safely assume pipefds[0] < pipefds[1].
> | > |
> | > | The solution could be to use "trampoline" fds. Suppose last_fd is the
> | > | largest fd that exists in the final checkpointed/restarting application.
> | > | We could do (Skipping the PT_FUNC "notation" for clarity):
> | >
> | > |
> | > |
> | > | pipe(pipefds); /* returns 5 and 4 in elements 0 and 1 */
> | > | /* use fds after last_fd as trampolines for fds we want to create */
> | > | dup2(pipefds[0], last_fd + 1);
> | > | dup2(pipefds[1], last_fd + 2);
> | > | close(pipefds[0]);
> | > | close(pipefds[1]);
> | > | dup2(last_fd + 1, <orig pipefd[0]>);
> | > | dup2(last_fd + 2, <orig pipefd[1]>);
> | > | close(last_fd + 1);
> | > | close(last_fd + 2);
> | > |
> | > |
> | > | Which is alot more code but should work no matter which fds we get back
> | > | from pipe(). Of course this assumes the checkpointed application hasn't
> | > | used all of its fds. :(
> | > |
>
> It appears that this last_fd approach will fit in easier with current
> design of cryo (where we process one or two fds at a time and don't have
> the src_fds and dest_fds handy).
>
> BTW, we should be able to accomplish the above with a single-unused fd
> right (i.e no need for last_fd+2) ?

Yes, I think that's sufficient:

```
    int pipefds[2];

 ...

 restarted_read_fd = 11;
 restarted_write_fd = 12;

 ...
```

```
        pipe(pipefds);

 /*
  * 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 (pipefds[1] == restarted_read_fd) {
  dup2(pipefds[1], last_fd + 1);
  pipefds[1] = last_fd + 1;
 }

 if (pipefds[0] != restarted_read_fd) {
  dup2(pipefds[0], restarted_read_fd);
  close(pipefds[0]);
 }

 if (pipefds[0] != restarted_read_fd) {
        dup2(pipefds[1], restarted_write_fd);
  close(pipefds[1]);
 }
```

I think this code does the minimal number of operations needed in the
restarted application too -- it counts on the second dup2() closing one
of the fds if pipefds[1] == restarted_read_fd.

Cheers,
 -Matt

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

---

Subject: Re: [RFC][PATCH][cryo] Save/restore state of unnamed pipes
Posted by Sukadev Bhattiprolu on Fri, 20 Jun 2008 01:54:10 GMT
View Forum Message <> Reply to Message

Matt Helsley [matthltc@us.ibm.com] wrote:
|
| Yes, I think that's sufficient:
|
| int pipefds[2];
|
| ...
|
| restarted_read_fd = 11;

```
|  restarted_write_fd = 12;
|
|  ...
|
|         pipe(pipefds);
|
|  /*
|   * 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 (pipefds[1] == restarted_read_fd) {
|   dup2(pipefds[1], last_fd + 1);
|   pipefds[1] = last_fd + 1;
|  }
|
|  if (pipefds[0] != restarted_read_fd) {
|   dup2(pipefds[0], restarted_read_fd);
|   close(pipefds[0]);
|  }
|
|  if (pipefds[0] != restarted_read_fd) {
|          dup2(pipefds[1], restarted_write_fd);
|   close(pipefds[1]);
|  }
```

Shouldn't the last if be

```
 if (pipefds[1] != restarted_wrte_fd) ?
```

(otherwise it would break if pipefds[0] = 11 and pipefds[1] = 200)

I came up with something similar, but with an extra close(). And
in my code, I had restarted_* names referring to pipefds[] making
it a bit confusing initially.

How about using actual_fds[] (instead of pipefds) and expected_fds[]
instead of (restart_*) ?

Thanks,

Suka

```
|
| I think this code does the minimal number of operations needed in the
| restarted application too -- it counts on the second dup2() closing one
| of the fds if pipefds[1] == restarted_read_fd.
|
```

| Cheers,
| -Matt

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

---

Subject: Re: [RFC][PATCH][cryo] Save/restore state of unnamed pipes
Posted by Matt Helsley on Fri, 20 Jun 2008 02:48:19 GMT
View Forum Message <> Reply to Message

On Thu, 2008-06-19 at 18:54 -0700, sukadev@us.ibm.com wrote:
> Matt Helsley [matthltc@us.ibm.com] wrote:

<snip>

> | if (pipefds[0] != restarted_read_fd) {
> | dup2(pipefds[1], restarted_write_fd);
> | close(pipefds[1]);
> | }
>
> Shouldn't the last if be
>
> if (pipefds[1] != restarted_wrte_fd) ?
>
> (otherwise it would break if pipefds[0] = 11 and pipefds[1] = 200)

Argh, copy-paste error. You are correct.

> I came up with something similar, but with an extra close(). And
> in my code, I had restarted_* names referring to pipefds[] making
> it a bit confusing initially.
>
> How about using actual_fds[] (instead of pipefds) and expected_fds[]
> instead of (restart_*) ?

I like actual_fds[] instead of pipefds[] but still prefer the restart_*
names over expected_fds[].

Cheers,
 -Matt

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

---

Subject: Re: [RFC][PATCH][cryo] Save/restore state of unnamed pipes
Posted by Oren Laadan on Sun, 22 Jun 2008 21:40:52 GMT
View Forum Message <> Reply to Message

sukadev@us.ibm.com wrote:
> From fd13986de32af31621b1badbcf7bfb5626648e0e 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.

One issue with this approach is that the operation (checkpoint) becomes
non-transparent to the "checkpointee"s: re-writing the data back into
the pipe/fifo may generate a SIGIO at the read-side of the pipe/fifo, or
an inotify event on the inode, without a real reason. This may confuse
the application if its logic relies on "normal" behavior.

>
> - 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.
>
> Basic unit-testing done at this point (using tests/pipe.c).
>
> TODO:
> - Additional testing (with multiple-processes and multiple-pipes)
> - Named-pipes
>

> Signed-off-by: Sukadev Bhattiprolu <sukadev@us.ibm.com>
> ---
> cr.c | 215 ++----
> 1 files changed, 203 insertions(+), 12 deletions(-)
>
> diff --git a/cr.c b/cr.c
> index 5163a3d..0cb9774 100644
> --- a/cr.c
> +++ b/cr.c
> @@ -84,6 +84,11 @@ 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 */
> @@ -468,6 +473,128 @@ 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;
> +
> + 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;
> @@ -587,6 +714,8 @@ 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++)
> @@ -839,6 +968,29 @@ 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]);
>   }
>
>   /*
> @@ -847,20 +999,8 @@ 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);
> @@ -1231,6 +1371,55 @@ 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("Error %d writing to FIFO '%s'\n",
> +      errno, fname);
> +   }
> +   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];
> @@ -1314,6 +1503,8 @@ 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")) {
```

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

---

Subject: Re: [RFC][PATCH][cryo] Save/restore state of unnamed pipes
Posted by Dave Hansen on Mon, 23 Jun 2008 23:30:28 GMT
View Forum Message <> Reply to Message

On Sun, 2008-06-22 at 17:40 -0400, Oren Laadan wrote:
> One issue with this approach is that the operation (checkpoint) becomes

> non-transparent to the "checkpointee"s: re-writing the data back into
> the pipe/fifo may generate a SIGIO at the read-side of the pipe/fifo, or
> an inotify event on the inode, without a real reason. This may confuse
> the application if its logic relies on "normal" behavior.

Yeah, I guess this is true, somewhat.

But, for something expecting to read from a pipe, they'll get a SIGIO when the data is first written. If the signal has not yet been delivered then we're fine because we won't generate *another* one.

If they've already received the signal and left the data in the pipe, what harm does another SIGIO do? The kernel is free to give the app a SIGIO for every single byte stuck in the pipe, right? So, what difference does any number of SIGIOs make when there's data sitting in the pipe unread?

As for inotify, yeah that might be an actual issue. It's probably good to note this potential issue, and to watch for anyone doing inotifies on a pipe fd when we get to checkpointing inotify.

Overall, seems like an awfully minor issue to me.

-- Dave

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