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

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

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

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

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

TODO:

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

- Temporarily using a regular-file's fd as 'trampoline-fd' when all fds are in use
- Maybe read all fdinfo into memory during restart, so we can reduce the information we save into the checkpoint-file (see comments near 'struct fdinfo').
- Check logic of detecting 'dup's of pipe fds (any hidden gotchas ?) See pair_pipe_fds()
- Alloc ppi_list[] dynamically (see getfdinfo()).
- Use getrlimit() to compute max-open-fds (see near caller of pair_pipe_fds()).
- [Oleg Nesterov]: SIGIO/inotify() issues associated with writing-back to pipes (fixing this would require some assistance from kernel ?)

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

Signed-off-by: Sukadev Bhattiprolu <sukadev@us.ibm.com>

cr.c | 262 ++-----
1 files changed, 240 insertions(+), 22 deletions(-)

```

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

+/*
+ * TODO: restore_fd() processes each fd as it reads it of the checkpoint
+ * file. To avoid making a second-pass at the file, we store following
+ * fields during checkpoint (for now).
+ *
+ * peer_fdnum, dup_fdnum, create_pipe, tramp_fd' fields can be
+ *
+ * We could eliminate this fields by reading all fdinfo into memory
+ * and then 'computing' the above fields before processing the fds.
+ * But this would require a non-trivial rewrite of the restore_fd()
+ * logic. Hopefully that can be done without significant impact to
+ * rest of the logic associated with saving/restoring pipes.
+ */
typedef struct fdinfo_t {
    int fdnum; /* file descriptor number */
+ int peer_fdnum; /* peer fd for pipes */
+ int dup_fdnum; /* fd, if fd is dup of another pipe fd */
+ int create_pipe; /* TRUE if this is the create-end of the pipe */
+ int tramp_fd; /* trampoline-fd for use in restoring pipes */
    mode_t mode; /* mode as per stat(2) */
    off_t offset; /* read/write pointer position for regular files */
    int flag; /* open(2) flag */
@@ -117,6 +134,7 @@ typedef struct pinfo_t {
    int nt; /* number of thread child (0 if no thread lib) */
    pid_t *tpid; /* array of thread info */
    struct pinfo_t *pmt; /* multithread: pointer to main thread info */
+ int tramp_fd; /* trampoline-fd for use in restoring pipes */
} pinfo_t;

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

+typedef struct pipe_peer_info {
+ fdinfo_t *pipe_fdi;
+ //fdinfo_t *peer_fdi;
+ __ino_t pipe_ino;
+} pipe_peer_info_t;

```

```

+
+ __ino_t get_fd_ino(char *fname)
+{
+ struct stat sbuf;
+
+ if (stat(fname, &sbuf) < 0)
+ ERROR("stat() on fd %s failed, errno %d\n", fname, errno);
+
+ return sbuf.st_ino;
+}
+
+static void pair_pipe_fds(pipe_peer_info_t *ppi_list, int npipe_fds)
+{
+ int i, j;
+ pipe_peer_info_t *xppi, *yppi;
+ fdinfo_t *xfdi, *yfdi;
+
+ /*
+  * TODO: This currently assumes pipefds have not been dup'd.
+  * Of course, need to kill this assumption soon.
+  */
+ for (i = 0; i < npipe_fds; i++) {
+ xppi = &ppi_list[i];
+ xfdi = xppi->pipe_fdi;
+
+ j = i + 1;
+ for (j = i+1; j < npipe_fds; j++) {
+ yppi = &ppi_list[j];
+ yfdi = yppi->pipe_fdi;
+
+ if (yppi->pipe_ino != xppi->pipe_ino)
+ continue;
+
+ DEBUG("Checking flag i %d, j %d\n", i, j);
+ /*
+  * i and j refer to same pipe. Check if they are
+  * peers or aliases (dup'd fds). dup'd fds share
+  * file-status flags. Peer fds of unnamed pipes
+  * differ in O_WRONLY bit.
+  */
+ * TODO:
+ * CHECK ABOVE ASSUMPTION
+ */
+ if (xfdi->flag == yfdi->flag) {
+ yfdi->dup_fdnum = xfdi->fdnum;
+ DEBUG("Pipe fds %d and %d are dups\n",
+ xfdi->fdnum, yfdi->fdnum);
+ } else {

```

```

+   DEBUG("Pipe fds %d and %d are peers\n",
+       xfdi->fdnum, yfdi->fdnum);
+
+   /*
+    * If we have already paired it or determined
+    * it is a dup, ignore
+    */
+   if (xfdi->peer_fdnum != -1 ||
+       xfdi->dup_fdnum != -1)
+       continue;
+
+   DEBUG("Pipe fd %d not paired yet\n",
+       xfdi->fdnum);
+   /*
+    * Create pipe on first end of the pipe we
+    * come across.
+    */
+   if (xfdi->fdnum < yfdi->fdnum)
+       xfdi->create_pipe = 1;
+
+   xfdi->peer_fdnum = yfdi->fdnum;
+   yfdi->peer_fdnum = xfdi->fdnum;
+   }
+   DEBUG("Clearing ino i %d, j %d\n", i, j);
+   }
+   }
+
+   DEBUG("Done building pipe list i %d, j %d\n", i, j);
+}
+
+/*
+ * getfds() parse the process open file descriptors as found in /proc.
+ */
@@ -275,6 +376,10 @@ int getfdinfo(pinfo_t *pi)
    int len, n = 0;
    pid_t syscallpid = pi->syscallpid ? pi->syscallpid : pi->pid;

+   int npipe_fds = 0;
+   pipe_peer_info_t ppi_list[256]; // TODO: alloc dynamically
+   pipe_peer_info_t *ppi;
+
    snprintf(dname, sizeof(dname), "/proc/%u/fd", pi->pid);
    if (! (dir = opendir(dname))) return 0;
    while ((dent = readdir(dir))) {
@@ -288,14 +393,54 @@ int getfdinfo(pinfo_t *pi)
    stat(dname, &st);
    pi->fi[n].mode = st.st_mode;
    pi->fi[n].flag = PT_FCNTL(syscallpid, pi->fi[n].fdnum, F_GETFL, 0);

```

```

+ pi->fi[n].create_pipe = 0;
+ pi->fi[n].tramp_fd = -1;
+ pi->fi[n].dup_fdnum = -1;
+ pi->fi[n].peer_fdnum = -1;
+ if (S_ISREG(st.st_mode))
+   pi->fi[n].offset = (off_t)PT_LSEEK(syscallpid, pi->fi[n].fdnum, 0, SEEK_CUR);
- else if (S_ISFIFO(st.st_mode))
+ else if (S_ISFIFO(st.st_mode)) {
+   t_s("fifo");
+   ppi = &ppi_list[npipe_fds];
+   ppi->pipe_fdi = &pi->fi[n];
+   //ppi->peer_fdi = NULL;
+   ppi->pipe_ino = get_fd_ino(dname);
+
+   DEBUG("Found a pipe: fd %d, flag 0x%x ino %d\n",
+     ppi->pipe_fdi->fdnum,
+     ppi->pipe_fdi->flag, ppi->pipe_ino);
+   npipe_fds++;
+
+ }
+ else if (S_ISSOCK(st.st_mode))
+   getsockinfo(syscallpid, pi, pi->fi[n].fdnum);
+   n++;
+ }
+
+ if (n) {
+   pi->tramp_fd = pi->fi[n-1].fdnum + 1;
+   DEBUG("Using %d as trampoline-fd\n", pi->tramp_fd);
+ }
+
+ /*
+  * TODO: replace 1024 with rlim.rlim_cur (but need to execute
+  *   getrlimit() in checkpointed-process)
+  */
+ if (npipe_fds && pi->tramp_fd >= 1024) {
+   /*
+    * TODO:
+    * This restriction can be relaxed a bit. We only
+    * need to give-up here if all fds are in use as
+    * pipe-fds.
+    */
+   ERROR("Cannot allocate a 'trampoline_fd' for pipes");
+ }
+
+ /*
+  * Now that we found all fds, pair up any pipe_fds
+  */
+ pair_pipe_fds(ppi_list, npipe_fds);

```

```

+
end:
    closedir(dir);
    return n;
@@ -550,7 +695,11 @@ static int save_process_fifo_info(pinfo_t *pi, int fd)
    DEBUG("FIFO fd %d (%s), flag 0x%x\n", fi[i].fdnum, fi[i].name,
        fi[i].flag);

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

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

+static void match_pipe_ends(int pid, int tramp_fd, int expected_fds[],
+ int actual_fds[])
+{
+ int ret;
+ int expected_read_fd = expected_fds[0];
+ int expected_write_fd = expected_fds[1];
+
+ DEBUG("tramp_fd %d\n", tramp_fd);
+ /*
+  * pipe() may have returned one (or both) of the restarted fds
+  * at the wrong end of the pipe. This could cause dup2() to
+  * accidentally close the pipe. Avoid that with an extra dup().
+  */
+ if (actual_fds[1] == expected_read_fd) {

```

```

+         t_d(ret = PT_DUP2(pid, actual_fds[1], tramp_fd + 1));
+         actual_fds[1] = tramp_fd + 1;
+     }
+
+     if (actual_fds[0] != expected_read_fd) {
+         t_d(ret = PT_DUP2(pid, actual_fds[0], expected_read_fd));
+         t_d(PT_CLOSE(pid, actual_fds[0]));
+     }
+
+     if (actual_fds[1] != expected_write_fd) {
+         t_d(ret = PT_DUP2(pid, actual_fds[1], expected_write_fd));
+         t_d(PT_CLOSE(pid, actual_fds[1]));
+     }
+ }
+
+static void recreate_pipe(int pid, int tramp_fd, fdinfo_t *fdinfo)
+{
+    int actual_fds[2] = { 0, 0 };
+    int expected_fds[2];
+
+    DEBUG("Creating pipe for fd %d\n", fdinfo->fdnum);
+
+    t_d(PT_PIPE(pid, actual_fds));
+    t_d(actual_fds[0]);
+    t_d(actual_fds[1]);
+
+    /*
+     * Find read-end and write-end of the checkpointed pipe
+     * (i.e don't assume that read-side fd is smaller than
+     * write-side fd)
+     */
+    if (fdinfo->flag & O_WRONLY) {
+        expected_fds[0] = fdinfo->peer_fdnum;
+        expected_fds[1] = fdinfo->fdnum;
+    } else {
+        expected_fds[0] = fdinfo->fdnum;
+        expected_fds[1] = fdinfo->peer_fdnum;
+    }
+
+    /*
+     * Match the ends of newly created pipe with the ends of the
+     * checkpointed pipe.
+     */
+    match_pipe_ends(pid, tramp_fd, expected_fds, actual_fds);
+
+    /*
+     * for debug, use fcntl() on fdinfo->fdnum and fdinfo->peer_fdnum
+     * to ensure ends match

```

```

+ */
+
+ DEBUG("Done creating pipe '{%d, %d}'\n", fdinfo->fdnum,
+ fdinfo->peer_fdnum);
+}
+
+int restore_fd(int fd, pid_t pid)
+{
+    char item[64];
@@ -1001,34 +1224,29 @@ int restore_fd(int fd, pid_t pid)
+    if (pfd != fdinfo->fdnum) t_d(PT_CLOSE(pid, pfd));
+}
+} else if (S_ISFIFO(fdinfo->mode)) {
- int pipefds[2] = { 0, 0 };
-
+ if (fdinfo->dup_fdnum != -1) {
+     t_d(ret = PT_DUP2(pid, fdinfo->dup_fdnum,
+         fdinfo->fdnum));
+ }
+ /*
- * We create the pipe when we see the pipe's read-fd.
- * Just ignore the pipe's write-fd.
+ * When checkpointing, we arbitrarily mark one end
+ * of the pipe as the 'create-end'. Create a pipe
+ * if this fd is the 'create-end' and then restore
+ * the fcntl-flags. For the other end of the pipe,
+ * just restore its fcntl-flags.
+ */
- if (fdinfo->flag == O_WRONLY)
-     continue;
-
- DEBUG("Creating pipe for fd %d\n", fdinfo->fdnum);
-
- t_d(PT_PIPE(pid, pipefds));
- t_d(pipefds[0]);
- t_d(pipefds[1]);
-
- if (pipefds[0] != fdinfo->fdnum) {
-     DEBUG("Hmm, new pipe has fds %d, %d "
-         "Old pipe had fd %d\n", pipefds[0],
-         pipefds[1], fdinfo->fdnum); getchar();
-     exit(1);
- }
- DEBUG("Done creating pipefds[0] %d\n", pipefds[0]);
+ else if (fdinfo->create_pipe)
+     recreate_pipe(pid, fdinfo->tramp_fd, fdinfo);
+ }

```



```

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

```

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