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
++++++
```

```
> | > > +
> | > > +}
> | > > +
> | > > +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