Subject: [PATCH 0/11] user-cr: support for pids as shared objects (v2)
Posted by Oren Laadan on Mon, 07 Feb 2011 17:21:21 GMT

View Forum Message <> Reply to Message

This patchset adds the necessary support in user-cr related to handling of pids as proper shared objets. You must use this if you use the corresponding kernel-cr patchset recetly posted.

Changelog[v2]:
- Cleanups and many bug fixes: it now passes all tests in tests-cr

Oren.

____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containe rs

Subject: [PATCH 04/11] restart: improve success/failure reporting
Posted by Oren Laadan on Mon, 07 Feb 2011 17:21:25 GMT

View Forum Message <> Reply to Message

This patch improves the final status report (success/fail) of restart by adding a new ctx->success variable. The motivation is twofold:

1) Before, there was a confusion between who reports an error when (in --pidns) the root task isn't container init - in which case we add a dummy init. For this, we also improve how the dummy init communicates the status (success/fail) to the original restart task.

2) Even after restart succeeds, there may be errors (which may not affect the restarted task, but should be nevertheless reported).

Signed-off-by: Oren Laadan <orenl@cs.columbia.edu>

```
---
 restart.c |   81 +++++++++++++++++++++++++++++++++++++++++++++++++-------------------------
 1 files changed, 46 insertions(+), 35 deletions(-)

diff --git a/restart.c b/restart.c
index 9535543..a1af631 100644
--- a/restart.c
+++ b/restart.c
@@ -125,6 +125,7 @@ struct ckpt_ctx {
  } whoami;

  int error;
+ int success;
```

```
  pid_t root_pid;
  int pipe_in;
@@ -677,9 +678,21 @@ int cr_restart(struct cr_restart_args *args)
  if (global_feeder_pid)
   waitpid(global_feeder_pid, NULL, 0);

- if (ret < 0)
+ if (ctx.error)
   errno = ctx.error;

+ if (ctx.success) {
+  ckpt_dbg("c/r succeeded\n");
+  ckpt_verbose("Restart succeeded\n");
+  if (ctx.error)
+   ckpt_verbose("Post restart error: %d\n", ctx.error);
+ } else {
+  ckpt_dbg("c/r failed ?\n");
+  ckpt_perror("restart");
+  ckpt_verbose("Restart failed\n");
+  ret = -1;
+ }
+
  return ret;
 }

@@ -827,7 +840,7 @@ static int ckpt_pretend_reaper(struct ckpt_ctx *ctx)
  return ckpt_parse_status(global_child_status, 1, 0);
 }

-static int ckpt_probe_child(pid_t pid, char *str)
+static int ckpt_probe_child(struct ckpt_ctx *ctx, pid_t pid, char *str)
 {
  int status, ret;

@@ -840,17 +853,16 @@ static int ckpt_probe_child(pid_t pid, char *str)
  ret = waitpid(pid, &status, WNOHANG);
  if (ret == pid) {
   report_exit_status(status, str, 0);
-  errno = ECHILD;
-  return -1;
+  return ctx_ret_errno(ctx, ECHILD);
  } else if (ret < 0 && errno == ECHILD) {
   ckpt_err("WEIRD: %s exited without trace (%s)\n",
     str, strerror(errno));
-  return -1;
+  return ctx_set_errno(ctx);
  } else if (ret != 0) {
```

```
   ckpt_err("waitpid for %s (%s)", str, strerror(errno));
   if (ret > 0)
    errno = ECHILD;
-  return -1;
+  return ctx_set_errno(ctx);
  }
  return 0;
 }
@@ -900,30 +912,39 @@ static int __ckpt_coordinator(void *arg)
  if (!ctx->args->wait)
   close(ctx->pipe_coord[0]);

- return ckpt_coordinator(ctx);
+ /* set the exit status properly */
+ return ckpt_coordinator(ctx) >= 0 ? 0 : 1;
 }

 static int ckpt_coordinator_status(struct ckpt_ctx *ctx)
 {
- int status = -1;
+ int status;
  int ret;

  close(ctx->pipe_coord[1]);
  ctx->pipe_coord[1] = -1;  /* mark unused */

  ret = read(ctx->pipe_coord[0], &status, sizeof(status));
- if (ret < 0)
-  ckpt_perror("read coordinator status");
- else if (ret == 0) {
-  /* coordinator failed to report */
-  ckpt_dbg("Coordinator failed to report status.");
- } else
-  status = 0;

  close(ctx->pipe_coord[0]);
  ctx->pipe_coord[0] = -1;  /* mark unused */

- return status;
+ if (ret < 0) {
+  ckpt_perror("read coordinator status");
+  return ctx_set_errno(ctx);
+ } else if (ret != sizeof(status)) {
+  /* coordinator failed to report */
+  ckpt_dbg("Coordinator failed to report status\n");
+  return ctx_ret_errno(ctx, EIO);
+ } else if (status != 0) {
+  /* coordinator reported failure */
```

```
+  ckpt_dbg("Coordinator reported error\n");
+  return ctx_ret_errno(ctx, status);
+ }
+
+ /* success ! */
+ ctx->success = 1;
+ return 0;
 }

 static int ckpt_coordinator_pidns(struct ckpt_ctx *ctx)
@@ -977,8 +998,8 @@ static int ckpt_coordinator_pidns(struct ckpt_ctx *ctx)
   * The child (coordinator) may have already exited before the
   * signal handler was plugged; verify that it's still there.
   */
- if (ckpt_probe_child(coord_pid, "coordinator") < 0)
-  return ctx_set_errno(ctx);
+ if (ckpt_probe_child(ctx, coord_pid, "coordinator") < 0)
+  return -1;

  ctx->args->copy_status = copy;

@@ -1017,8 +1038,8 @@ static int ckpt_coordinator(struct ckpt_ctx *ctx)
   * The child (root_task) may have already exited before the
   * signal handler was plugged; verify that it's still there.
   */
- if (ckpt_probe_child(root_pid, "root task") < 0)
-  return ctx_set_errno(ctx);
+ if (ckpt_probe_child(ctx, root_pid, "root task") < 0)
+  return -1;

  if (ctx->args->keep_frozen)
   flags |= RESTART_FROZEN;
@@ -1028,20 +1049,15 @@ static int ckpt_coordinator(struct ckpt_ctx *ctx)
  ret = restart(root_pid, ctx->args->infd,
        flags, ctx->args->klogfd);

- if (ret < 0) {
-  ckpt_perror("restart failed");
-  ckpt_verbose("Failed\n");
-  ckpt_dbg("restart failed ?\n");
-  return ret;
+ if (ret >= 0) {
+  ctx->success = 1;  /* restart succeeded ! */
+  ret = 0;
  }

- ckpt_verbose("Success\n");
- ckpt_dbg("restart succeeded\n");
```

```
-
- ret = 0;
-
  if (ctx->args->pidns && ctx->tasks_arr[0].pid != 1) {
   /* Report success/failure to the parent */
+  if (ret < 0)
+   ret = ctx->error;
   if (write(ctx->pipe_coord[1], &ret, sizeof(ret)) < 0) {
    ckpt_perror("failed to report status");
    return ctx_set_errno(ctx);
@@ -1068,11 +1084,6 @@ static int ckpt_coordinator(struct ckpt_ctx *ctx)
   ret = ckpt_collect_child(ctx);
  }

- if (ret < 0)
-  ckpt_dbg("c/r failed ?\n");
- else
-  ckpt_dbg("c/r succeeded\n");
-
  return ret;
 }

--
1.7.1
```

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

---

Subject: [PATCH 05/11] restart: obtain pid_max from /proc/sys/kernel/pid_max
Posted by Oren Laadan on Mon, 07 Feb 2011 17:21:26 GMT
View Forum Message <> Reply to Message

```
Signed-off-by: Oren Laadan <orenl@cs.columbia.edu>
---
 restart.c |   19 ++++++++++++++++--
 1 files changed, 17 insertions(+), 2 deletions(-)

diff --git a/restart.c b/restart.c
index a1af631..05d101b 100644
--- a/restart.c
+++ b/restart.c
@@ -158,6 +158,9 @@ struct ckpt_ctx {
  struct cr_restart_args *args;

  char *freezer;
```

```
+
+ /* system limits */
+ pid_t pid_max;
 };

 struct pid_swap {
@@ -489,6 +492,9 @@ int process_args(struct cr_restart_args *args)

 static void init_ctx(struct ckpt_ctx *ctx)
 {
+ FILE *file;
+ char buf[1024];
+
  memset(ctx, 0, sizeof(*ctx));

  /* mark all fds as unused */
@@ -500,6 +506,15 @@ static void init_ctx(struct ckpt_ctx *ctx)
  ctx->pipe_feed[1] = -1;
  ctx->pipe_coord[0] = -1;
  ctx->pipe_coord[1] = -1;
+
+ /* system limits */
+ ctx->pid_max = SHRT_MAX;  /* default */
+ file = fopen("/proc/sys/kernel/pid_max", "r");
+ if (file) {
+  if (fgets(buf, 1024, file))
+   ctx->pid_max = atoi(buf);
+  fclose(file);
+ }
 }

 static void exit_ctx(struct ckpt_ctx *ctx)
@@ -1223,12 +1238,12 @@ static int ckpt_alloc_pid(struct ckpt_ctx *ctx)
   * (this will become inefficient if pid-space is exhausted)
   */
  do {
-  if (ctx->tasks_pid == INT_MAX)
+  if (ctx->tasks_pid == ctx->pid_max)
    ctx->tasks_pid = CKPT_RESERVED_PIDS;
   else
    ctx->tasks_pid++;

-  if (n++ == INT_MAX) { /* ohhh... */
+  if (n++ == ctx->pid_max) { /* ohhh... */
    ckpt_err("pid namsepace exhausted");
    return -1;
   }
--
```

1.7.1

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

Subject: [PATCH 07/11] restart: explicitly disallow orphan tasks with --no-pidns
Posted by Oren Laadan on Mon, 07 Feb 2011 17:21:28 GMT
View Forum Message <> Reply to Message

Add a test to ckpt_build_tree() that, if --no-pidns is required, will verify that the tasks data structure will not lead to an orphan task during the restart (since this should not have been possible in the original subtree checkpoint).

This is useful beyond a sanity check:

If --no-pidns is set, then every child task asks to be notified of the parent's death via prctl (for better cleanup). This is ok because from a subtree checkpoint there should never be orphan tasks. But if we do have an orphan task (due to e.g. a bug or incorrectly modified image), then the following may happen:

A ghost/dead task that has a child (should be impossilbe in subtree image) may exit before that child; In that case, the child receives the signal and dies prematurely. Restart may either fail if this happens early enough, or simply hang since the coordinator will be waiting forever for all the tasks to complete.

Besides the sanity test, we also modify the parent death signal to be SIGUSR1 instead of SIGKILL, and we catch it - so if the race happens while the restarting task is still in userspace, we can be verbose about it. We also add a hanlder for SIGSEGV, so we'll get a message about it too.

Signed-off-by: Oren Laadan <orenl@cs.columbia.edu>
---
 restart.c | 50 ++-----
 1 files changed, 45 insertions(+), 5 deletions(-)

diff --git a/restart.c b/restart.c
index 966f7a1..01566c2 100644
--- a/restart.c
+++ b/restart.c
@@ -306,6 +306,18 @@ static char *sig2str(int sig)
  return "UNKNOWN SIGNAL";

```
 }

+static void sigusr1_handler(int sig)
+{
+ ckpt_dbg("SIGUSER1: restarting process would become orphan.\n");
+ exit(1);
+}
+
+static void sigsegv_handler(int sig)
+{
+ ckpt_dbg("SIGSEGV: restarting process unexpected fault.\n");
+ exit(1);
+}
+
 static void sigchld_handler(int sig)
 {
  int collected = 0;
@@ -573,6 +585,7 @@ int cr_restart(struct cr_restart_args *args)
    ckpt_perror("unshare");
    goto cleanup;
   }
+
   /* chroot ? */
   if (args->root && chroot(args->root) < 0) {
    ckpt_perror("chroot");
@@ -912,6 +925,9 @@ static int __ckpt_coordinator(void *arg)

  /* none of this requires cleanup: we're forked ... */

+ /* we want to be vocal about sigsegv */
+ signal(SIGSEGV, sigsegv_handler);
+
  /* chroot ? */
  if (ctx->args->root && chroot(ctx->args->root) < 0) {
   ckpt_perror("chroot");
@@ -1114,7 +1130,7 @@ static inline struct task *ckpt_init_task(struct ckpt_ctx *ctx)
 static int ckpt_build_tree(struct ckpt_ctx *ctx)
 {
  struct task *task;
- int i;
+ int i, found;

  /*
   * Allow for additional tasks to be added on demand for
@@ -1167,6 +1183,25 @@ static int ckpt_build_tree(struct ckpt_ctx *ctx)
  ckpt_dbg(".....\n");
 #endif
```

```
+ /*
+  * For --no-pidns verify that the resulting tree won't create
+  * orphan tasks (comment in ckpt_fork_stub() explain why).
+  */
+ if (!ctx->args->pidns) {
+  found = 0;
+  for (i = 0; i < ctx->tasks_nr; i++) {
+   task = &ctx->tasks[i];
+   if (!(task->flags & (TASK_GHOST | TASK_DEAD)))
+    continue;
+   if (!task->children)
+    continue;
+   ckpt_err("Bad orphan task (#%d) with --no-pidns\n", i);
+   found = 1;
+  }
+  if (found)
+   return ctx_ret_errno(ctx, EINVAL);
+ }
+
  return 0;
 }

@@ -1814,15 +1849,20 @@ int ckpt_fork_stub(void *data)
   * death by asking to be killed then. When restart succeeds,
   * it will have replaced this with the original value.
   *
-  * This works because in the --no-pids case, the hierarchy of
-  * tasks does not contain zombies (else, there must also be a
-  * container init, whose pid (==1) is clearly already taken).
+  * This works because in the --no-pidns case, the hierarchy of
+  * tasks does not contain zombies; else, there must also be a
+  * container init, whose pid (==1) is clearly already taken.
+  * This is verified in ckpt_build_tree().
   *
   * Thus, if a the parent of this task dies before this prctl()
   * call, it suffices to test getppid() == task->parent_pid.
+  *
+  * We use SIGUSR1 so that we can catch it and issue an error
+  * or debug message when this happens.
   */
  if (!ctx->args->pidns) {
-  if (prctl(PR_SET_PDEATHSIG, SIGKILL, 0, 0, 0) < 0) {
+  signal(SIGUSR1, sigusr1_handler);
+  if (prctl(PR_SET_PDEATHSIG, SIGUSR1, 0, 0, 0) < 0) {
    ckpt_perror("prctl");
    return ctx_set_errno(ctx);
   }
--
```

1.7.1

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

Subject: [PATCH 08/11] udpate kernel headers: support for pids objects
Posted by Oren Laadan on Mon, 07 Feb 2011 17:21:29 GMT
View Forum Message <> Reply to Message

---
include/linux/checkpoint_hdr.h | 29 ++++++++++++++++++++++++++-----
1 files changed, 24 insertions(+), 5 deletions(-)

diff --git a/include/linux/checkpoint_hdr.h b/include/linux/checkpoint_hdr.h
index f7c4d9a..227bfbe 100644
--- a/include/linux/checkpoint_hdr.h
+++ b/include/linux/checkpoint_hdr.h
@@ -94,7 +94,9 @@ enum {
CKPT_HDR_SECURITY,
#define CKPT_HDR_SECURITY CKPT_HDR_SECURITY

- CKPT_HDR_TREE = 101,
+ CKPT_HDR_PIDS = 101,
+#define CKPT_HDR_PIDS CKPT_HDR_PIDS
+ CKPT_HDR_TREE,
#define CKPT_HDR_TREE CKPT_HDR_TREE
CKPT_HDR_TASK,
#define CKPT_HDR_TASK CKPT_HDR_TASK
@@ -232,6 +234,8 @@ struct ckpt_hdr_objref {
enum obj_type {
CKPT_OBJ_IGNORE = 0,
#define CKPT_OBJ_IGNORE CKPT_OBJ_IGNORE
+ CKPT_OBJ_PID,
+#define CKPT_OBJ_PID CKPT_OBJ_PID
CKPT_OBJ_INODE,
#define CKPT_OBJ_INODE CKPT_OBJ_INODE
CKPT_OBJ_FILE_TABLE,
@@ -343,24 +347,39 @@ struct ckpt_hdr_container {
*/
} __attribute__((aligned(8)));;

+/* pids array */
+struct ckpt_hdr_pids {
+ struct ckpt_hdr h;
+ __u32 nr_pids;

```
+ __u32 nr_vpids;
+ __u32 offset;
+} __attribute__((aligned(8)));
+
+struct ckpt_pids {
+ __u32 depth;
+ __s32 numbers[1];
+} __attribute__((aligned(8)));
+
 /* task tree */
 struct ckpt_hdr_tree {
  struct ckpt_hdr h;
- __s32 nr_tasks;
+ __u32 nr_tasks;
 } __attribute__((aligned(8)));

-struct ckpt_pids {
+struct ckpt_task_pids {
  /* These pids are in the root_nsproxy's pid ns */
  __s32 vpid;
  __s32 vppid;
  __s32 vtgid;
  __s32 vpgid;
  __s32 vsid;
- __s32 depth; /* pid namespace depth relative to container init */
+ __u32 depth;
 } __attribute__((aligned(8)));

 /* pids */
-#define CKPT_PID_NULL -1
+/* (negative but not valid error) */
+#define CKPT_PID_NULL (-4096) /* null pid pointer */
+#define CKPT_PID_ROOT (-4097) /* pid same as root task */

 /* task data */
 struct ckpt_hdr_task {
--
1.7.1
```

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

---

Subject: [PATCH 09/11] ckptinfo: s/ckpt_pids/ckpt_task_pids/ after kerenl header update

Posted by Oren Laadan on Mon, 07 Feb 2011 17:21:30 GMT
View Forum Message <> Reply to Message

Signed-off-by: Oren Laadan <orenl@cs.columbia.edu>
---
 ckptinfo.c | 4 ++--
 1 files changed, 2 insertions(+), 2 deletions(-)

diff --git a/ckptinfo.c b/ckptinfo.c
index d73b38c..1361c21 100644
--- a/ckptinfo.c
+++ b/ckptinfo.c
@@ -254,7 +254,7 @@ static int image_parse(int fd, struct args *args)
 static int image_parse_tree(struct ckpt_hdr *h, int fd, struct args *args)
 {
  struct ckpt_hdr_tree *hh;
- struct ckpt_pids *pp;
+ struct ckpt_task_pids *pp;
  int nr_tasks;
  int i, ret;

@@ -268,7 +268,7 @@ static int image_parse_tree(struct ckpt_hdr *h, int fd, struct args *args)
  if (ret <= 0)
   return -1;

- pp = (struct ckpt_pids *) h;
+ pp = (struct ckpt_task_pids *) h;

  if (args->show_task_tree) {
   for (i = 0; i < nr_tasks; i++) {
--
1.7.1

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

Subject: [PATCH 10/11] restart: fix support for nested pid namespaces
Posted by Oren Laadan on Mon, 07 Feb 2011 17:21:31 GMT
View Forum Message <> Reply to Message

Adapt restart code to the new pids handling in kernel-cr that handles
pids as a proper shared object.

DISCLAIMER Disclaimer: this patch is bug and intrusive ... Here is a
summary of the changes that it makes:

1) The main change is that we read the 'ckpt_pids' that hold the
 actual pids numbers, and then everything else uses tags that refer to
 these objects. Since the ctx->pids_arr is an array of variable-length
 entries, it is inconvenient ot point to it with an index. So we use
 another array, ctx->pids, that maps from a linear index to the offset
 in the ctx->pids_arr where the data is found.

2) Now all pids other than those in 'ckpt_pids' are indices into that
 array (more precisely, into ctx->pids array), the variables now have
 a "_ind" suffix, e.g. "pid_ind" instead of "pid". There are helpers
 to translate from index to pids structure.

3) Document the data structures used to track pids and tasks within
the restart code.

4) To support (linearly) nested-pids, the pids hash table was extended
to have depth, so that if we need to allocate a new (dummy) pid, we
can choose unique pids at all pid-ns levels, not just the top.

5) Accordingly, dummy pid allocation is done at all possible depths in
 the hash.

6) Throw away ckpt_{read/write/assign}_vpids - it is no longer needed.
Instead, the seuqence of calls is now:
 ckpt_read_pids()
 ckpt_read_tree()
 ckpt_build_pids()
 ckpt_build_tree()

7) Disallow restart with --no-pids if there are nested pid-ns, because
 because is it quite complicated to find ou the pids of all tasks at
 all nested levels from userspace.

8) If the root task's is not a session leader (must be from a subtree
 checkpoint), then it should now inherit its sid from the coordinator.
 Furthermore, other tasks with sid/pgid inherited from above the root
 task should also do the same. For this to work we use a special value
 for their {sid,pgid}_ind: we can't use 0, because that already means
 a pid from an ancestor pid-ns; instead we mark it with CKPT_PID_ROOT,
 and the kernel code knows how to handle it.

 NOTE: this is only necessary when the root task is not a session
 leader. Otherwise, we can just add a placeholder task to accopmlish
 the same effect (recall it's a subtree). But a placeholder cannot be
 placed above the root task...

 NOTE2: by doing this, we squash all the sids/pgids from above the

root task into a single common value at restart, even though they
may have been distinct at checkpoint. This is considered a feature
until someone really needs this to behave differently ...

9) Fix a subtle bug in the session-propagation logic, whereas we don't
need a placeholder if we reach the root task _and_ we are a child of
the root task (becaus we will inherit the sid from the root task).

10) In ckpt_fork_child() we can use the 'ckpt_pids' structure for the
pids rather than manually build one.

11) In adjust_pids() and --no-pids we only try to update the numbers[0]
of the pid; we don't support nested pid-ns for --no-pids.

Cc: Sukadev Bhattiprolu <sukadev@linux.vnet.ibm.com>
Signed-off-by: Oren Laadan <orenl@cs.columbia.edu>
---
restart.c | 1192 ++---------- -
1 files changed, 787 insertions(+), 405 deletions(-)

diff --git a/restart.c b/restart.c
index 01566c2..f64e508 100644
--- a/restart.c
+++ b/restart.c
@@ -45,6 +45,12 @@
#include "common.h"

/*
+ * To re-create the tasks tree in user space, 'restart' reads the
+ * header and tree data from the checkpoint image tree. It makes up
+ * for the data that was consumed by using a helper process that
+ * provides the data back to the restart syscall, followed by the rest
+ * of the checkpoint image stream.
+ *
 * By default, 'restart' creates a new pid namespace in which the
 * restart takes place, using the original pids from the time of the
 * checkpoint. This requires that CLONE_NEWPID and eclone() be enabled.
@@ -54,19 +60,15 @@
 * by default, 'restart' creates an equivalen tree without restoring
 * the original pids, assuming that the application can tolerate this.
 * For this, the 'ckpt_pids' array is transformed on-the-fly before it
- * is fed to the kernel.
+ * is fed to the kernel. This mode of operation is permitted only if
+ * all the restarting tasks belong to a single pid-namespace (i.e. no
+ * pid-namespace nesting).
 *
- * By default, "--pids" implied "--pidns" and vice-versa. The user can
+ * By default, "--pids" implies "--pidns" and vice-versa. The user can

```
  * use "--pids --no-pidns" for a restart in the currnet namespace -
  * 'restart' will attempt to create the new tree with the original pids
  * from the time of the checkpoint, if possible. This requires that
  * eclone() be enabled.
- *
- * To re-create the tasks tree in user space, 'restart' reads the
- * header and tree data from the checkpoint image tree. It makes up
- * for the data that was consumed by using a helper process that
- * provides the data back to the restart syscall, followed by the rest
- * of the checkpoint image stream.
  */

 struct hashent {
@@ -78,6 +80,75 @@ struct hashent {
 struct task;
 struct ckpt_ctx;

+/*
+ * The following data structres are used to track pids:
+ *
+ * ctx->pids_arr[]:
+ *   Array of (variable sized) 'struct ckpt_pids' from the checkpoint
+ *   image, each entry indicates the level (depth) relative to the
+ *   root task, and the pids at each level. NOTE: the order of pids
+ *   matches order of adding them to the objhash during checkpoint
+ *   (hence their tags).
+ *
+ * ctx->pids_copy[]:
+ *   Array used to hold a copy of pids_arr[] during --no-pids restart
+ *   when converting the task's pids from the original values from
+ *   the checkpoint image, to the real pids produced by forks.
+ *
+ * ctx->pids_new[]:
+ *   Array of (variable sized) 'struct ckpt_pids' to hold new pids
+ *   objects allocated by the MakeForst algorithm fo the restart.
+ *
+ * ctx->pids_index[]:
+ *   Array of integers that provides mapping from a pid object (tag)
+ *   to the byte offset inside ctx->pids_arr where that pid object
+ *   is. It is useful since the entries in the latter are of variable
+ *   size.
+ *
+ * ctx->tasks_arr[]:
+ *   Array of 'struct ckpt_task_pids' from the checkpoint image, each
+ *   entry indicates a task's pids (pid,tgid,pgid,sid,ppid) and the
+ *   pid-namespace nesting level. NOTE: the pids store the tags of the
+ *   corresponding pid objects (and thus their order in ctx->pids_arr)
+ *   rather then the pid values themselves.
```

```
+ *
+ * ctx->tasks[]:
+ *   Array of 'struct task' that holds information about all the tasks
+ *   neede to be created in userespace (the input and output of the
+ *   DumpForst and CreateForest algorithms). NOTE: the pids here also
+ *   store the tags of the corresponding pid objects).
+ *
+ * When restart algorithm needs to create dead tasks or produce dummy
+ * tasks, it stores new 'ckpt_pids' objects in ctx->pids_new[], and
+ * extends ctx->pids[] and ctx->tasks[] to store index to new pids
+ * and new tasks, respectively.
+ *
+ * ctx->pids_nr:  (original) size of ctx->pids_arr
+ * ctx->pids_cnt: current size of ctx->pids_index
+ * ctx->pids_max: maximum size of ctx->pids_index
+ * ctx->pids_off: current offset in ctx->pids_new[]
+ * ctx->pids_len: maximum offset in ctx->pids_new[]
+ *
+ * ctx->tasks_nr:  size of ctx->pids_arr
+ * ctx->tasks_cnt: current size of ctx->tasks
+ * ctx->tasks_max: maximum size of ctx->tasks
+ *
+ * Given a byte offset in ctx->pids_arr, to get the 'ckpt_pids':
+ *   pids = pid_at_index(ctx, @offset)
+ *
+ * Given a pid-index from ctx->tasks/ctx->tasks_arr, to get the byte
+ * offset of the matching 'ckpt_pids' in ctx->pids_arr:
+ *   ctx->pids_index[@index]
+ *
+ * And to get the 'ckpt_pids' from an index:
+ *   pids = pids_of_index(@index)
+ *
+ *
+ * ctx->tasks_pids[]:
+ *   Array of pid values indicating the next hint for pid allocation
+ *   at each nesting level of pid-namespace.
+ */
+
 struct task {
  int flags;  /* state and (later) actions */

@@ -91,13 +162,12 @@ struct task {
  int vidx;  /* index into vpid array, -1 if none */
  int piddepth;

- pid_t pid;  /* process IDs, our bread-&-butter */
- pid_t ppid;
- pid_t tgid;
```

```
- pid_t sid;
+ /* Following are INDEX values into ctx->pids_index */
+ int pid_ind;  /* process IDs, our bread-&-butter */
+ int ppid_ind;
+ int tgid_ind;
+ int sid_ind;

- pid_t rpid;  /* [restart without vpids] actual (real) pid */
-
  struct ckpt_ctx *ctx; /* points back to the c/r context */

  pid_t real_parent; /* pid of task's real parent */
@@ -127,32 +197,45 @@ struct ckpt_ctx {
  int error;
  int success;

- pid_t root_pid;
  int pipe_in;
  int pipe_out;
- int pids_nr;
- int vpids_nr;

  int pipe_child[2]; /* for children to report status */
  int pipe_feed[2]; /* for feeder to provide input */
  int pipe_coord[2]; /* for coord to report status (if needed) */

+ int root_pid;
+ int pid_offset;
+
  struct ckpt_pids *pids_arr;
- struct ckpt_pids *copy_arr;
- __s32 *vpids_arr;
+ struct ckpt_pids *pids_new;
+ struct ckpt_pids *pids_copy;
+ int *pids_index;
+
+ int pids_nr;
+ int vpids_nr;
+ int pids_cnt;
+ int pids_max;
+ int pids_off;
+ int pids_len;

+ struct ckpt_task_pids *tasks_arr;
  struct task *tasks;
+
  int tasks_nr;
+ int tasks_cnt;
```

```
  int tasks_max;
- int tasks_pid;

- struct hashent **hash_arr;
+ /* an array of pid hash-tables: one hash-table per pidns level */
+ struct hashent ***hash_arr;
+ int *hash_last_pid;
+ int hash_depth;

  char header[BUFSIZE];
  char header_arch[BUFSIZE];
  char container[BUFSIZE];
  char tree[BUFSIZE];
- char vpids[BUFSIZE];
+ char pids[BUFSIZE];
  char buf[BUFSIZE];

  struct cr_restart_args *args;
@@ -194,9 +277,9 @@ int global_send_sigint = -1;
 static int ckpt_remount_proc(struct ckpt_ctx *ctx);
 static int ckpt_remount_devpts(struct ckpt_ctx *ctx);

+static int ckpt_build_pids(struct ckpt_ctx *ctx);
 static int ckpt_build_tree(struct ckpt_ctx *ctx);
 static int ckpt_init_tree(struct ckpt_ctx *ctx);
-static int assign_vpids(struct ckpt_ctx *ctx);
 static int ckpt_set_creator(struct ckpt_ctx *ctx, struct task *task);
 static int ckpt_placeholder_task(struct ckpt_ctx *ctx, struct task *task);
 static int ckpt_propagate_session(struct ckpt_ctx *ctx, struct task *session);
@@ -219,8 +302,8 @@ static int ckpt_write_obj(struct ckpt_ctx *ctx, struct ckpt_hdr *h);
 static int ckpt_write_header(struct ckpt_ctx *ctx);
 static int ckpt_write_header_arch(struct ckpt_ctx *ctx);
 static int ckpt_write_container(struct ckpt_ctx *ctx);
+static int ckpt_write_pids(struct ckpt_ctx *ctx);
 static int ckpt_write_tree(struct ckpt_ctx *ctx);
-static int ckpt_write_vpids(struct ckpt_ctx *ctx);

 static int _ckpt_read(int fd, void *buf, int count);
 static int ckpt_read(int fd, void *buf, int count);
@@ -232,12 +315,14 @@ static int ckpt_read_header(struct ckpt_ctx *ctx);
 static int ckpt_read_header_arch(struct ckpt_ctx *ctx);
 static int ckpt_read_container(struct ckpt_ctx *ctx);
 static int ckpt_read_tree(struct ckpt_ctx *ctx);
-static int ckpt_read_vpids(struct ckpt_ctx *ctx);
+static int ckpt_read_pids(struct ckpt_ctx *ctx);

 static int hash_init(struct ckpt_ctx *ctx);
 static void hash_exit(struct ckpt_ctx *ctx);
```

```
-static int hash_insert(struct ckpt_ctx *ctx, long key, void *data);
 static void *hash_lookup(struct ckpt_ctx *ctx, long key);
+static void *hash_lookup_level(struct ckpt_ctx *ctx, long key, int level);
+static void *hash_lookup_ind(struct ckpt_ctx *ctx, int n);
+static int hash_insert(struct ckpt_ctx *ctx, long key, void *data, int level);

 static inline pid_t _gettid(void)
 {
@@ -533,14 +618,20 @@ static void exit_ctx(struct ckpt_ctx *ctx)
 {
  if (ctx->freezer)
   free(ctx->freezer);
+
  if (ctx->tasks)
   free(ctx->tasks);
+ if (ctx->tasks_arr)
+  free(ctx->tasks_arr);
+
+ if (ctx->pids_index)
+  free(ctx->pids_index);
  if (ctx->pids_arr)
   free(ctx->pids_arr);
- if (ctx->copy_arr)
-  free(ctx->copy_arr);
- if (ctx->vpids_arr)
-  free(ctx->vpids_arr);
+ if (ctx->pids_new)
+  free(ctx->pids_new);
+ if (ctx->pids_copy)
+  free(ctx->pids_copy);

  /* unused fd will be silently ignored */
  close(ctx->pipe_in);
@@ -553,6 +644,57 @@ static void exit_ctx(struct ckpt_ctx *ctx)
  close(ctx->pipe_coord[1]);
 }

+static inline struct task *ckpt_init_task(struct ckpt_ctx *ctx)
+{
+ return &ctx->tasks[0];
+}
+
+static inline struct ckpt_pids *pids_at_offset(struct ckpt_ctx *ctx, int n)
+{
+ return (struct ckpt_pids *)(((char *) ctx->pids_arr) + n);
+}
+
+static inline struct ckpt_pids *pids_copy_at_offset(struct ckpt_ctx *ctx, int n)
```

```
+{
+ return (struct ckpt_pids *)(((char *) ctx->pids_copy) + n);
+}
+
+static inline struct ckpt_pids *pids_new_at_offset(struct ckpt_ctx *ctx, int n)
+{
+ return (struct ckpt_pids *)(((char *) ctx->pids_new) + n);
+}
+
+struct ckpt_pids pids_zero = {
+ .depth = 0,
+ .numbers = {0},
+};
+
+struct ckpt_pids pids_root = {
+ .depth = 0,
+ .numbers = {-1},
+};
+
+static inline struct ckpt_pids *pids_of_index(struct ckpt_ctx *ctx, int n)
+{
+ if (n == 0)
+  return &pids_zero;
+ else if (n == CKPT_PID_ROOT)
+  return &pids_root;
+ if (n < 0 || n > ctx->pids_cnt) {
+  errno = EINVAL;
+  return NULL;
+ }
+ if (n <= ctx->pids_nr)
+  return pids_at_offset(ctx, ctx->pids_index[n]);
+ else
+  return pids_new_at_offset(ctx, ctx->pids_index[n]);
+}
+
+static inline pid_t pid_at_index(struct ckpt_ctx *ctx, int n)
+{
+ return pids_of_index(ctx, n)->numbers[0];
+}
+
 int cr_restart(struct cr_restart_args *args)
 {
  struct ckpt_ctx ctx;
@@ -562,11 +704,10 @@ int cr_restart(struct cr_restart_args *args)

  ctx.args = args;
  ctx.whoami = CTX_RESTART;  /* for sanity checked */
- ctx.tasks_pid = CKPT_RESERVED_PIDS;
```

```
  ret = process_args(args);
  if (ret < 0)
-  return -1;
+  return ret;

  /* freezer preparation */
  if (args->freezer && freezer_prepare(&ctx) < 0)
@@ -621,13 +762,13 @@ int cr_restart(struct cr_restart_args *args)
   goto cleanup;
  }

- ret = ckpt_read_tree(&ctx);
+ ret = ckpt_read_pids(&ctx);
  if (ret < 0) {
-  ckpt_perror("read c/r tree");
+  ckpt_perror("read c/r pids");
   goto cleanup;
  }

- ret = ckpt_read_vpids(&ctx);
+ ret = ckpt_read_tree(&ctx);
  if (ret < 0) {
   ckpt_perror("read c/r tree");
   goto cleanup;
@@ -636,12 +777,12 @@ int cr_restart(struct cr_restart_args *args)
  /* build creator-child-relationship tree */
  if (hash_init(&ctx) < 0)
   goto cleanup;
- ret = ckpt_build_tree(&ctx);
- hash_exit(&ctx);
+
+ ret = ckpt_build_pids(&ctx);
  if (ret < 0)
   goto cleanup;

- ret = assign_vpids(&ctx);
+ ret = ckpt_build_tree(&ctx);
  if (ret < 0)
   goto cleanup;

@@ -660,7 +801,7 @@ int cr_restart(struct cr_restart_args *args)
  if (ctx.args->root)
   ctx.tasks[0].flags |= TASK_NEWROOT;

- if (ctx.args->pidns && ctx.tasks[0].pid != 1) {
+ if (ctx.args->pidns && pid_at_index(&ctx, ctx.tasks[0].pid_ind) != 1) {
   ckpt_dbg("new pidns without init\n");
```

```
   if (global_send_sigint == -1)
    global_send_sigint = SIGINT;
@@ -697,6 +838,7 @@ int cr_restart(struct cr_restart_args *args)
   ret = -1;

  cleanup:
+ hash_exit(&ctx);
  exit_ctx(&ctx);

  /* feeder doesn't exit - to avoid SIGCHILD to coordinator */
@@ -1085,7 +1227,7 @@ static int ckpt_coordinator(struct ckpt_ctx *ctx)
   ret = 0;
  }

- if (ctx->args->pidns && ctx->tasks[0].pid != 1) {
+ if (ctx->args->pidns && ctx->tasks[0].pid_ind != 1) {
   /* Report success/failure to the parent */
   if (ret < 0)
    ret = ctx->error;
@@ -1118,9 +1260,78 @@ static int ckpt_coordinator(struct ckpt_ctx *ctx)
  return ret;
 }

-static inline struct task *ckpt_init_task(struct ckpt_ctx *ctx)
+/*
+ * ckpt_build_pids - prepare pids array: this array will index into
+ * the pids_arr array pointing at the beginning of the individual
+ * 'struct ckpt_pids' elements there (as they are of variable size)
+ */
+static int ckpt_build_pids(struct ckpt_ctx *ctx)
 {
- return (&ctx->tasks[0]);
+ struct ckpt_pids *pids;
+ int i = 0, n = 0;
+ int depth = 0;
+ int s, len;
+
+ /*
+  * Allow for additional pids to be added on demand for
+  * placeholder tasks (each session leader may have at most
+  * one) Added +1 because index count starts from 1.
+  */
+ ctx->pids_max = ctx->pids_nr * 2;
+ ctx->pids_index = malloc(sizeof(int) * (ctx->pids_max + 1));
+ if (!ctx->pids_index) {
+  ckpt_perror("malloc pids index");
+  return ctx_set_errno(ctx);
+ }
```

```
+
+ len = ctx->pids_nr * sizeof(*pids) + ctx->vpids_nr * sizeof(__s32);
+ if (len <= 0)   /* overflow ? */
+  return ctx_ret_errno(ctx, EOVERFLOW);
+
+ while (n < ctx->pids_nr && len > 0) {
+  s = sizeof(*pids);
+
+  pids = pids_at_offset(ctx, i);
+  if (pids->depth < 0)
+   return ctx_ret_errno(ctx, EINVAL);
+  s += pids->depth * sizeof(__s32);
+  if (s > len)
+   return ctx_ret_errno(ctx, EINVAL);
+
+  depth += pids->depth;
+  if (depth > ctx->vpids_nr)
+   return ctx_ret_errno(ctx, EINVAL);
+
+  ctx->pids_index[n + 1] = i;
+
+  len -= s;
+  i += s;
+  n++;
+ }
+
+ if (n != ctx->pids_nr || depth != ctx->vpids_nr || len != 0)
+  return ctx_ret_errno(ctx, EINVAL);
+
+ ctx->pids_cnt = ctx->pids_nr;
+
+ if (!ctx->args->pidns && depth > 0) {
+  ckpt_err("need --pidns for nested pidns container");
+  return ctx_ret_errno(ctx, EINVAL);
+ }
+
+#ifdef CHECKPOINT_DEBUG
+ ckpt_dbg("====== PIDS\n");
+ for (n = 1; n <= ctx->pids_nr; n++) {
+  pids = pids_of_index(ctx, n);
+  ckpt_dbg("\t[%d] depth %d pids", n, pids->depth);
+  for (i = 0; i <= pids->depth; i++)
+   ckpt_dbg_cont(" %d", pids->numbers[i]);
+  ckpt_dbg_cont("\n");
+ }
+ ckpt_dbg(".....\n");
+#endif
+
```

```
+ return 0;
 }

 /*
@@ -1134,15 +1345,31 @@ static int ckpt_build_tree(struct ckpt_ctx *ctx)

  /*
   * Allow for additional tasks to be added on demand for
-  * referenced pids of dead tasks (each task can introduce at
-  * most two: session and process group IDs), as well as for
-  * placeholder tasks (each session id may have at most one)
+  * placeholder tasks (tgid/sid/pgid ids may each add one)
   */
  ctx->tasks_max = ctx->pids_nr * 4;
  ctx->tasks = malloc(sizeof(*ctx->tasks) * ctx->tasks_max);
  if (!ctx->tasks) {
   ckpt_perror("malloc tasks array");
-  return -1;
+  return ctx_set_errno(ctx);
+ }
+
+ /*
+  * In subtree checkpoint, the pids objects are the first to
+  * enter the objhash, and get their tags counting from 1.  In
+  * container checkpoint, other objects are inserted first, but
+  * we still assume they count from 1 - so we afjust all the
+  * pids indices by -offset.
+  */
+ if (ctx->pid_offset) {
+  for (i = 0; i < ctx->tasks_nr; i++) {
+   ctx->tasks_arr[i].vpid -= ctx->pid_offset;
+   ctx->tasks_arr[i].vtgid -= ctx->pid_offset;
+   if (ctx->tasks_arr[i].vpgid > 0)
+    ctx->tasks_arr[i].vpgid -= ctx->pid_offset;
+   if (ctx->tasks_arr[i].vsid > 0)
+    ctx->tasks_arr[i].vsid -= ctx->pid_offset;
+  }
  }

  /* initialize tree */
@@ -1150,7 +1377,7 @@ static int ckpt_build_tree(struct ckpt_ctx *ctx)
   return -1;

  /* assign a creator to each task */
- for (i = 0; i < ctx->tasks_nr; i++) {
+ for (i = 0; i < ctx->tasks_cnt; i++) {
   task = &ctx->tasks[i];
   if (task->creator)
```

```
    continue;
@@ -1160,17 +1387,26 @@ static int ckpt_build_tree(struct ckpt_ctx *ctx)

 #ifdef CHECKPOINT_DEBUG
  ckpt_dbg("====== TASKS\n");
- for (i = 0; i < ctx->tasks_nr; i++) {
+ for (i = 0; i < ctx->tasks_cnt; i++) {
   task = &ctx->tasks[i];
-  ckpt_dbg("\t[%d] pid %d ppid %d sid %d creator %d",
-    i, task->pid, task->ppid, task->sid,
-    task->creator->pid);
+  ckpt_dbg("\t[%3d] pid %5d(%3d) (tgid %3d) ppid %5d(%3d)"
+    "  (pgid %3d) sid %5d(%3d) creator %5d(%3d)",
+    i, pid_at_index(ctx, task->pid_ind), task->pid_ind,
+    i < ctx->tasks_nr ? ctx->tasks_arr[i].vtgid : -1,
+    pid_at_index(ctx, task->ppid_ind), task->ppid_ind,
+    i < ctx->tasks_nr ? ctx->tasks_arr[i].vpgid : -1,
+    task->sid_ind >= 0 ?
+    pid_at_index(ctx, task->sid_ind) : -1, task->sid_ind,
+    pid_at_index(ctx, task->creator->pid_ind));
   if (task->next_sib)
-   ckpt_dbg_cont(" next %d", task->next_sib->pid);
+   ckpt_dbg_cont(" next %3d",
+         pid_at_index(ctx, task->next_sib->pid_ind));
   if (task->prev_sib)
-   ckpt_dbg_cont(" prev %d", task->prev_sib->pid);
+   ckpt_dbg_cont(" prev %3d",
+         pid_at_index(ctx, task->prev_sib->pid_ind));
   if (task->phantom)
-   ckpt_dbg_cont(" placeholder %d", task->phantom->pid);
+   ckpt_dbg_cont(" placeholder %3d",
+         pid_at_index(ctx, task->phantom->pid_ind));
   ckpt_dbg_cont(" %c%c%c%c%c%c",
         (task->flags & TASK_THREAD) ? 'T' : ' ',
         (task->flags & TASK_SIBLING) ? 'P' : ' ',
@@ -1202,27 +1438,48 @@ static int ckpt_build_tree(struct ckpt_ctx *ctx)
    return ctx_ret_errno(ctx, EINVAL);
  }

+ /*
+  * Restore the original pid indices if they were adjusted above
+  * because the feeder will feed this array to the kernel.
+  */
+ if (ctx->pid_offset) {
+  for (i = 0; i < ctx->tasks_nr; i++) {
+   ctx->tasks_arr[i].vpid += ctx->pid_offset;
+   ctx->tasks_arr[i].vtgid += ctx->pid_offset;
+   if (ctx->tasks_arr[i].vpgid > 0)
```

```
+    ctx->tasks_arr[i].vpgid += ctx->pid_offset;
+   if (ctx->tasks_arr[i].vsid > 0)
+    ctx->tasks_arr[i].vsid += ctx->pid_offset;
+  }
+ }
+
  return 0;
 }

-static int ckpt_setup_task(struct ckpt_ctx *ctx, pid_t pid, pid_t ppid)
+static int ckpt_setup_task(struct ckpt_ctx *ctx, int pid_ind, int ppid_ind)
 {
  struct task *task;
+ struct ckpt_pids *pids;
+ int j;

- if (pid == 0)  /* ignore if outside namespace */
+ /* ignore if outside namespace */
+ if (pid_ind == 0 || pid_ind == CKPT_PID_ROOT)
   return 0;

- if (hash_lookup(ctx, pid))  /* already handled */
+ pids = pids_of_index(ctx, pid_ind);
+
+ /* skip if already handled */
+ if (hash_lookup(ctx, pids->numbers[0]))
   return 0;

- task = &ctx->tasks[ctx->tasks_nr++];
+ task = &ctx->tasks[ctx->tasks_cnt++];

  task->flags = TASK_GHOST;

- task->pid = pid;
- task->ppid = ppid;
- task->tgid = pid;
- task->sid = ppid;
+ task->pid_ind = pid_ind;
+ task->ppid_ind = ppid_ind;
+ task->tgid_ind = pid_ind;
+ task->sid_ind = ppid_ind;

  task->children = NULL;
  task->next_sib = NULL;
@@ -1230,20 +1487,20 @@ static int ckpt_setup_task(struct ckpt_ctx *ctx, pid_t pid, pid_t ppid)
  task->creator = NULL;
  task->phantom = NULL;
```

```
- task->rpid = -1;
  task->ctx = ctx;

- if (hash_insert(ctx, pid, task) < 0)
-  return -1;
-
- /* remember the max pid seen */
- if (task->pid > ctx->tasks_pid)
-  ctx->tasks_pid = task->pid;
+ for (j = 0; j <= pids->depth; j++) {
+  if (hash_insert(ctx, pids->numbers[j], task, j) < 0)
+   return -1;
+  /* remember the max pid seen */
+  if (pids->numbers[j] > ctx->hash_last_pid[j])
+   ctx->hash_last_pid[j] = pids->numbers[j];
+ }

  return 0;
 }

-static int ckpt_valid_pid(struct ckpt_ctx *ctx, pid_t pid, char *which, int i)
+static int _ckpt_valid_pid(struct ckpt_ctx *ctx, pid_t pid, char *which, int i)
 {
  if (pid < 0) {
   ckpt_err("Invalid %s %d (for task#%d)\n", which, pid, i);
@@ -1264,104 +1521,161 @@ static int ckpt_valid_pid(struct ckpt_ctx *ctx, pid_t pid, char
*which, int i)
  return 1;
 }

-static int ckpt_alloc_pid(struct ckpt_ctx *ctx)
+static int ckpt_valid_pid(struct ckpt_ctx *ctx, int index, char *which, int i)
 {
- int n = 0;
+ struct ckpt_pids *pids;
+ int j;
+
+ pids = pids_of_index(ctx, index);
+ if (pids == NULL)
+  return 0;
+ for (j = 0; j <= pids->depth; j++) {
+  if (!_ckpt_valid_pid(ctx, pids->numbers[j], which, i))
+   return 0;
+ }
+ return 1;
+}
+
+static int ckpt_alloc_pid(struct ckpt_ctx *ctx, int depth)
```

```
+{
+ struct ckpt_pids *pids;
+ int j, n, last, len;
+ int pid_ind;
+
+ len = sizeof(*pids) + depth * sizeof(__s32);
+
+ /* need to expand the ctx->pids_new[] array ? */
+ if (ctx->pids_off + len > ctx->pids_len) {
+  pids = realloc(ctx->pids_new, ctx->pids_len * 3 / 2);
+  if (!pids) {
+   ckpt_perror("allocate new pids table");
+   return ctx_set_errno(ctx);
+  }
+  ctx->pids_new = pids;
+ }
+
+ /* need to expand the ctx->pids_index[] array ? */
+ if (ctx->pids_cnt >= ctx->pids_max)  {
+  /* shouldn't happen, beacuse we prepared enough */
+  ckpt_err("out of space in task table !");
+  return ctx_ret_errno(ctx, EOVERFLOW);
+ }
+
+ ctx->pids_cnt += 1;
+ pid_ind = ctx->pids_cnt;
+
+ ctx->pids_index[pid_ind] = ctx->pids_off;
+ ctx->pids_off += len;

  /*
-  * allocate an unused pid for the placeholder
+  * allocate an unused pid for the placeholder in each pid-namespace
   * (this will become inefficient if pid-space is exhausted)
   */
- do {
-  if (ctx->tasks_pid == ctx->pid_max)
-   ctx->tasks_pid = CKPT_RESERVED_PIDS;
-  else
-   ctx->tasks_pid++;

-  if (n++ == ctx->pid_max) { /* ohhh... */
-   ckpt_err("pid namsepace exhausted");
-   return -1;
-  }
- } while (hash_lookup(ctx, ctx->tasks_pid));
+ pids = pids_of_index(ctx, pid_ind);
```

```
- return ctx->tasks_pid;
-}
+ for (j = 0; j <= depth; j++) {
+  n = 0;
+  last = ctx->hash_last_pid[j];

-static int ckpt_zero_pid(struct ckpt_ctx *ctx)
-{
- pid_t pid;
+  do {
+   if (last >= ctx->pid_max)
+    last = CKPT_RESERVED_PIDS;
+   else
+    last++;

- pid = ckpt_alloc_pid(ctx);
- if (pid < 0)
-  return -1;
- if (ckpt_setup_task(ctx, pid, ctx->pids_arr[0].vpid) < 0)
-  return -1;
- return pid;
+   if (n++ == ctx->pid_max) {  /* ohhh... */
+    ckpt_err("pid namsepace exhausted");
+    return ctx_ret_errno(ctx, EOVERFLOW);
+   }
+  } while (hash_lookup_level(ctx, last, j));
+
+  ctx->hash_last_pid[j] = last;
+  pids->numbers[j] = last;
+ }
+
+ return pid_ind;
 }

 static int ckpt_init_tree(struct ckpt_ctx *ctx)
 {
- struct ckpt_pids *pids_arr = ctx->pids_arr;
- int pids_nr = ctx->pids_nr;
+ struct ckpt_task_pids *tasks_arr;
+ struct ckpt_pids *pids;
  struct task *task;
- pid_t root_pid;
- pid_t root_sid;
- pid_t zero_pid = 0;
- int i;
+ int root_pid_ind;
+ int root_sid_ind;
+ int ppid_ind;
```

```
+ int i, j;
+
+ tasks_arr = ctx->tasks_arr;

- root_pid = pids_arr[0].vpid;
- root_sid = pids_arr[0].vsid;
+ root_pid_ind = tasks_arr[0].vpid;
+ root_sid_ind = tasks_arr[0].vsid;

  /*
+  * Any zero value (tgid/pgid/sid) means that at checkpoint the
+  * original pid came from an ancestor pid-ns. If we find any,
+  * the caller must also have requested --pidns, otherwise fail.
+  * This is done in _ckpt_valid_pid(), but users can choose to
+  * only issue a warning, and we'll convert them to inherit the
+  * sid of the root task instead.
+  *
   * The case where root_sid != root_pid is special. It must be
   * from a subtree checkpoint (in container, root_sid is either
   * same as root_pid or 0), and root_sid was inherited from an
   * ancestor of that subtree.
   *
-  * If we restart with --pidns, make the root-task also inherit
-  * sid from its ancestor (== coordinator), whatever 'restart'
-  * task currently has.  For that, we force the root-task's sid
-  * and all references to it from other tasks (via sid and
-  * pgid), to 0. Later, the feeder will substitute the
-  * cooridnator's sid for them.
+  * When creating the tasks tree, the root task will inherit
+  * its sid from its ancestor (usually from the coordinator;
+  * however, for --pidns and non-init root task it will be the
+  * stub init task inserted by us). This sid will be whatever
+  * 'restart' process (or our caller) current has.
+  *
+  * For that, we force the root-task's sid and all references
+  * to it to be CKPT_PID_ROOT. This tells restart to treat them
+  * as such, and ensures that we don't call setsid() on the
+  * root task (because sid != pid). CKPT_PID_ROOT is gracefully
+  * handled both by ckpt_set_creator() when tracking the sid
+  * heritage, and by the kernel when restoring a task's pgid.
   *
   * (Note that this still works even if the coordinator's sid
   * is "used" by a restarting task: a new-pidns restart will
   * fail because the pid is in use, and in an old-pidns restart
   * the task will be assigned a new pid anyway).
-  *
-  * If we restart with --no-pidns, we'll add a ghost task below
-  * whose pid will be used instead of these zeroed entried.
```

```
   */

  /* forcing root_sid to -1, will make comparisons below fail */
- if (root_sid == root_pid)
-  root_sid = -1;
+ if (root_sid_ind == root_pid_ind)
+  root_sid_ind = -1;

  /* populate with known tasks */
- for (i = 0; i < pids_nr; i++) {
+ for (i = 0; i < ctx->tasks_nr; i++) {
   task = &ctx->tasks[i];

   task->flags = 0;

-  if (!ckpt_valid_pid(ctx, pids_arr[i].vpid, "pid", i))
-   return -1;
-  else if (!ckpt_valid_pid(ctx, pids_arr[i].vtgid, "tgid", i))
-   return -1;
-  else if (!ckpt_valid_pid(ctx, pids_arr[i].vsid, "sid", i))
-   return -1;
-  else if (!ckpt_valid_pid(ctx, pids_arr[i].vpgid, "pgid", i))
-   return -1;
+  /* pid and tgid must be native in our namespace ! */
+  if (tasks_arr[i].vpid == 0 || tasks_arr[i].vtgid == 0)
+   return ctx_ret_errno(ctx, EINVAL);

-  if (pids_arr[i].vsid == root_sid)
-   pids_arr[i].vsid = 0;
-  if (pids_arr[i].vpgid == root_sid)
-   pids_arr[i].vpgid = 0;
+  if (!ckpt_valid_pid(ctx, tasks_arr[i].vpid, "pid", i))
+   return ctx_ret_errno(ctx, EINVAL);
+  else if (!ckpt_valid_pid(ctx, tasks_arr[i].vtgid, "tgid", i))
+   return ctx_ret_errno(ctx, EINVAL);
+  else if (!ckpt_valid_pid(ctx, tasks_arr[i].vsid, "sid", i))
+   return ctx_ret_errno(ctx, EINVAL);
+  else if (!ckpt_valid_pid(ctx, tasks_arr[i].vpgid, "pgid", i))
+   return ctx_ret_errno(ctx, EINVAL);
+
+  if (tasks_arr[i].vsid == root_sid_ind)
+   tasks_arr[i].vsid = CKPT_PID_ROOT;
+  if (tasks_arr[i].vpgid == root_sid_ind)
+   tasks_arr[i].vpgid = CKPT_PID_ROOT;

-  task->pid = pids_arr[i].vpid;
-  task->ppid = pids_arr[i].vppid;
-  task->tgid = pids_arr[i].vtgid;
```

```
-  task->sid = pids_arr[i].vsid;
+  task->pid_ind = tasks_arr[i].vpid;
+  task->ppid_ind = tasks_arr[i].vppid;
+  task->tgid_ind = tasks_arr[i].vtgid;
+  task->sid_ind = tasks_arr[i].vsid;

   task->children = NULL;
   task->next_sib = NULL;
@@ -1369,30 +1683,29 @@ static int ckpt_init_tree(struct ckpt_ctx *ctx)
   task->creator = NULL;
   task->phantom = NULL;

-  task->rpid = -1;
   task->ctx = ctx;

-  if (hash_insert(ctx, task->pid, task) < 0)
-   return -1;
+  pids = pids_of_index(ctx, tasks_arr[i].vpid);
+  for (j = 0; j <= pids->depth; j++) {
+   if (hash_insert(ctx, pids->numbers[j], task, j) < 0)
+    return -1;
+   /* remember the max pid seen */
+   if (pids->numbers[j] > ctx->hash_last_pid[j])
+    ctx->hash_last_pid[j] = pids->numbers[j];
+  }
  }

- ctx->tasks_nr = pids_nr;
+ ctx->tasks_cnt = ctx->tasks_nr;

  /* add pids unaccounted for (no tasks) */
- for (i = 0; i < pids_nr; i++) {
-  pid_t sid;
-
-  sid = pids_arr[i].vsid;
+ for (i = 0; i < ctx->tasks_nr; i++) {

-  /* Remember if we find any vsid/vpgid - see below */
-  if (pids_arr[i].vsid == 0 || pids_arr[i].vpgid == 0)
-   zero_pid = 1;
   /*
    * An unaccounted-for sid belongs to a task that was a
    * session leader and died. We can safe set its parent
    * (and creator) to be the root task.
    */
-  if (ckpt_setup_task(ctx, sid, root_pid) < 0)
+  if (ckpt_setup_task(ctx, tasks_arr[i].vsid, root_pid_ind) < 0)
    return -1;
```

```
  /*
@@ -1400,15 +1713,17 @@ static int ckpt_init_tree(struct ckpt_ctx *ctx)
   * ancestor of root_task, and more specifically, via
   * root_task itself: make root_task our parent.
   */
-  if (sid == 0)
-   sid = root_pid;
+
+  ppid_ind = tasks_arr[i].vsid;
+  if (ppid_ind == 0 || ppid_ind == CKPT_PID_ROOT)
+   ppid_ind = root_pid_ind;

  /*
   * If a pid belongs to a dead thread group leader, we
   * need to add it with the same sid as current (and
   * other) threads.
   */
-  if (ckpt_setup_task(ctx, pids_arr[i].vtgid, sid) < 0)
+  if (ckpt_setup_task(ctx, tasks_arr[i].vtgid, ppid_ind) < 0)
    return -1;

  /*
@@ -1419,36 +1734,16 @@ static int ckpt_init_tree(struct ckpt_ctx *ctx)
   * same sid as us: all tasks with same pgrp must have
   * their sid matching.
   */
-  if (ckpt_setup_task(ctx, pids_arr[i].vpgid, sid) < 0)
-   return -1;
- }
-
- /*
-  * Zero sid/pgid is disallowed in --no-pidns mode. If there
-  * were any, we invent a new ghost-zero task and substitute
-  * its pid for those any sid/pgid.
-  */
- if (zero_pid && !ctx->args->pidns) {
-  zero_pid = ckpt_zero_pid(ctx);
-  if (zero_pid < 0)
+  if (ckpt_setup_task(ctx, tasks_arr[i].vpgid, ppid_ind) < 0)
    return -1;
-  for (i = 0; i < pids_nr; i++) {
-   if (pids_arr[i].vsid == 0) {
-    pids_arr[i].vsid = zero_pid;
-    pids_arr[i].vppid = zero_pid;
-   }
-   if (pids_arr[i].vpgid == 0) {
-    pids_arr[i].vpgid = zero_pid;
```

```
-    pids_arr[i].vppid = zero_pid;
-   }
-  }
   }

   /* mark root task(s), and set its "creator" to be zero_task */
   ckpt_init_task(ctx)->flags |= TASK_ROOT;
   ckpt_init_task(ctx)->creator = &zero_task;

- ckpt_dbg("total tasks (including ghosts): %d\n", ctx->tasks_nr);
+ ckpt_dbg("total tasks (excluding ghosts): %d\n", ctx->tasks_nr);
+ ckpt_dbg("total tasks (including ghosts): %d\n", ctx->tasks_cnt);
   return 0;
  }

@@ -1517,50 +1812,67 @@ static int ckpt_init_tree(struct ckpt_ctx *ctx)
   * leader.  This is done using a placeholder in a manner similar to
   * how we handle orphans that are not session leaders.
   */
+
  static int ckpt_set_creator(struct ckpt_ctx *ctx, struct task *task)
  {
- struct task *session = hash_lookup(ctx, task->sid);
- struct task *parent = hash_lookup(ctx, task->ppid);
+ struct task *session;
+ struct task *parent;
   struct task *creator;

+ session = hash_lookup_ind(ctx, task->sid_ind);
+ parent = hash_lookup_ind(ctx, task->ppid_ind);
+
   if (task == ckpt_init_task(ctx)) {
-  ckpt_err("pid %d: logical error\n", ckpt_init_task(ctx)->pid);
-  return -1;
+  ckpt_err("pid %d: logical error\n",
+     pid_at_index(ctx, ckpt_init_task(ctx)->pid_ind));
+  return ctx_ret_errno(ctx, EINVAL);
   }

   /* sid == 0 must have been inherited from outside the container */
- if (task->sid == 0)
-  session = ckpt_init_task(ctx);
+ if (task->sid_ind == 0 || task->sid_ind == CKPT_PID_ROOT)
+  task->flags |= TASK_SESSION;

- if (task->tgid != task->pid) {
+ if (task->tgid_ind != task->pid_ind) {
    /* thread: creator is thread-group-leader */
```

```
-  ckpt_dbg("pid %d: thread tgid %d\n", task->pid, task->tgid);
-  creator = hash_lookup(ctx, task->tgid);
+  ckpt_dbg("pid %d: thread tgid %d\n",
+    pid_at_index(ctx, task->pid_ind),
+    pid_at_index(ctx, task->tgid_ind));
+  creator = hash_lookup_ind(ctx, task->tgid_ind);
   if (!creator) {
    /* oops... thread group leader MIA */
-   ckpt_err("pid %d: no leader %d\n", task->pid, task->tgid);
-   return -1;
+   ckpt_err("pid %d: no leader %d\n",
+     pid_at_index(ctx, task->pid_ind),
+     pid_at_index(ctx, task->tgid_ind));
+   return ctx_ret_errno(ctx, EINVAL);
   }
   task->flags |= TASK_THREAD;
- } else if (task->ppid == 0 || !parent) {
+ } else if (task->ppid_ind == 0 || !parent) {
   /* only root_task can have ppid == 0, parent must always exist */
-  ckpt_err("pid %d: invalid ppid %d\n", task->pid, task->ppid);
-  return -1;
- } else if (task->pid == task->sid) {
+  ckpt_err("pid %d: invalid ppid %d\n",
+    pid_at_index(ctx, task->pid_ind),
+    pid_at_index(ctx, task->ppid_ind));
+  return ctx_ret_errno(ctx, EINVAL);
+ } else if (task->pid_ind == task->sid_ind) {
   /* session leader: creator is parent */
-  ckpt_dbg("pid %d: session leader\n", task->pid);
+  ckpt_dbg("pid %d: session leader\n",
+    pid_at_index(ctx, task->pid_ind));
   creator = parent;
  } else if (task->flags & TASK_DEAD) {
   /* dead: creator is session leader */
-  ckpt_dbg("pid %d: task is dead\n", task->pid);
+  ckpt_dbg("pid %d: task is dead\n",
+    pid_at_index(ctx, task->pid_ind));
   creator = session;
- } else if (task->sid == parent->sid) {
+ } else if (task->sid_ind == parent->sid_ind) {
   /* (non-session-leader) inherit: creator is parent */
-  ckpt_dbg("pid %d: inherit sid %d\n", task->pid, task->sid);
+  ckpt_dbg("pid %d: inherit sid %d\n",
+    pid_at_index(ctx, task->pid_ind),
+    pid_at_index(ctx, task->sid_ind));
   creator = parent;
- } else if (task->ppid == 1) {
+ } else if (task->ppid_ind == 1) {
```

```
   /* (non-session-leader) orphan: creator is dummy */
-  ckpt_dbg("pid %d: orphan session %d\n", task->pid, task->sid);
+  ckpt_dbg("pid %d: orphan session %d\n",
+     pid_at_index(ctx, task->pid_ind),
+     pid_at_index(ctx, task->sid_ind));
   if (!session->phantom)
    if (ckpt_placeholder_task(ctx, task) < 0)
     return -1;
@@ -1570,27 +1882,31 @@ static int ckpt_set_creator(struct ckpt_ctx *ctx, struct task *task)
   if (!session->creator) {
    /* (non-session-leader) recursive: session's creator */
    ckpt_dbg("pid %d: recursive session creator %d\n",
-         task->pid, task->sid);
+     pid_at_index(ctx, task->pid_ind),
+     pid_at_index(ctx, task->sid_ind));
    if (ckpt_set_creator(ctx, session) < 0)
     return -1;
   }
   /* then use it to decide what to do */
-  if (session->creator->pid == task->ppid) {
+  if (session->creator->pid_ind == task->ppid_ind) {
    /* init must not be sibling creator (CLONE_PARENT) */
    if (session == ckpt_init_task(ctx)) {
     ckpt_err("pid %d: sibling session prohibited"
-          " with init as creator\n", task->pid);
-    return -1;
+      " with init as creator\n",
+      pid_at_index(ctx, task->pid_ind));
+    return ctx_ret_errno(ctx, EINVAL);
    }
    /* (non-session-leader) sibling: creator is sibling */
    ckpt_dbg("pid %d: sibling session %d\n",
-         task->pid, task->sid);
+     pid_at_index(ctx, task->pid_ind),
+     pid_at_index(ctx, task->sid_ind));
    creator = session;
    task->flags |= TASK_SIBLING;
   } else {
    /* (non-session-leader) session: fork before setsid */
    ckpt_dbg("pid %d: propagate session %d\n",
-         task->pid, task->sid);
+     pid_at_index(ctx, task->pid_ind),
+     pid_at_index(ctx, task->sid_ind));
    creator = parent;
    task->flags |= TASK_SESSION;
   }
@@ -1603,7 +1919,9 @@ static int ckpt_set_creator(struct ckpt_ctx *ctx, struct task *task)
   next->prev_sib = task;
```

```
  }

- ckpt_dbg("pid %d: creator set to %d\n", task->pid, creator->pid);
+ ckpt_dbg("pid %d: creator set to %d\n",
+    pid_at_index(ctx, task->pid_ind),
+    pid_at_index(ctx, creator->pid_ind));
  task->creator = creator;
  creator->children = task;

@@ -1616,26 +1934,29 @@ static int ckpt_set_creator(struct ckpt_ctx *ctx, struct task *task)

 static int ckpt_placeholder_task(struct ckpt_ctx *ctx, struct task *task)
 {
- struct task *session = hash_lookup(ctx, task->sid);
- struct task *holder = &ctx->tasks[ctx->tasks_nr++];
- pid_t pid;
+ struct task *session;
+ struct task *holder;
+ int pid_ind;
+
+ session = hash_lookup_ind(ctx, task->sid_ind);
+ holder = &ctx->tasks[ctx->tasks_cnt++];

- if (ctx->tasks_nr > ctx->tasks_max) {
+ if (ctx->tasks_cnt > ctx->tasks_max) {
    /* shouldn't happen, beacuse we prepared enough */
    ckpt_err("out of space in task table !");
-   return -1;
+   return ctx_ret_errno(ctx, EOVERFLOW);
  }

- pid = ckpt_alloc_pid(ctx);
- if (pid < 0)
+ pid_ind = ckpt_alloc_pid(ctx, pids_of_index(ctx, task->pid_ind)->depth);
+ if (pid_ind < 0)
    return -1;

  holder->flags = TASK_DEAD;

- holder->pid = pid;
- holder->ppid = ckpt_init_task(ctx)->pid;
- holder->tgid = pid;
- holder->sid = task->sid;
+ holder->pid_ind = pid_ind;
+ holder->ppid_ind = ckpt_init_task(ctx)->pid_ind;
+ holder->tgid_ind = pid_ind;
+ holder->sid_ind = task->sid_ind;
```

```
  holder->children = NULL;
  holder->next_sib = NULL;
@@ -1643,7 +1964,6 @@ static int ckpt_placeholder_task(struct ckpt_ctx *ctx, struct task_struct *task)
  holder->creator = NULL;
  holder->phantom = NULL;

- holder->rpid = -1;
  holder->ctx = ctx;

  holder->creator = session;
@@ -1671,28 +1991,57 @@ static int ckpt_placeholder_task(struct ckpt_ctx *ctx, struct task_struct
*task)

 static int ckpt_propagate_session(struct ckpt_ctx *ctx, struct task_struct *task)
 {
- struct task_struct *session = hash_lookup(ctx, task->sid);
+ struct task_struct *session;
  struct task_struct *creator;
- pid_t sid = task->sid;
+
+ session = hash_lookup_ind(ctx, task->sid_ind);
+
+ /*
+  * propagate the TASK_SESSION up the ancestry until we reach
+  * our session owner, so that all of them pass the sid we wnat
+  * down before (possibly) changing their own.
+  */

  do {
-  ckpt_dbg("pid %d: set session\n", task->pid);
+  ckpt_dbg("pid %d: set session\n",
+    pid_at_index(ctx, task->pid_ind));
   task->flags |= TASK_SESSION;

   creator = task->creator;
-  if (creator->pid == 1) {
+  /*
+   * If we reached the root task, and the root task is
+   * not our real parent, then we add a placeholder here
+   * as a child of the root task and our creator. The
+   * placeholder will inherit the session, and pass it
+   * to us (and then terminate).
+   */
+  if (creator == ckpt_init_task(ctx) &&
+      creator != hash_lookup_ind(ctx, task->ppid_ind)) {
    if (ckpt_placeholder_task(ctx, task) < 0)
     return -1;
   }
```

```
-  ckpt_dbg("pid %d: moving up to %d\n", task->pid, creator->pid);
+  ckpt_dbg("pid %d: moving up to %d\n",
+    pid_at_index(ctx, task->pid_ind),
+    pid_at_index(ctx, creator->pid_ind));
   task = creator;

   if(!task->creator) {
    if (ckpt_set_creator(ctx, task) < 0)
     return -1;
   }
- } while (task->sid != sid &&
+
+  /*
+   * (Note that now @task already points to our creator!)
+   * We don't propagate anymore if:
+   *
+   * (a) our creator has the same session as us
+   * (b) our creator is the root task of the restart
+   * (c) our creator already has TASK_SESSION,
+   * (d) our creator's creator is our session (leader)
+   */
+
+ } while (hash_lookup_ind(ctx, task->sid_ind) != session &&
    task != ckpt_init_task(ctx) &&
    !(task->flags & TASK_SESSION) &&
    task->creator != session);
@@ -1739,26 +2088,27 @@ static int ckpt_make_tree(struct ckpt_ctx *ctx, struct task *task)
  int ret;

  ckpt_dbg("pid %d: pid %d sid %d parent %d\n",
-        task->pid, _gettid(), getsid(0), getppid());
+   pid_at_index(ctx, task->pid_ind),
+   _gettid(), getsid(0), getppid());

  /* 1st pass: fork children that inherit our old session-id */
  for (child = task->children; child; child = child->next_sib) {
   if (child->flags & TASK_SESSION) {
    ckpt_dbg("pid %d: fork child %d with session\n",
-          task->pid, child->pid);
+     pid_at_index(ctx, task->pid_ind),
+     pid_at_index(ctx, child->pid_ind));
    newpid = ckpt_fork_child(ctx, child);
    if (newpid < 0)
-    return -1;
-   child->rpid = newpid;
+    return ctx_set_errno(ctx);
   }
```

```
  }

  /* change session id, if necessary */
- if (task->pid == task->sid) {
+ if (task->pid_ind == task->sid_ind) {
   ret = setsid();
-  if (ret < 0 && task != ckpt_init_task(ctx)) {
+  if (ret < 0) {
    ckpt_perror("setsid");
-   return -1;
+   return ctx_set_errno(ctx);
   }
  }

@@ -1766,11 +2116,11 @@ static int ckpt_make_tree(struct ckpt_ctx *ctx, struct task *task)
  for (child = task->children; child; child = child->next_sib) {
   if (!(child->flags & TASK_SESSION)) {
    ckpt_dbg("pid %d: fork child %d without session\n",
-           task->pid, child->pid);
+     pid_at_index(ctx, task->pid_ind),
+     pid_at_index(ctx, child->pid_ind));
    newpid = ckpt_fork_child(ctx, child);
    if (newpid < 0)
-    return -1;
-   child->rpid = newpid;
+    return ctx_set_errno(ctx);
   }
  }

@@ -1786,15 +2136,13 @@ static int ckpt_make_tree(struct ckpt_ctx *ctx, struct task *task)
   */

  /* communicate via pipe that all is well */
- swap.old = task->pid;
+ swap.old = pid_at_index(ctx, task->pid_ind);
  swap.new = _gettid();
  ret = write(ctx->pipe_out, &swap, sizeof(swap));
  if (ret != sizeof(swap)) {
   ckpt_perror("write swap");
-  return -1;
+  return ctx_ret_errno(ctx, EPIPE);
  }
- close(ctx->pipe_out);
- ctx->pipe_out = -1;  /* mark unused */

  /*
   * At this point restart may have already begun in the kernel.
@@ -1893,18 +2241,20 @@ int ckpt_fork_stub(void *data)
```

```
 static pid_t ckpt_fork_child(struct ckpt_ctx *ctx, struct task *child)
 {
  struct clone_args clone_args;
- genstack stk;
+ struct ckpt_pids *pids, *ppids;
  unsigned long flags = SIGCHLD;
+ pid_t *numbers;
  pid_t pid = 0;
- pid_t *pids = &pid;
- int i, depth;
+ genstack stk;
+ int j;

- ckpt_dbg("fork child vpid %d flags %#x\n", child->pid, child->flags);
+ ckpt_dbg("fork child vpid %d flags %#x\n",
+    pid_at_index(ctx, child->pid_ind), child->flags);

  stk = genstack_alloc(PTHREAD_STACK_MIN);
  if (!stk) {
   ckpt_perror("ckpt_fork_child genstack_alloc");
-  return -1;
+  return ctx_set_errno(ctx);
  }

  if (child->flags & TASK_THREAD)
@@ -1921,13 +2271,15 @@ static pid_t ckpt_fork_child(struct ckpt_ctx *ctx, struct task *child)

  memset(&clone_args, 0, sizeof(clone_args));
  clone_args.nr_pids = 1;
- /* select pid if --pids, otherwise it's 0 */
- if (ctx->args->pids) {
-  depth = child->piddepth + 1;
-  clone_args.nr_pids = depth;

-  pids = &ctx->vpids_arr[child->vidx];
+ pids = pids_of_index(ctx, child->pid_ind);
+ ppids = pids_of_index(ctx, child->creator->pid_ind);
+ numbers = pids_zero.numbers;

+ /* select pid if --pids, otherwise it's 0 */
+ if (ctx->args->pids) {
+  clone_args.nr_pids = pids->depth + 1;
+  numbers = pids->numbers;
 #ifndef CLONE_NEWPID
   if (child->piddepth > child->creator->piddepth) {
    ckpt_err("nested pidns but CLONE_NEWPID undefined");
@@ -1937,19 +2289,25 @@ static pid_t ckpt_fork_child(struct ckpt_ctx *ctx, struct task *child)
    ctx_ret_errno(ctx, ENOSYS);
```

```
   }
 #else /* CLONE_NEWPID */
-  if (child->piddepth > child->creator->piddepth) {
+  if (pids->depth > ppids->depth + 1) {
+    ckpt_err("unsupported form of pidns nesting");
+    ctx_ret_errno(ctx, ENOSYS);
+  }
+  if (pids->depth > ppids->depth) {
     child->flags |= TASK_NEWPID;
     flags |= CLONE_NEWPID;
     clone_args.nr_pids--;
+    numbers = pids->numbers + 1;
   } else if (child->flags & TASK_NEWPID) {
-    /* The TASK_NEWPID could have been set for root task */
-    pids[0] = 0;
+    /*
+     * This happens for a restart with --pidns in which
+     * the root task is init in its namespace (the flag
+     * TASK_NEWPID was set for this root task).
+     */
+    assert(pids->depth == 0);
     flags |= CLONE_NEWPID;
-  }
-  if (flags & CLONE_NEWPID && !ctx->args->pidns) {
-    ckpt_err("need --pidns for nested pidns container");
-    errno = -EINVAL;
-    return -1;
+    clone_args.nr_pids--;
+    numbers = pids->numbers + 1;
   }
 #endif /* CLONE_NEWPID */
  }
@@ -1966,18 +2324,30 @@ static pid_t ckpt_fork_child(struct ckpt_ctx *ctx, struct task *child)
  clone_args.child_stack_size = genstack_size(stk);

  ckpt_dbg("task %d forking with flags %lx numpids %d\n",
-  child->pid, flags, clone_args.nr_pids);
- for (i = 0; i < clone_args.nr_pids; i++)
-  ckpt_dbg("task %d pid[%d]=%d\n", child->pid, i, pids[i]);
+    pid_at_index(ctx, child->pid_ind), flags, clone_args.nr_pids);
+ ckpt_dbg("task %d pids:", pid_at_index(ctx, child->pid_ind));
+ for (j = 0; j < clone_args.nr_pids; j++)
+  ckpt_dbg_cont(" %d\n", numbers[j]);
+ ckpt_dbg("...\n");

- pid = eclone(ckpt_fork_stub, child, flags, &clone_args, pids);
- if (pid < 0)
+ pid = eclone(ckpt_fork_stub, child, flags, &clone_args, numbers);
```

```
+ if (pid < 0) {
   ckpt_perror("eclone");
+  ctx_set_errno(ctx);
+ }

  if (pid < 0 || !(child->flags & TASK_THREAD))
   genstack_release(stk);

- ckpt_dbg("forked child vpid %d (asked %d)\n", pid, child->pid);
+ ckpt_dbg("forked child vpid %d (asked %d)\n",
+    pid, pid_at_index(ctx, child->pid_ind));
+
+ if (ctx->args->pids && pids->numbers[0] != 1 &&
+     pid != pids->numbers[ppids->depth]) {
+  ckpt_err("failed to create specific pid with eclone\n");
+  return ctx_ret_errno(ctx, EAGAIN);
+ }
+
  return pid;
 }

@@ -2075,7 +2445,6 @@ static void ckpt_read_write_inspect(struct ckpt_ctx *ctx)

  while (1) {
   ret = _ckpt_read(STDIN_FILENO, &h, sizeof(h));
-ckpt_dbg("ret %d len %d type %d\n", ret, h.len, h.type);
   if (ret == 0)
    break;
   if (ret < 0)
@@ -2111,7 +2480,6 @@ ckpt_dbg("ret %d len %d type %d\n", ret, h.len, h.type);

    h.len -= ret;
    ret = ckpt_write(STDOUT_FILENO, ctx->buf, ret);
-ckpt_dbg("write len %d (%d)\n", len, ret);
    if (ret < 0)
     ckpt_abort(ctx, "write output");
   }
@@ -2165,12 +2533,12 @@ static int ckpt_do_feeder(struct ckpt_ctx *ctx)
  if (ckpt_write_container(ctx) < 0)
   ckpt_abort(ctx, "write container section");

+ if (ckpt_write_pids(ctx) < 0)
+  ckpt_abort(ctx, "write c/r pids");
+
  if (ckpt_write_tree(ctx) < 0)
   ckpt_abort(ctx, "write c/r tree");

- if (ckpt_write_vpids(ctx) < 0)
```

```
-  ckpt_abort(ctx, "write vpids");
-
   /* read rest -> write rest */
   if (ctx->args->inspect)
    ckpt_read_write_inspect(ctx);
@@ -2196,8 +2564,9 @@ static int ckpt_do_feeder(struct ckpt_ctx *ctx)
   */
  static int ckpt_adjust_pids(struct ckpt_ctx *ctx)
  {
+ struct ckpt_pids *pids, *copy;
   struct pid_swap swap;
- int n, m, len, ret;
+ int n, m, off, len, ret;
   pid_t coord_sid;

   coord_sid = getsid(0);
@@ -2212,23 +2581,22 @@ static int ckpt_adjust_pids(struct ckpt_ctx *ctx)
    *    but correct should be: [][][B][][A][]...
    */

- len = sizeof(struct ckpt_pids) * ctx->pids_nr;
+ len = ctx->pids_nr * sizeof(*pids) + ctx->vpids_nr * sizeof(__s32);

  #ifdef CHECKPOINT_DEBUG
   ckpt_dbg("====== PIDS ARRAY\n");
   for (m = 0; m < ctx->pids_nr; m++) {
-  struct ckpt_pids *p;
-  p = &ctx->pids_arr[m];
-  ckpt_dbg("[%d] pid %d ppid %d sid %d pgid %d\n",
-    m, p->vpid, p->vppid, p->vsid, p->vpgid);
+  pids = pids_of_index(ctx, m + 1);
+  ckpt_dbg("[%d] pid %d depth %d\n",
+    m, pids->numbers[0], pids->depth);
   }
   ckpt_dbg(".....\n");
  #endif

- memcpy(ctx->copy_arr, ctx->pids_arr, len);
+ memcpy(ctx->pids_copy, ctx->pids_arr, len);

- /* read in 'pid_swap' data and adjust ctx->pids_arr */
- for (n = 0; n < ctx->tasks_nr; n++) {
+ /* read in 'pid_swap' data and adjust ctx->pids */
+ for (n = 0; n < ctx->tasks_cnt; n++) {
    /* get pid info from next task */
    ret = read(ctx->pipe_in, &swap, sizeof(swap));
    if (ret < 0)
@@ -2240,33 +2608,30 @@ static int ckpt_adjust_pids(struct ckpt_ctx *ctx)
```

```
   ckpt_dbg("c/r swap old %d new %d\n", swap.old, swap.new);
   for (m = 0; m < ctx->pids_nr; m++) {
-   if (ctx->pids_arr[m].vpid == swap.old)
-    ctx->copy_arr[m].vpid = swap.new;
-   if (ctx->pids_arr[m].vtgid == swap.old)
-    ctx->copy_arr[m].vtgid = swap.new;
-   if (ctx->pids_arr[m].vsid == swap.old)
-    ctx->copy_arr[m].vsid = swap.new;
-   if (ctx->pids_arr[m].vpgid == swap.old)
-    ctx->copy_arr[m].vpgid = swap.new;
+   off = ctx->pids_index[m + 1];
+   pids = pids_at_offset(ctx, off);
+   copy = pids_copy_at_offset(ctx, off);
+   if (pids->numbers[0] == swap.old)
+    copy->numbers[0] = swap.new;
   }
  }

- memcpy(ctx->pids_arr, ctx->copy_arr, len);
+ free(ctx->pids_arr);
+ ctx->pids_arr = ctx->pids_copy;
+ ctx->pids_copy = NULL;

 #ifdef CHECKPOINT_DEBUG
  if (!ctx->args->pids) {
   ckpt_dbg("====== PIDS ARRAY (swaped)\n");
   for (m = 0; m < ctx->pids_nr; m++) {
-   struct ckpt_pids *p;
-   p = &ctx->pids_arr[m];
-   ckpt_dbg("[%d] pid %d ppid %d sid %d pgid %d\n",
-     m, p->vpid, p->vppid, p->vsid, p->vpgid);
+   pids = pids_of_index(ctx, m + 1);
+   ckpt_dbg("[%d] pid %d depth %d\n",
+     m, pids->numbers[0], pids->depth);
   }
   ckpt_dbg(".....\n");
  }
 #endif

- close(ctx->pipe_in);  /* called by feeder, no need to mark */
  return 0;
 }

@@ -2477,35 +2842,46 @@ static int ckpt_read_container(struct ckpt_ctx *ctx)
  return ckpt_read_obj_type(ctx, ptr, 200, CKPT_HDR_LSM_INFO);
 }
```

```
-static int ckpt_read_tree(struct ckpt_ctx *ctx)
+static int ckpt_read_pids(struct ckpt_ctx *ctx)
 {
- struct ckpt_hdr_tree *h;
+ struct ckpt_hdr_pids *h;
  int len, ret;

- h = (struct ckpt_hdr_tree *) ctx->tree;
- ret = ckpt_read_obj_type(ctx, h, sizeof(*h), CKPT_HDR_TREE);
+ h = (struct ckpt_hdr_pids *) ctx->pids;
+ ret = ckpt_read_obj_type(ctx, h, sizeof(*h), CKPT_HDR_PIDS);
  if (ret < 0)
   return ret;

- ckpt_dbg("number of tasks: %d\n", h->nr_tasks);
+ ckpt_dbg("number of pids %d, vpids %d\n", h->nr_pids, h->nr_vpids);

- if (h->nr_tasks <= 0) {
-  ckpt_err("invalid number of tasks %d", h->nr_tasks);
-  errno = EINVAL;
-  return -1;
+ if (h->nr_pids <= 0) {
+  ckpt_err("invalid number of pids %d", h->nr_pids);
+  return ctx_ret_errno(ctx, EINVAL);
+ }
+ if (h->nr_vpids < 0) {
+  ckpt_err("invalid number of vpids %d", h->nr_vpids);
+  return ctx_ret_errno(ctx, EINVAL);
+ }
+ if (h->offset < 0) {
+  ckpt_err("invalid pid offset %d", h->offset);
+  return ctx_ret_errno(ctx, EINVAL);
  }

- /* get a working a copy of header */
- memcpy(ctx->buf, ctx->tree, BUFSIZE);
-
- ctx->pids_nr = h->nr_tasks;
+ ctx->pids_nr = h->nr_pids;
+ ctx->vpids_nr = h->nr_vpids;
+ ctx->pid_offset = h->offset;

- len = sizeof(struct ckpt_pids) * ctx->pids_nr;
+       len = ctx->pids_nr * sizeof(struct ckpt_pids) +
+  ctx->vpids_nr * sizeof(__s32);

  ctx->pids_arr = malloc(len);
- ctx->copy_arr = malloc(len);
```

```
- if (!ctx->pids_arr || !ctx->copy_arr)
-  return -1;
+ ctx->pids_copy = malloc(len);
+ ctx->pids_new = malloc(len);
+ if (!ctx->pids_arr || !ctx->pids_copy || !ctx->pids_new)
+  return ctx_ret_errno(ctx, EINVAL);
+
+ ctx->pids_off = 0;
+ ctx->pids_len = len;

  ret = ckpt_read_obj_ptr(ctx, ctx->pids_arr, len, CKPT_HDR_BUFFER);
  if (ret < 0)
@@ -2514,95 +2890,36 @@ static int ckpt_read_tree(struct ckpt_ctx *ctx)
  return ret;
 }

-/*
- * transform vpids arrays to the format convenient for eclone:
- * prefix the level 0 pid to every sequence of nested pids.
- * also,  set the vpids pointers in all the tasks.
- */
-static int assign_vpids(struct ckpt_ctx *ctx)
+static int ckpt_read_tree(struct ckpt_ctx *ctx)
 {
- __s32 *vpids_arr;
- int depth, hidx, vidx, tidx;
- struct task *task;
-
- vpids_arr = malloc(sizeof(__s32) * (ctx->vpids_nr + ctx->pids_nr));
- if (vpids_arr == NULL) {
-  perror("assign_vpids malloc");
-  return -1;
- }
-
- for (tidx = 0, hidx = 0, vidx = 0; tidx < ctx->pids_nr; tidx++) {
-  task = &ctx->tasks[tidx];
-  depth = ctx->pids_arr[tidx].depth;
-
-  task->vidx = vidx;
-  task->piddepth = depth;
+ struct ckpt_hdr_tree *h;
+ int len, ret;

-  /* set task's and top level pid */
-  vpids_arr[vidx++] = task->pid;
-  /* copy task's nested pids */
-  memcpy(&vpids_arr[vidx], &ctx->vpids_arr[hidx],
-         sizeof(__s32) * depth);
```

```
+ h = (struct ckpt_hdr_tree *) ctx->tree;
+ ret = ckpt_read_obj_type(ctx, h, sizeof(*h), CKPT_HDR_TREE);
+ if (ret < 0)
+  return ret;

-  vidx += depth;
-  hidx += depth;
+ ckpt_dbg("number of tasks: %d\n", h->nr_tasks);

-#ifdef CHECKPOINT_DEBUG
-  ckpt_dbg("task[%d].vidx = %d (depth %d, rpid %d)\n",
-   tidx, vidx, depth, ctx->pids_arr[tidx].vpid);
-  while (depth-- > 0)  {
-   ckpt_dbg("task[%d].vpid[%d] = %d\n", tidx,
-     depth, vpids_arr[hidx - depth - 1]);
-  }
-#endif
+ if (h->nr_tasks <= 0) {
+  ckpt_err("invalid number of tasks %d", h->nr_tasks);
+  return ctx_ret_errno(ctx, EINVAL);
  }

- /* relpace "raw" vpids_arr with this one */
- free(ctx->vpids_arr);
- ctx->vpids_arr = vpids_arr;
+ ctx->tasks_nr = h->nr_tasks;

- return 0;
-}
+ len = sizeof(struct ckpt_task_pids) * ctx->tasks_nr;

-static int ckpt_read_vpids(struct ckpt_ctx *ctx)
-{
- int i, len, ret;
-
- for (i = 0; i < ctx->pids_nr; i++) {
-  if (ctx->pids_arr[i].depth < 0) {
-   ckpt_err("Invalid depth %d for pid %d",
-     ctx->pids_arr[i].depth,
-     ctx->tasks[i].pid);
-   errno = -EINVAL;
-   return -1;
-  }
-
-  ctx->vpids_nr += ctx->pids_arr[i].depth;
-
-  if(ctx->vpids_nr < 0) {
-   ckpt_err("Number of vpids overflowed");
```

```
-   errno = -E2BIG;
-   return -1;
-  }
- }
-
- ckpt_dbg("number of vpids: %d\n", ctx->vpids_nr);
-
- if (!ctx->vpids_nr)
-  return 0;
+ ctx->tasks_arr = malloc(len);
+ if (!ctx->tasks_arr)
+  return ctx_ret_errno(ctx, EINVAL);

- len = sizeof(__s32) * ctx->vpids_nr;
- if (len < 0) {
-  ckpt_err("Length of vpids array overflowed");
-  errno = -EINVAL;
-  return -1;
- }
+ ret = ckpt_read_obj_ptr(ctx, ctx->tasks_arr, len, CKPT_HDR_BUFFER);

- ctx->vpids_arr = malloc(len);
- if (!ctx->pids_arr)
-  return -1;
+ if (ret < 0)
+  return ret;

- ret = ckpt_read_obj_ptr(ctx, ctx->vpids_arr, len, CKPT_HDR_BUFFER);
  return ret;
 }

@@ -2660,32 +2977,43 @@ static int ckpt_write_container(struct ckpt_ctx *ctx)
  return ckpt_write_obj(ctx, (struct ckpt_hdr *) ptr);
 }

-static int ckpt_write_tree(struct ckpt_ctx *ctx)
+static int ckpt_write_pids(struct ckpt_ctx *ctx)
 {
- struct ckpt_hdr_tree *h;
+ struct ckpt_hdr_pids *h;
  int len;

- h = (struct ckpt_hdr_tree *) ctx->tree;
+ h = (struct ckpt_hdr_pids *) ctx->pids;
  if (ckpt_write_obj(ctx, (struct ckpt_hdr *) h) < 0)
-  ckpt_abort(ctx, "write tree");
+  ckpt_abort(ctx, "write pids");
```

```
- len = sizeof(struct ckpt_pids) * ctx->pids_nr;
+        len = ctx->pids_nr * sizeof(struct ckpt_pids) +
+  ctx->vpids_nr * sizeof(__s32);
  if (ckpt_write_obj_ptr(ctx, ctx->pids_arr, len, CKPT_HDR_BUFFER) < 0)
   ckpt_abort(ctx, "write pids");

  return 0;
 }

-static int ckpt_write_vpids(struct ckpt_ctx *ctx)
+static int ckpt_write_tree(struct ckpt_ctx *ctx)
 {
+ struct ckpt_hdr_tree *h;
  int len;

- if (!ctx->vpids_nr)
-  return 0;
- len = sizeof(__s32) * ctx->vpids_nr;
- if (ckpt_write_obj_ptr(ctx, ctx->vpids_arr, len, CKPT_HDR_BUFFER) < 0)
-  ckpt_abort(ctx, "write vpids");
- ckpt_dbg("wrote %d bytes for %d vpids\n", len, ctx->vpids_nr);
+ h = (struct ckpt_hdr_tree *) ctx->tree;
+ if (ckpt_write_obj(ctx, (struct ckpt_hdr *) h) < 0)
+  ckpt_abort(ctx, "write tree");
+
+ len = sizeof(struct ckpt_task_pids) * ctx->tasks_nr;
+ckpt_dbg("len = %d\n");
+ if (ckpt_write_obj_ptr(ctx, ctx->tasks_arr, len, CKPT_HDR_BUFFER) < 0)
+  ckpt_abort(ctx, "write pids");
+
+ for (len = 0; len < ctx->tasks_nr; len++) {
+  struct ckpt_task_pids *task;
+  task = &ctx->tasks_arr[len];
+  ckpt_dbg("\t[%d] pid %d tgid %d pgid %d sid %d\n", len,
+    task->vpid, task->vtgid, task->vpgid, task->vsid);
+ }

  return 0;
 }
@@ -2697,31 +3025,65 @@ static int ckpt_write_vpids(struct ckpt_ctx *ctx)
 #define HASH_BITS 11
 #define HASH_BUCKETS (2 << (HASH_BITS - 1))

-static int hash_init(struct ckpt_ctx *ctx)
+static int hash_expand(struct ckpt_ctx *ctx, int depth)
 {
- struct hashent **hash;
+ struct hashent ***hash;
```

```
+ int *hash_last_pid;

- ctx->hash_arr = malloc(sizeof(*hash) * HASH_BUCKETS);
- if (!ctx->hash_arr) {
-  ckpt_perror("malloc hash table");
-  return -1;
+ hash = ctx->hash_arr;
+ hash = realloc(hash, sizeof(*hash) * depth);
+ if (!hash) {
+  ckpt_perror("allocate hash table");
+  return ctx_set_errno(ctx);
+ } else
+  ctx->hash_arr = hash;
+
+ hash_last_pid = ctx->hash_last_pid;
+ hash_last_pid = realloc(hash_last_pid, sizeof(*hash_last_pid) * depth);
+ if (!hash_last_pid) {
+  ckpt_perror("allocate hash table");
+  return ctx_set_errno(ctx);
+ } else
+  ctx->hash_last_pid = hash_last_pid;
+
+ while (ctx->hash_depth < depth) {
+  hash[ctx->hash_depth] = malloc(sizeof(**hash) * HASH_BUCKETS);
+  if (!hash[ctx->hash_depth]) {
+   ckpt_perror("allocate hash table");
+   return ctx_set_errno(ctx);
+  }
+  memset(hash[ctx->hash_depth], 0, sizeof(**hash) * HASH_BUCKETS);
+  hash_last_pid[ctx->hash_depth] = CKPT_RESERVED_PIDS;
+  ctx->hash_depth++;
  }
- memset(ctx->hash_arr, 0, sizeof(*hash) * HASH_BUCKETS);
+
  return 0;
 }

+static int hash_init(struct ckpt_ctx *ctx)
+{
+ return hash_expand(ctx, 1);
+}
+
 static void hash_exit(struct ckpt_ctx *ctx)
 {
  struct hashent *hash, *next;
- int i;
+ int i, j;
+
```

```
+ if (!ctx->hash_arr)
+  return;

- for (i = 0; i < HASH_BUCKETS; i++) {
-  for (hash = ctx->hash_arr[i]; hash; hash = next) {
-   next = hash->next;
-   free(hash);
+ for (i = 0; i < ctx->hash_depth; i++) {
+  for (j = 0; j < HASH_BUCKETS; j++) {
+   for (hash = ctx->hash_arr[i][i]; hash; hash = next) {
+    next = hash->next;
+    free(hash);
+   }
   }
+  free(ctx->hash_arr[i]);
  }

+ free(ctx->hash_last_pid);
  free(ctx->hash_arr);
 }

@@ -2736,35 +3098,55 @@ static inline int hash_func(long key)
  return (hash >> (sizeof(key)*8 - HASH_BITS));
 }

-static int hash_insert(struct ckpt_ctx *ctx, long key, void *data)
+static int hash_insert(struct ckpt_ctx *ctx, long key, void *data, int level)
 {
  struct hashent *hash;
  int bucket;

+ if (level >= ctx->hash_depth)
+  if (hash_expand(ctx, level) < 0)
+   return ctx_set_errno(ctx);
+
  hash = malloc(sizeof(*hash));
  if (!hash) {
   ckpt_perror("malloc hash");
-  return -1;
+  return ctx_set_errno(ctx);
  }
  hash->key = key;
  hash->data = data;

  bucket = hash_func(key);
- hash->next = ctx->hash_arr[bucket];
- ctx->hash_arr[bucket] = hash;
+ hash->next = ctx->hash_arr[level][bucket];
```

```
+ ctx->hash_arr[level][bucket] = hash;

  return 0;
 }

-static void *hash_lookup(struct ckpt_ctx *ctx, long key)
+static void *hash_lookup_level(struct ckpt_ctx *ctx, long key, int level)
 {
  struct hashent *hash;
  int bucket;

+ if (level > ctx->hash_depth)
+  return NULL;
+
  bucket = hash_func(key);
- for (hash = ctx->hash_arr[bucket]; hash; hash = hash->next) {
+ for (hash = ctx->hash_arr[level][bucket]; hash; hash = hash->next) {
   if (hash->key == key)
    return hash->data;
  }
  return NULL;
 }
+
+static void *hash_lookup(struct ckpt_ctx *ctx, long key)
+{
+ return hash_lookup_level(ctx, key, 0);
+}
+
+static void *hash_lookup_ind(struct ckpt_ctx *ctx, int n)
+{
+ if (n == 0 || n == CKPT_PID_ROOT)
+  return hash_lookup_level(ctx, pid_at_index(ctx, 1), 0);
+ else
+  return hash_lookup_level(ctx, pid_at_index(ctx, n), 0);
+}
--
1.7.1
```

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

---

Subject: [PATCH 11/11] restart: account for all ghost sids before ghost pgids
Posted by Oren Laadan on Mon, 07 Feb 2011 17:21:32 GMT
View Forum Message <> Reply to Message

This patch fixes the following two bugs:

1) Consider a subtree checkpoint of one task with pid != sid and also pgid == sid. At restart, ckpt_init_tree() will add a ghost task that accounts for the pgid, and ckpt_setup_task() will set its ppid and sid to be the root task. But because the root task is _not_ a session leader, ckpt_set_creator() will incorrecly add a dead task between the root and the ghost.

We fix this by ensuring that in this case we set the sid of the ghost task to be _the same_ as that of the parent task (and not the pid of the parent task itself - which only works when the parent is indeed the session leader).

2) Consider a checkpoint with 2 tasks: one has a pgid corresponding to a dead process, the other has its sid corresponding to the same dead process. If the former is processed by ckpt_setup_task() first, it will produce a ghost task that is _not_ a session leader - which is incorrect. Moreover, this will not be recrtified later when the latter task is found.

We fix this by splitting the work into two loops: in the first we only consider tasks' sids and add the corresponding ghosts. This ensures that they are session leaders. In the second we consider the dead pgids and the tgids.

The patch also refactors ckpt_set_task() into smaller pieces to simplify its logic and improve readability.

Signed-off-by: Oren Laadan <orenl@cs.columbia.edu>

```
---
 restart.c |   83  +++++++++++++++++++++++++++++++++++++++++++++++---------------
 1 files changed, 63 insertions(+), 20 deletions(-)

diff --git a/restart.c b/restart.c
index f64e508..e60335c 100644
--- a/restart.c
+++ b/restart.c
@@ -1456,22 +1456,12 @@ static int ckpt_build_tree(struct ckpt_ctx *ctx)
  return 0;
 }

-static int ckpt_setup_task(struct ckpt_ctx *ctx, int pid_ind, int ppid_ind)
+static int ckpt_ghost_task(struct ckpt_ctx *ctx, struct ckpt_pids *pids,
+      int pid_ind, int sid_ind, int ppid_ind)
 {
  struct task *task;
- struct ckpt_pids *pids;
```

```
  int j;

- /* ignore if outside namespace */
- if (pid_ind == 0 || pid_ind == CKPT_PID_ROOT)
-  return 0;
-
- pids = pids_of_index(ctx, pid_ind);
-
- /* skip if already handled */
- if (hash_lookup(ctx, pids->numbers[0]))
-  return 0;
-
  task = &ctx->tasks[ctx->tasks_cnt++];

  task->flags = TASK_GHOST;
@@ -1479,7 +1469,7 @@ static int ckpt_setup_task(struct ckpt_ctx *ctx, int pid_ind, int ppid_ind)
  task->pid_ind = pid_ind;
  task->ppid_ind = ppid_ind;
  task->tgid_ind = pid_ind;
- task->sid_ind = ppid_ind;
+ task->sid_ind = sid_ind;

  task->children = NULL;
  task->next_sib = NULL;
@@ -1500,6 +1490,47 @@ static int ckpt_setup_task(struct ckpt_ctx *ctx, int pid_ind, int
ppid_ind)
  return 0;
 }

+static struct ckpt_pids *ckpt_consider_pid(struct ckpt_ctx *ctx, int pid_ind)
+{
+ struct ckpt_pids *pids;
+
+ /* ignore if outside namespace */
+ if (pid_ind == 0 || pid_ind == CKPT_PID_ROOT)
+  return NULL;
+
+ pids = pids_of_index(ctx, pid_ind);
+
+ /* skip if already handled */
+ if (hash_lookup(ctx, pids->numbers[0]))
+  return NULL;
+
+ return pids;
+}
+
+static int ckpt_set_session(struct ckpt_ctx *ctx, int pid_ind, int ppid_ind)
+{
```

```
+ struct ckpt_pids *pids;
+
+ pids = ckpt_consider_pid(ctx, pid_ind);
+ if (!pids)
+  return 0;
+
+ return ckpt_ghost_task(ctx, pids, pid_ind, pid_ind, ppid_ind);
+}
+
+static int ckpt_set_task(struct ckpt_ctx *ctx, int pid_ind, int ppid_ind)
+{
+ struct ckpt_pids *pids;
+ struct task *parent;
+
+ pids = ckpt_consider_pid(ctx, pid_ind);
+ if (!pids)
+  return 0;
+
+ parent = hash_lookup_ind(ctx, ppid_ind);
+ return ckpt_ghost_task(ctx, pids, pid_ind, parent->sid_ind, ppid_ind);
+}
+
 static int _ckpt_valid_pid(struct ckpt_ctx *ctx, pid_t pid, char *which, int i)
 {
  if (pid < 0) {
@@ -1697,17 +1728,29 @@ static int ckpt_init_tree(struct ckpt_ctx *ctx)

  ctx->tasks_cnt = ctx->tasks_nr;

- /* add pids unaccounted for (no tasks) */
+ /*
+  * Add sids unaccounted for (no tasks): find those orphan pids
+  * that are used as some task's sid, and create ghost session
+  * leaders for them. Since they should be session leaders, we
+  * must do this before adding other ghost tasks for tgid/pgid
+  * below: the latter is added a non-session leader.
+  */
  for (i = 0; i < ctx->tasks_nr; i++) {
-
   /*
    * An unaccounted-for sid belongs to a task that was a
-   * session leader and died. We can safe set its parent
+   * session leader and died. We can safely set its parent
    * (and creator) to be the root task.
    */
-  if (ckpt_setup_task(ctx, tasks_arr[i].vsid, root_pid_ind) < 0)
+
+  if (ckpt_set_session(ctx, tasks_arr[i].vsid, root_pid_ind) < 0)
```

```
    return -1;
+ }

+ /*
+  * Add tgids/pgids unaccounted for (no tasks): find those
+  * orphan tgids/pgids and create ghost tasks for them.
+  */
+ for (i = 0; i < ctx->tasks_nr; i++) {
   /*
    * An sid == 0 means that the session was inherited an
    * ancestor of root_task, and more specifically, via
@@ -1723,18 +1766,18 @@ static int ckpt_init_tree(struct ckpt_ctx *ctx)
    * need to add it with the same sid as current (and
    * other) threads.
    */
-  if (ckpt_setup_task(ctx, tasks_arr[i].vtgid, ppid_ind) < 0)
+  if (ckpt_set_task(ctx, tasks_arr[i].vtgid, ppid_ind) < 0)
    return -1;

   /*
    * If pgrp == sid, then the pgrp/sid will already have
    * been hashed by now (e.g. by the call above) and the
-   * ckpt_setup_task() will return promptly.
+   * ckpt_set_task() will return promptly.
    * If pgrp != sid, then the pgrp 'owner' must have the
    * same sid as us: all tasks with same pgrp must have
    * their sid matching.
    */
-  if (ckpt_setup_task(ctx, tasks_arr[i].vpgid, ppid_ind) < 0)
+  if (ckpt_set_task(ctx, tasks_arr[i].vpgid, ppid_ind) < 0)
    return -1;
  }

--
1.7.1
```

_____

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