Subject: [PATCH 5/6] c/r: checkpoint and restart pids objects
Posted by Oren Laadan on Mon, 07 Feb 2011 17:18:07 GMT
View Forum Message <> Reply to Message

Make use of (shared) pids objects instead of simply saving the pid_t numbers in both checkpoint and restart.

The motivation for this change is twofold. First, since pid-ns came to life pid's in the kenrel _are_ shared objects and should be treated as such. This is useful e.g. for tty handling and also file-ownership (the latter waiting for this feature). Second, to properly support nested namesapces we need to report with each pid the entire list of pid numbers, not only a single pid. While current we do that for all "live" pids (those that belong to live tasks), we didn't do it for "dead" pids (to be assigned to ghost restarting tasks).

Note, that ideally the list of vpids of a pid object should also include the pid-ns to which each level belongs; however, in this patch we don't yet hanlde that. So only linear pid-nesting works well and not arbitrary tree.

DICLAIMER: this patch is big and intrusive! Here is a summary of the changes that it makes:

CHECKPOINT:

1) Modified the data structures used to describe pids and tasks' pids:
 struct ckpt_pids - for the data of a pids object (depth, numbers)
   (pids objects are collected in the order found, and are assigned
   tags sequentially, starting from 1)
 struct ckpt_task_pids - for a task's pids, holding the _tag_ of the
   corresponding pids object rather than their pid numbers themselves.

2) Accordingly, two arrays are used to hold this information:
 ctx->pids_arr - array of 'struct ckpt_pids' collected from the tasks
   and the pids they reference. Entries are of variable size depending
   on the pid-ns nesting level.
 ctx->tasks_arr - array of 'struct ckpt_task_pids' collected from teh
   tasks. Entries are of fixed size, and hold the objref tags to the
 shared pids objects rather than actual pid numbers.
 (the old vpids_arr is no longer needed, nor written separately).

3) We now first write the pids information, then tasks' pids.

4) checkpoint_pids() builds and writes the ctx->pids_arr:
 checkpoint_pids_build() - iterates over the tasks and collects the
   unique pids in a flex_array (also inserts them into the objhash)
 checkpoint_pids_dumps() - dumps the data from the flex_array in

the format of ctx->tasks_arr

5) checkpoint_tree() dumps the tasks' pids information, by scanning
all the tasks and writing out tags of the pids they reference. If
a pgid/sid is zero, i.e. from an ancestor pid-ns, then the tag will
be zero.

6) In container checkpoint, pids out of our namesapce are disallwed.
We don't do leak detection on pids objects (should we ?).

RESTART:

1) We first call prepare_descendants() to set the ->checkpoint_ctx
of the restarting tasks, and _then_ read the pids data followed by
the tasks' pids data. We validate both against existing tasks.

2) restore_read_pids() reads the pids data, validates that each pid
exists (*) and adds the pids to the objhash. Verify that the owner
task is within our restart context.

(*) We validate them from the root task's point of view, by seeing
that the task has the correct 'struct pid' pointer. NOTE: user-cr
does not support restart --no-pids when there are nested pis-ns,
because is it quite complicated to find ou the pids of all tasks
at all nested levels from userspace.

3) restore_read_tasks() reads the tasks' pids data, validates each
task and adds it to the ctx->tasks_arr. Verify that the task is
within our restart context.

4) We track the array of restarting _tasks_ and the active _task_
instead an array of restarting pids and the active pid. The helpers
to wake-up, sync, check active task etc were modified accordingly.
It improves and simplifies the logic, e.g. restore_activate_next().

5) There are two special values for pgid/sid tags:
0 - means that it is from an ancestor namespace, so we verify that
this is the case. For sid, user-cr should have created the task
properly; for pgid, use the coordinator's (or coordinator's parent)
pid if from differnet namespace, or fail.
CKPT_PID_ROOT - means that we want to reuse the root task's sid,
useful for when the root task is _not_ a conatiner init (e.g. in
subtree c/r) and its session (like our pgrp) was inherited from
somewhere above).

6) Restoring of a task's pgid was moved to when task is validated,
as this should be part of the validation.

NOTE: the patch does not yet allow non-linear nesting of pid-ns.
This would require to make pid-ns a shared object and track it by
the 'struct ckpt_pids' on the kernel side, and in userspace we'll
need to update the logic of MakeForest algorithm to be pid-ns aware
(probably similarly to how sid constraints are handled).

Signed-off-by: Oren Laadan <orenl@cs.columbia.edu>
---
 include/linux/checkpoint_hdr.h | 23 ++-
 include/linux/checkpoint_types.h | 10 +-
 kernel/checkpoint/checkpoint.c | 450 ++++++++++++++++++---------
 kernel/checkpoint/process.c | 108 +-------
 kernel/checkpoint/restart.c | 568 +++++++++++++++++++++++++++++++-------
 kernel/checkpoint/sys.c | 5 -
 kernel/signal.c | 8 +-
 7 files changed, 806 insertions(+), 366 deletions(-)

diff --git a/include/linux/checkpoint_hdr.h b/include/linux/checkpoint_hdr.h
index 922eff0..6f991c6 100644
--- a/include/linux/checkpoint_hdr.h
+++ b/include/linux/checkpoint_hdr.h
@@ -107,7 +107,9 @@ 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
@@ -358,20 +360,33 @@ 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 */
diff --git a/include/linux/checkpoint_types.h b/include/linux/checkpoint_types.h
index 87a569a..60c664f 100644
--- a/include/linux/checkpoint_types.h
+++ b/include/linux/checkpoint_types.h
@@ -68,16 +68,14 @@ struct ckpt_ctx {

  int nr_vpids;   /* total count of vpids */

- /* [checkpoint] */
- struct task_struct *tsk; /* current target task */
  struct task_struct **tasks_arr; /* array of all tasks */
  int nr_tasks;   /* size of tasks array */

+ /* [checkpoint] */
+ struct task_struct *tsk; /* current target task */
+
  /* [restart] */
- struct pid_namespace *coord_pidns;/* coordinator pid_ns */
- struct ckpt_pids *pids_arr; /* array of all pids [restart] */
- int nr_pids;   /* size of pids array */
- int active_pid;   /* (next) position in pids array */
+ int active_task;  /* (next) position in pids array */
  atomic_t nr_total;  /* total tasks count */
  struct completion complete; /* completion for container root */
  wait_queue_head_t waitq; /* waitqueue for restarting tasks */
diff --git a/kernel/checkpoint/checkpoint.c b/kernel/checkpoint/checkpoint.c
index a850423..342590b 100644
```

```
--- a/kernel/checkpoint/checkpoint.c
+++ b/kernel/checkpoint/checkpoint.c
@@ -26,6 +26,7 @@
 #include <linux/utsname.h>
 #include <linux/magic.h>
 #include <linux/hrtimer.h>
+#include <linux/flex_array.h>
 #include <linux/deferqueue.h>
 #include <linux/checkpoint.h>
 #include <linux/pid_namespace.h>
@@ -312,133 +313,6 @@ static int may_checkpoint_task(struct ckpt_ctx *ctx, struct task_struct
*t)
  return ret;
 }

-#define CKPT_HDR_PIDS_CHUNK 256
-
-/*
- * Write the pids in ctx->root_nsproxy->pidns.  This info is
- * needed at restart to unambiguously dereference tasks.
- */
-static int checkpoint_pids(struct ckpt_ctx *ctx)
-{
- struct ckpt_pids *h;
- struct pid_namespace *root_pidns;
- struct task_struct *task;
- struct task_struct **tasks_arr;
- int nr_tasks, n, pos = 0, ret = 0;
-
- root_pidns = ctx->root_nsproxy->pid_ns;
- tasks_arr = ctx->tasks_arr;
- nr_tasks = ctx->nr_tasks;
- BUG_ON(nr_tasks <= 0);
-
- ret = ckpt_write_obj_type(ctx, NULL,
-       sizeof(*h) * nr_tasks,
-       CKPT_HDR_BUFFER);
- if (ret < 0)
-  return ret;
-
- h = ckpt_hdr_get(ctx, sizeof(*h) * CKPT_HDR_PIDS_CHUNK);
- if (!h)
-  return -ENOMEM;
-
- do {
-  rcu_read_lock();
-  for (n = 0; n < min(nr_tasks, CKPT_HDR_PIDS_CHUNK); n++) {
-   struct pid_namespace *task_pidns;
```

```
-   task = tasks_arr[pos];
-
-   h[n].vpid = task_pid_nr_ns(task, root_pidns);
-   h[n].vtgid = task_tgid_nr_ns(task, root_pidns);
-   h[n].vpgid = task_pgrp_nr_ns(task, root_pidns);
-   h[n].vsid = task_session_nr_ns(task, root_pidns);
-   h[n].vppid = task_tgid_nr_ns(task->real_parent,
-     root_pidns);
-   task_pidns = task_active_pid_ns(task);
-   h[n].depth = task_pidns->level - root_pidns->level;
-
-   ckpt_debug("task[%d]: vpid %d vtgid %d parent %d\n",
-       pos, h[n].vpid, h[n].vtgid, h[n].vppid);
-   ctx->nr_vpids += h[n].depth;
-   pos++;
-  }
-  rcu_read_unlock();
-
-  n = min(nr_tasks, CKPT_HDR_PIDS_CHUNK);
-  ret = ckpt_kwrite(ctx, h, n * sizeof(*h));
-  if (ret < 0)
-   break;
-
-  nr_tasks -= n;
- } while (nr_tasks > 0);
-
- _ckpt_hdr_put(ctx, h, sizeof(*h) * CKPT_HDR_PIDS_CHUNK);
- return ret;
-}
-
-static int checkpoint_vpids(struct ckpt_ctx *ctx)
-{
- __s32 *h;  /* vpid array */
- struct pid_namespace *root_pidns, *task_pidns = NULL, *active_pidns;
- struct task_struct *task;
- int ret, nr_tasks = ctx->nr_tasks;
- int tidx = 0; /* index into task array */
- int hidx = 0; /* pids written into current __s32 chunk */
- int vidx = 0; /* vpid index for current task */
-
- root_pidns = ctx->root_nsproxy->pid_ns;
- nr_tasks = ctx->nr_tasks;
-
- ret = ckpt_write_obj_type(ctx, NULL,
-       sizeof(*h) * ctx->nr_vpids,
-       CKPT_HDR_BUFFER);
- if (ret < 0)
-  return ret;
```

```
-
- h = ckpt_hdr_get(ctx, sizeof(*h) * CKPT_HDR_PIDS_CHUNK);
- if (!h)
-  return -ENOMEM;
-
- do {
-  rcu_read_lock();
-  while (tidx < nr_tasks && hidx < CKPT_HDR_PIDS_CHUNK) {
-   int nsdelta;
-
-   task = ctx->tasks_arr[tidx];
-   active_pidns = task_active_pid_ns(task);
-   nsdelta = active_pidns->level - root_pidns->level;
-   if (hidx + nsdelta - vidx > CKPT_HDR_PIDS_CHUNK)
-    /*
-     * We will release rcu before recording the
-     * remaining vpids, but neither task nor its
-     * pid can disappear.
-     */
-    nsdelta = CKPT_HDR_PIDS_CHUNK - hidx + vidx;
-
-   if (vidx == 0)
-    task_pidns = active_pidns;
-   while (vidx++ < nsdelta) {
-    h[hidx++] = task_pid_nr_ns(task, task_pidns);
-    task_pidns = task_pidns->parent;
-   }
-
-   if (task_pidns == root_pidns) {
-    tidx++;
-    vidx = 0;
-   }
-  }
-  rcu_read_unlock();
-
-  ret = ckpt_kwrite(ctx, h, hidx * sizeof(*h));
-  if (ret < 0)
-   break;
-
-  hidx = 0;
- } while (tidx < nr_tasks);
-
- _ckpt_hdr_put(ctx, h, sizeof(*h) * CKPT_HDR_PIDS_CHUNK);
- return ret;
-}
-
 static int collect_objects(struct ckpt_ctx *ctx)
 {
```

```
  int n, ret = 0;
@@ -545,31 +419,332 @@ static int build_tree(struct ckpt_ctx *ctx)
  return 0;
 }

-/* dump the array that describes the tasks tree */
-static int checkpoint_tree(struct ckpt_ctx *ctx)
+static int checkpoint_pids_build(struct ckpt_ctx *ctx,
+     struct flex_array *pids_arr,
+     int *nr_pids, int *nr_vpids)
 {
- struct ckpt_hdr_tree *h;
+ struct pid_namespace *root_pidns;
+ struct pid *pid, *tgid, *pgrp, *session;
+ struct pid *root_session;
+ struct task_struct *task;
+ int i = 0, vpids = 0;
+ int n, new, ret = 0;
+
+ /* safe because we reference the task */
+ root_session = get_pid(task_session(ctx->root_task));
+ root_pidns = ctx->root_nsproxy->pid_ns;
+
+ for (n = 0; n < ctx->nr_tasks; n++) {
+  task = ctx->tasks_arr[n];
+
+  rcu_read_lock();
+  pid = get_pid(task_pid(task));
+  tgid = get_pid(task_tgid(task));
+  pgrp = get_pid(task_pgrp(task));
+  session = get_pid(task_session(task));
+  rcu_read_unlock();
+
+  BUG_ON(!pid);
+  BUG_ON(!tgid);
+  BUG_ON(!pgrp);
+  BUG_ON(!session);
+
+  /*
+   * How to handle references to pids outside our pid-ns ?
+   * In container checkpoint, such pids are prohibited, so
+   * we report an error.
+   * In subtree checkpoint it is valid, however, we don't
+   * collect them here to not leak data (it is irrelevant
+   * to userspace anyway), Instead, in checkpoint_tree() we
+   * substitute 0 for the such pgrp/session entries.
+   */
+
```

```
+  /* pid */
+  ret = ckpt_obj_lookup_add(ctx, pid,
+        CKPT_OBJ_PID, &new);
+  if (ret >= 0 && new) {
+   vpids += pid->level - root_pidns->level;
+   ret = flex_array_put(pids_arr, i++, &pid, GFP_KERNEL);
+   new = 0;
+  }
+
+  /* tgid: if tgid != pid) */
+  if (ret >= 0 && tgid != pid)
+   ret = ckpt_obj_lookup_add(ctx, tgid,
+         CKPT_OBJ_PID, &new);
+  if (ret >= 0 && new) {
+   vpids += tgid->level - root_pidns->level;
+   ret = flex_array_put(pids_arr, i++, &tgid, GFP_KERNEL);
+   new = 0;
+  }
+
+  /*
+   * pgrp: if in our pid-namespace, and
+   *       if pgrp != tgid, and if pgrp != root_session
+   */
+  if (pid_nr_ns(pgrp, root_pidns) == 0) {
+   /* pgrp must be ours in container checkpoint */
+   if (!(ctx->uflags & CHECKPOINT_SUBTREE))
+    ret = -EBUSY;
+  } else if (ret >= 0 && pgrp != tgid && pgrp != root_session)
+   ret = ckpt_obj_lookup_add(ctx, pgrp,
+         CKPT_OBJ_PID, &new);
+  if (ret >= 0 && new) {
+   vpids += pgrp->level - root_pidns->level;
+   ret = flex_array_put(pids_arr, i++, &pgrp, GFP_KERNEL);
+   new = 0;
+  }
+
+  /*
+   * session: if in our pid-namespace, and
+   *          if session != tgid, and if session != root_session
+   */
+  if (pid_nr_ns(session, root_pidns) == 0) {
+   /* session must be ours in container checkpoint */
+   if (!(ctx->uflags & CHECKPOINT_SUBTREE))
+    ret = -EBUSY;
+  } else if (ret >= 0 && session != tgid && session != root_session)
+   ret = ckpt_obj_lookup_add(ctx, session,
+         CKPT_OBJ_PID, &new);
+  if (ret >= 0 && new) {
```

```
+   vpids += session->level - root_pidns->level;
+   ret = flex_array_put(pids_arr, i++, &session, GFP_KERNEL);
+  }
+
+  /* pids added to the objhash are already referenced there */
+  put_pid(pid);
+  put_pid(tgid);
+  put_pid(pgrp);
+  put_pid(session);
+
+  if (ret < 0)
+   break;
+ }
+
+ *nr_pids = i;
+ *nr_vpids = vpids;
+
+ ckpt_debug("nr_pids = %d, nr_vpids = %d\n", i, vpids);
+ return ret;
+}
+
+static int checkpoint_pids_dump(struct ckpt_ctx *ctx,
+    struct flex_array *pids_arr,
+    int nr_pids, int nr_vpids)
+{
+ struct ckpt_pids *h;
+ struct pid *pid;
+ char *buf;
+ int n = 0, vpids = 0;
+ int root_level, level;
+ int len, pos, ret;
+
+ pos = (nr_pids * sizeof(*h)) + (nr_vpids * sizeof(__s32));
+ ret = ckpt_write_obj_type(ctx, NULL, pos, CKPT_HDR_BUFFER);
+ if (ret < 0)
+  return ret;
+
+ buf = ckpt_hdr_get(ctx, PAGE_SIZE);
+ if (!buf)
+  return -ENOMEM;
+
+ root_level = ctx->root_nsproxy->pid_ns->level;
+
+ while (n < nr_pids) {
+  pos = 0;
+
+  rcu_read_lock();
+  while (n < nr_pids) {
```

```
+   pid = flex_array_get_ptr(pids_arr, n);
+   level = pid->level - root_level;
+   len = sizeof(*h) + level * sizeof(__s32);
+
+   /* need to flush current buffer ? */
+   if (pos + len > PAGE_SIZE)
+    break;
+
+   h = (struct ckpt_pids *) &buf[pos];
+   h->depth = level;
+   vpids += level;
+   pos += len;
+   n++;
+  }
+  rcu_read_unlock();
+
+  /* something must have changed since last count... */
+  if (vpids != nr_vpids) {
+   ret = -EBUSY;
+   break;
+  }
+
+  ret = ckpt_kwrite(ctx, buf, pos);
+  if (ret < 0)
+   break;
+ }
+
+ _ckpt_hdr_put(ctx, buf, PAGE_SIZE);
+ return ret;
+}
+
+/* dump the array of all pids encountered so far */
+static int checkpoint_pids(struct ckpt_ctx *ctx)
+{
+ struct flex_array *pids_arr;
+ struct ckpt_hdr_pids *h;
+ int nr_vpids = 0;
+ int nr_pids = 0;
+ int offset;
  int ret;

- h = ckpt_hdr_get_type(ctx, sizeof(*h), CKPT_HDR_TREE);
+ /*
+  * Note total items in the objhash already; will be used in
+  * checkpoint_tree() to compute the real index of pid objects
+  * in the order they were found. See checkpoint_tree().
+  */
+ offset = ckpt_obj_count(ctx) - 1;
```

```
+
+ pids_arr = flex_array_alloc(sizeof(struct pid),
+        4 * ctx->nr_tasks,
+        GFP_KERNEL);
+ if (!pids_arr)
+  return -ENOMEM;
+
+ ret = checkpoint_pids_build(ctx, pids_arr, &nr_pids, &nr_vpids);
+ if (ret < 0)
+  goto out;
+
+ h = ckpt_hdr_get_type(ctx, sizeof(*h), CKPT_HDR_PIDS);
  if (!h)
   return -ENOMEM;

- h->nr_tasks = ctx->nr_tasks;
+ h->nr_pids = nr_pids;
+ h->nr_vpids = nr_vpids;
+ h->offset = offset;

  ret = ckpt_write_obj(ctx, &h->h);
  ckpt_hdr_put(ctx, h);
  if (ret < 0)
+  goto out;
+
+ ret = checkpoint_pids_dump(ctx, pids_arr, nr_pids, nr_vpids);
+ out:
+ flex_array_free(pids_arr);
+ return ret;
+}
+
+int ckpt_lookup_pid(struct ckpt_ctx *ctx, struct pid *pid)
+{
+ int objref;
+
+ if (!pid)
+  return CKPT_PID_NULL;
+
+ objref = ckpt_obj_lookup(ctx, pid, CKPT_OBJ_PID);
+ if (objref < 0) {
+  objref = 0;  /* outside namespace */
+  WARN(ckpt_pid_vnr(ctx, pid),
+       "unrecognized pid %d (vnr: %d)\n",
+       pid->numbers[0].nr, pid_vnr(pid));
+ }
+
+ return objref;
+}
```

```
+
+/* dump the array that describes the tasks tree */
+static int checkpoint_tree(struct ckpt_ctx *ctx)
+{
+ struct pid_namespace *root_pidns;
+ struct pid_namespace *task_pidns;
+ struct task_struct *task, *pp;
+ struct ckpt_hdr_tree *hh;
+ struct ckpt_task_pids *h;
+ int pos, n = 0;
+ int ret;
+
+ hh = ckpt_hdr_get_type(ctx, sizeof(*hh), CKPT_HDR_TREE);
+ if (!hh)
+  return -ENOMEM;
+
+ hh->nr_tasks = ctx->nr_tasks;
+
+ ret = ckpt_write_obj(ctx, &hh->h);
+ ckpt_hdr_put(ctx, hh);
+ if (ret < 0)
   return ret;

- ret = checkpoint_pids(ctx);
+ pos = ctx->nr_tasks * sizeof(*h);
+ ret = ckpt_write_obj_type(ctx, NULL, pos, CKPT_HDR_BUFFER);
  if (ret < 0)
   return ret;

- if (ctx->nr_vpids == 0)
-  return 0;
+ h = ckpt_hdr_get(ctx, PAGE_SIZE);
+ if (!h)
+  return -ENOMEM;
+
+ root_pidns = ctx->root_nsproxy->pid_ns;

- return checkpoint_vpids(ctx);
+ /*
+  * Record the pids, as seen from ctx->root_nsproxy->pidns,
+  * saving them as pid-object tags. This info is needed at
+  * restart to unambiguously dereference tasks.
+  *
+  * NOTE: if a pgid/sid refers to an ancestor namespace, we
+  * simply store a "0" for the tag. This allows restart to
+  * correctly restore such out-of-pidns references. However,
+  * this means that restart of non-container checkpoint may be
+  * inaccurate: sid of different tasks may come from different
```

```
+  * out-of-pidns pids originally, but will refer to the same
+  * one after the restart. (I wonder if that is needed ?)
+  */
+
+ while (n < ctx->nr_tasks) {
+  pos = 0;
+
+  rcu_read_lock();
+  while (pos < PAGE_SIZE / sizeof(*h) && n < ctx->nr_tasks) {
+   task = ctx->tasks_arr[pos];
+   pp = task->real_parent;
+   task_pidns = task_active_pid_ns(task);
+
+   h[n].vpid = ckpt_lookup_pid(ctx, task_pid(task));
+   h[n].vtgid = ckpt_lookup_pid(ctx, task_tgid(task));
+   h[n].vpgid = ckpt_lookup_pid(ctx, task_pgrp(task));
+   h[n].vsid = ckpt_lookup_pid(ctx, task_session(task));
+   h[n].vppid = ckpt_lookup_pid(ctx, task_pid(pp));
+
+   h[n].depth = task_pidns->level - root_pidns->level;
+
+   ckpt_debug("task[%d]: vpid %d vtgid %d "
+       "vpgid %d vsid %d parent %d\n",
+       pos, task_pid_nr_ns(task, root_pidns),
+       task_tgid_nr_ns(task, root_pidns),
+       task_pgrp_nr_ns(task, root_pidns),
+       task_session_nr_ns(task, root_pidns),
+       task_tgid_nr_ns(pp, root_pidns));
+
+   if (h[n].vpid <= 0 || h[n].vtgid <= 0 ||
+       h[n].vpgid < 0 || h[n].vsid < 0)
+    break;
+
+   pos++;
+   n++;
+  }
+  rcu_read_unlock();
+
+  /*
+   * The root task's vppid must be 0 because the parent pid
+   * was never collected. If that is not the case, something
+   * went wrong ...
+   */
+  WARN_ON(n == pos && h[0].vppid != 0);
+
+  ret = ckpt_kwrite(ctx, h, pos * sizeof(*h));
+  if (ret < 0)
+   break;
```

```
+ }
+
+ _ckpt_hdr_put(ctx, h, PAGE_SIZE);
+ return ret;
 }

 static struct task_struct *get_freezer_task(struct task_struct *root_task)
@@ -714,6 +889,9 @@ long do_checkpoint(struct ckpt_ctx *ctx, pid_t pid)
  ret = checkpoint_container(ctx);
  if (ret < 0)
   goto out;
+ ret = checkpoint_pids(ctx);
+ if (ret < 0)
+  goto out;
  ret = checkpoint_tree(ctx);
  if (ret < 0)
   goto out;
diff --git a/kernel/checkpoint/process.c b/kernel/checkpoint/process.c
index 33bb62d..a39a6d4 100644
--- a/kernel/checkpoint/process.c
+++ b/kernel/checkpoint/process.c
@@ -25,56 +25,6 @@
 #include <linux/checkpoint.h>
 #include <linux/pid_namespace.h>

-pid_t ckpt_pid_nr(struct ckpt_ctx *ctx, struct pid *pid)
-{
- return pid ? pid_nr_ns(pid, ctx->root_nsproxy->pid_ns) : CKPT_PID_NULL;
-}
-
-/* must be called with tasklist_lock or rcu_read_lock() held */
-struct pid *_ckpt_find_pgrp(struct ckpt_ctx *ctx, pid_t pgid)
-{
- struct task_struct *p;
- struct pid *pgrp;
-
- if (pgid == 0) {
-  /*
-   * At checkpoint the pgid owner lived in an ancestor
-   * pid-ns. The best we can do (sanely and safely) is
-   * to examine the parent of this restart's root: if in
-   * a distinct pid-ns, use its pgrp; otherwise fail.
-   */
-  p = ctx->root_task->real_parent;
-  if (p->nsproxy->pid_ns == current->nsproxy->pid_ns)
-   return NULL;
-  pgrp = task_pgrp(p);
- } else {
```

```
-  /*
-   * Find the owner process of this pgid (it must exist
-   * if pgrp exists). It must be a thread group leader.
-   */
-  pgrp = find_pid_ns(pgid, ctx->root_nsproxy->pid_ns);
-  p = pid_task(pgrp, PIDTYPE_PID);
-  if (!p || !thread_group_leader(p))
-   return NULL;
-  /*
-   * The pgrp must "belong" to our restart tree (compare
-   * p->checkpoint_ctx to ours). This prevents malicious
-   * input from (guessing and) using unrelated pgrps. If
-   * the owner is dead, then it doesn't have a context,
-   * so instead compare against its (real) parent's.
-   */
-  if (p->exit_state == EXIT_ZOMBIE)
-   p = p->real_parent;
-  if (p->checkpoint_ctx != ctx)
-   return NULL;
- }
-
- if (task_session(current) != task_session(p))
-  return NULL;
-
- return pgrp;
-}
-
 #ifdef CONFIG_FUTEX
 static void save_task_robust_futex_list(struct ckpt_hdr_task *h,
      struct task_struct *t)
@@ -601,7 +551,7 @@ static int restore_task_ns(struct ckpt_ctx *ctx)
   goto out;
  }

- if (nsproxy != task_nsproxy(current)) {
+ if (nsproxy != current->nsproxy) {
   spin_lock(&checkpoint_nslock);
   if (!nsproxy->pid_ns)
    nsproxy->pid_ns = get_pid_ns(current->nsproxy->pid_ns);
@@ -610,7 +560,7 @@ static int restore_task_ns(struct ckpt_ctx *ctx)
   switch_task_namespaces(current, nsproxy);
  }
  out:
- ckpt_debug("nsproxy: ret %d (%p)\n", ret, task_nsproxy(current));
+ ckpt_debug("nsproxy: ret %d (%p)\n", ret, current->nsproxy);
  ckpt_hdr_put(ctx, h);
  return ret;
 }
```

```
@@ -839,56 +789,6 @@ int restore_restart_block(struct ckpt_ctx *ctx)
  return ret;
 }

-static int restore_task_pgid(struct ckpt_ctx *ctx)
-{
- struct task_struct *task = current;
- struct pid *pgrp;
- pid_t pgid;
- int ret;
-
- /*
-  * We enforce the following restrictions on restoring pgrp:
-  *  1) Only thread group leaders restore pgrp
-  *  2) Session leader cannot change own pgrp
-  *  3) Owner of pgrp must belong to same restart tree
-  *  4) Must have same session as other tasks in same pgrp
-  *  5) Change must pass setpgid security callback
-  *
-  * TODO - check if we need additional restrictions ?
-  */
-
- if (!thread_group_leader(task))  /* (1) */
-  return 0;
-
- pgid = ctx->pids_arr[ctx->active_pid].vpgid;
-
- if (pgid == task_pgrp_nr_ns(task, ctx->root_nsproxy->pid_ns))
-  return 0;  /* nothing to do */
-
- if (task->signal->leader)  /* (2) */
-  return -EINVAL;
-
- ret = -EINVAL;
-
- write_lock_irq(&tasklist_lock);
- pgrp = _ckpt_find_pgrp(ctx, pgid);  /* (3) and (4) */
- if (pgrp && task_pgrp(task) != pgrp) {
-  ret = security_task_setpgid(task, pgid);  /* (5) */
-  if (!ret)
-   change_pid(task, PIDTYPE_PGID, pgrp);
- }
- write_unlock_irq(&tasklist_lock);
-
- /* self-restart: be tolerant if old pgid isn't found */
- if (ctx->uflags & RESTART_TASKSELF)
-  ret = 0;
-
```

```
- if (ret < 0)
-  ckpt_err(ctx, ret, "setting pgid\n");
-
- return ret;
-}
-
 /* prepare the task for restore */
 int pre_restore_task(void)
 {
@@ -934,9 +834,6 @@ int restore_task(struct ckpt_ctx *ctx)
  if (ret)
   goto out;

- ret = restore_task_pgid(ctx);
- if (ret < 0)
-  goto out;
  ret = restore_thread(ctx);
  ckpt_debug("thread %d\n", ret);
  if (ret < 0)
@@ -949,6 +846,7 @@ int restore_task(struct ckpt_ctx *ctx)
  ckpt_debug("cpu %d\n", ret);
  if (ret < 0)
   goto out;
+
  ret = restore_task_objs(ctx);
  ckpt_debug("objs %d\n", ret);
  if (ret < 0)
diff --git a/kernel/checkpoint/restart.c b/kernel/checkpoint/restart.c
index 9aaab4f..25e3b6d 100644
--- a/kernel/checkpoint/restart.c
+++ b/kernel/checkpoint/restart.c
@@ -143,12 +143,16 @@ void restore_debug_free(struct ckpt_ctx *ctx)
  ckpt_debug("%d tasks registered, nr_tasks was %d nr_total %d\n",
      count, ctx->nr_tasks, atomic_read(&ctx->nr_total));

- ckpt_debug("active pid was %d, ctx->errno %d\n", ctx->active_pid,
+ ckpt_debug("active pid was %d, ctx->errno %d\n", ctx->active_task,
      ctx->errno);
  ckpt_debug("kflags %lu uflags %lu oflags %lu", ctx->kflags,
      ctx->uflags, ctx->oflags);
- for (i = 0; i < ctx->nr_pids; i++)
-  ckpt_debug("task[%d] to run %d\n", i, ctx->pids_arr[i].vpid);
+ for (i = 0; i < ctx->nr_tasks; i++) {
+  if (!ctx->tasks_arr[i])
+   continue;
+  ckpt_debug("task[%d] to run %d\n", i,
+      ckpt_task_vnr(ctx, ctx->tasks_arr[i]));
+ }
```

```
  list_for_each_entry_safe(s, p, &ctx->task_status, list) {
   if (s->flags & RESTART_DBG_COORD)
@@ -733,73 +737,430 @@ static int restore_read_tail(struct ckpt_ctx *ctx)
  return ret;
 }

-/* restore_read_tree - read the tasks tree into the checkpoint context */
-static int restore_read_tree(struct ckpt_ctx *ctx)
+static struct pid *restore_validate_pids(struct ckpt_ctx *ctx,
+       struct ckpt_pids *h)
 {
- struct ckpt_hdr_tree *h;
- int size, ret;
-
- h = ckpt_read_obj_type(ctx, sizeof(*h), CKPT_HDR_TREE);
- if (IS_ERR(h))
-  return PTR_ERR(h);
+ struct task_struct *task;
+ struct pid *pid, *pidret = NULL;
+ int i, j;

- ret = -EINVAL;
- if (h->nr_tasks <= 0)
+ rcu_read_lock();
+ task = find_task_by_pid_ns(h->numbers[0], ctx->root_nsproxy->pid_ns);
+ if (!task || task->checkpoint_ctx != ctx)
   goto out;
-
- ctx->nr_pids = h->nr_tasks;
- size = sizeof(*ctx->pids_arr) * ctx->nr_pids;
- if (size <= 0)  /* overflow ? */
+ pid = task_pid(task);
+ if (pid->level - ctx->root_nsproxy->pid_ns->level != h->depth)
   goto out;
+ for (i = 0, j = pid->level - h->depth; i <= h->depth; i++, j++)
+  if (pid->numbers[j].nr != h->numbers[i])
+   goto out;
+ pidret = get_pid(pid);
+ out:
+ rcu_read_unlock();
+ return pidret;
+}

- ctx->pids_arr = kmalloc(size, GFP_KERNEL);
- if (!ctx->pids_arr) {
-  ret = -ENOMEM;
-  goto out;
```

```
+/* restore_read_pids - read the pids and validate against new tree */
+static int restore_read_pids(struct ckpt_ctx *ctx)
+{
+ struct ckpt_hdr_pids *hh;
+ struct ckpt_pids *h;
+ struct pid *pid;
+ char *buf = NULL;
+ int p = 0, n = 0, s = 0;
+ int offset, len, tot, ret;
+ int nr_pids, nr_vpids;
+ int vpids = 0;
+
+ hh = ckpt_read_obj_type(ctx, sizeof(*hh), CKPT_HDR_PIDS);
+ if (IS_ERR(hh))
+  return PTR_ERR(hh);
+ nr_pids = hh->nr_pids;
+ nr_vpids = hh->nr_vpids;
+ offset = hh->offset;
+ ckpt_hdr_put(ctx, hh);
+
+ if (nr_pids == 0)
+  return -EINVAL;
+
+ tot = nr_pids * sizeof(*h) + nr_vpids * sizeof(__s32);
+ if (tot <= 0)   /* overflow ? */
+  return -EINVAL;
+
+ ret = _ckpt_read_obj_type(ctx, NULL, tot, CKPT_HDR_BUFFER);
+ if (ret < 0)
+  return ret;
+
+ buf = ckpt_hdr_get(ctx, PAGE_SIZE);
+ if (!buf)
+  return -ENOMEM;
+
+ while (n < nr_pids && s < tot) {
+  len = min(tot, (int) PAGE_SIZE - p);
+  ret = ckpt_kread(ctx, buf + p, len);
+  if (ret < 0)
+   break;
+
+  p = 0;
+
+  ret = -EINVAL;
+  while (n < nr_pids && len > 0) {
+   s = sizeof(*h);
+   if (s > len)
+    break;
```

```
+
+   h = (struct ckpt_pids *) &buf[p];
+   if (h->depth < 0)
+    goto out;
+
+   s += h->depth * sizeof(__s32);
+   if (s < 0)  /* overflow */
+    goto out;
+   if (s > len)
+    break;
+
+   pid = restore_validate_pids(ctx, h);
+   if (!pid)
+    goto out;
+
+   ret = ckpt_obj_insert(ctx, pid,
+           n + 1 + offset, CKPT_OBJ_PID);
+   put_pid(pid);
+   if (ret < 0)
+    goto out;
+
+   vpids += h->depth;
+   if (vpids > nr_vpids)
+    goto out;
+
+   len -= s;
+   p += s;
+   n++;
+  }
+
+  memcpy(buf, buf + p, len);
+  tot -= len - p;
  }
- ret = _ckpt_read_buffer(ctx, ctx->pids_arr, size);
+
+ if (n == nr_pids && tot == 0)
+  ret = 0;
  out:
- ckpt_hdr_put(ctx, h);
+ _ckpt_hdr_put(ctx, buf, PAGE_SIZE);
  return ret;
 }

-/*
- * read all the vpids - we don't actually care about them,
- * userspace did
- */
-static int restore_slurp_vpids(struct ckpt_ctx *ctx)
```

```
+static int restore_task_pgrp(struct ckpt_ctx *ctx,
+        struct task_struct *task,
+        struct pid *pgrp)
 {
- int size, ret, i;
+ struct task_struct *p;
+ pid_t pgid;
+ int ret;

- for (i = 0; i < ctx->nr_pids; i++)
-  ctx->nr_vpids += ctx->pids_arr[i].depth;
+ /*
+  * We enforce the following restrictions on restoring pgrp:
+  *  1) Only thread group leaders restore pgrp
+  *  2) Session leader cannot change own pgrp
+  *  3) Must have same session as other tasks in same pgrp
+  *  4) Change must pass setpgid security callback
+  *
+  * By now we know that the pgrp belongs to our restart tree,
+  * because we checked then in restore_validate_pids().
+  *
+  * TODO - check if we need additional restrictions ?
+  */

- if (ctx->nr_vpids == 0)
+ if (!thread_group_leader(task))  /* (1) */
   return 0;

- size = sizeof(__s32) * ctx->nr_vpids;
- if (size < 0)  /* overflow ? */
+ if (task->signal->leader)  /* (2) */
+  return -EINVAL;
+
+ ret = -EINVAL;
+ write_lock_irq(&tasklist_lock);
+ if (pgrp == NULL) {
+  /*
+   * At checkpoint the pgid owner lived in an ancestor
+   * pid-ns. The best we can do (sanely and safely) is
+   * to examine the parent of this restart's root: if in
+   * a distinct pid-ns, use its pgrp; otherwise fail.
+   */
+  p = ctx->root_task->real_parent;
+  if (p->nsproxy->pid_ns == current->nsproxy->pid_ns)
+   goto unlock;
+ } else {
+  /* Find the owner process - must be a thread leader */
+  p = pid_task(pgrp, PIDTYPE_PID);
```

```
+  if (!thread_group_leader(p))
+   goto unlock;
+  /*
+   * If the owner is dead, then it has no context, so
+   * instead compare against its (real) parent's.
+   */
+  if (p->exit_state == EXIT_ZOMBIE)
+   p = p->real_parent;
+  if (p->checkpoint_ctx != ctx)
+   goto unlock;
+ }
+
+ if (task_session(task) != task_session(p))  /* (3) */
+  goto unlock;
+
+ if (task_pgrp(task) != pgrp) {
+  pgid = pid_vnr(pgrp);
+  ret = security_task_setpgid(task, pgid);  /* (4) */
+  if (!ret)
+   change_pid(task, PIDTYPE_PGID, pgrp);
+ } else {
+  /*
+   * Weird!! We wouldn't be called in this case!!
+   * Anyway, the pgrp is now ok, so silently ignore
+   */
+  WARN(1, "pgid changed while restore\n");
+  ret = 0;
+ }
+
+ unlock:
+ write_unlock_irq(&tasklist_lock);
+
+ /* self-restart: be tolerant if old pgid isn't found */
+ if (ctx->uflags & RESTART_TASKSELF)
+  ret = 0;
+
+ if (ret < 0)
+  ckpt_err(ctx, ret, "restore pgid\n");
+
+ return ret;
+}
+
+static struct task_struct *restore_validate_task(struct ckpt_ctx *ctx,
+       struct ckpt_task_pids *h)
+{
+ struct task_struct *task;
+ struct pid_namespace *root_pidns;
+ struct pid *pid, *tgid;
```

```
+ struct pid *pgrp = NULL;
+ struct pid *session = NULL;
+ int ret = 0;
+
+ pid = ckpt_obj_fetch(ctx, h->vpid, CKPT_OBJ_PID);
+ tgid = ckpt_obj_fetch(ctx, h->vtgid, CKPT_OBJ_PID);
+
+ if (h->vpgid != 0 && h->vpgid != CKPT_PID_ROOT)
+  pgrp = ckpt_obj_fetch(ctx, h->vpgid, CKPT_OBJ_PID);
+ if (h->vsid != 0 && h->vsid != CKPT_PID_ROOT)
+  session = ckpt_obj_fetch(ctx, h->vsid, CKPT_OBJ_PID);
+ if (IS_ERR(pid))
+  return ERR_CAST(pid);
+ if (IS_ERR(tgid))
+  return ERR_CAST(tgid);
+ if (IS_ERR(pgrp))
+  return ERR_CAST(pgrp);
+ if (IS_ERR(session))
+  return ERR_CAST(session);
+
+ rcu_read_lock();
+ read_lock(&tasklist_lock);
+
+ task = pid_task(pid, PIDTYPE_PID);  /* get the owner */
+ root_pidns = ctx->root_nsproxy->pid_ns;
+
+ /*
+  * If vpgid/vsid is 0, then at checkpoint time it was a
+  * reference to a pid from an ancestor pid-ns, inherited
+  * through the root task. For vsid, verify that this is the
+  * case now. For pgid, verify that the coordinator (or its
+  * parent) is in ancestor pis-ns, and use their pid.
+  *
+  * NOTE: this means that restart of non-container checkpoint
+  * may be inaccurate: sid of different tasks may come from
+  * different out-of-pidns pids originally, but will refer to
+  * the same one after the restart. Don't think it's useful.
+  *
+  * If vpgid/vsid is CKPT_PID_ROOT, then at checkpoint time it
+  * was the same as the root task's sid, and the root task was
+  * not a session leader (otherwise, they would be 1). We now
+  * verify that the restarted root task isn't a session leader
+  * and substitue its sid for the vpgid/vsid in question.
+  */
+
+ if (!session) {
+  if (h->vsid == CKPT_PID_ROOT &&
+      !ctx->root_task->signal->leader &&
```

```
+      task_session(task) == task_session(ctx->root_task)) {
+    /* must have same sid as root task */
+    session = task_session(ctx->root_task);
+  } else if (h->vsid == 0 &&
+      pid_nr_ns(task_session(task), root_pidns) == 0) {
+    /*  must be in different pid-ns */
+    session = task_session(task);
+  }
+ }
+
+ if (!pgrp) {
+  if (h->vpgid == CKPT_PID_ROOT &&
+      !ctx->root_task->signal->leader &&
+      task_pgrp(task) == task_session(ctx->root_task)) {
+    /* must have same sid as root task */
+    pgrp = task_session(ctx->root_task);
+  } else if (h->vsid == 0) {
+    struct task_struct *t = current;
+    /*  must be in different pid-ns - try to grab
+        the session of coord's or coord's parent */
+    if (pid_nr_ns(task_pgrp(t), root_pidns) != 0)
+     t = rcu_dereference(current->real_parent);
+    if (pid_nr_ns(task_pgrp(t), root_pidns) == 0)
+     pgrp = task_pgrp(t);
+  } else
+    pgrp = ERR_PTR(-EINVAL);
+ }
+
+ /*
+  * Verify that the task matches our expectations in terms of:
+  * checkpoint context, tgid, session, and pidns depth. Don't
+  * enforce pgrp as it may not match (restored below)
+  */
+
+ if (!IS_ERR(task) && task->checkpoint_ctx == ctx &&
+     tgid == task_tgid(task) && session == task_session(task) &&
+     h->depth == task_active_pid_ns(task)->level - root_pidns->level) {
+  /* yes, this is our task ! */
+  get_task_struct(task);
+ } else {
+  task = ERR_PTR(-EINVAL);
+ }
+
+ rcu_read_unlock();
+ read_unlock(&tasklist_lock);
+
+ if (IS_ERR(task))
+  return task;
```

```
+
+ /*
+  * The pgrp may not match the task's pgrp (the restarted tree
+  * does not guarantee this), or could be NULL e.g., if we are
+  * restartign from a subtree. Call restore_task_pgrp() to fix.
+  */
+ if (pgrp != task_pgrp(task)) {
+  ret = restore_task_pgrp(ctx, task, pgrp);
+  if (ret < 0) {
+   put_task_struct(task);
+   task = ERR_PTR(ret);
+  }
+ }
+
+ return task;
+}
+
+/* restore_read_tree - read the tasks tree into the checkpoint context */
+static int restore_read_tree(struct ckpt_ctx *ctx)
+{
+ struct ckpt_hdr_tree *hh;
+ struct ckpt_task_pids *h;
+ struct task_struct *task;
+ int nr_tasks;
+ int len, ret;
+ int p, n = 0;
+
+ hh = ckpt_read_obj_type(ctx, sizeof(*hh), CKPT_HDR_TREE);
+ if (IS_ERR(hh))
+  return PTR_ERR(hh);
+ nr_tasks = hh->nr_tasks;
+ ckpt_hdr_put(ctx, hh);
+
+ if (nr_tasks <= 0)
+  return -EINVAL;
+
+ len = sizeof(*h) * nr_tasks;
+ if (len <= 0)  /* overflow ? */
   return -EINVAL;

- ret = ckpt_read_consume(ctx, size + sizeof(struct ckpt_hdr),
-    CKPT_HDR_BUFFER);
+ ret = _ckpt_read_obj_type(ctx, NULL, len, CKPT_HDR_BUFFER);
+ if (ret < 0)
+  return ret;
+
+ /*
+  * Actual tasks count may exceed ctx->nr_tasks due of 'dead'
```

```
+  * tasks used as place-holders for PGIDs, but not fall short.
+  */
+ if (atomic_read(&ctx->nr_total) < nr_tasks)
+  return -ESRCH;
+
+ /* last entry intentionally remains NULL: see get_active_task() */
+ ctx->nr_tasks = nr_tasks;
+ ctx->tasks_arr = kzalloc(len + 1, GFP_KERNEL);
+ if (!ctx->tasks_arr)
+  return -ENOMEM;
+
+ h = ckpt_hdr_get(ctx, PAGE_SIZE);
+ if (!h) {
+  ret = -ENOMEM;
+  goto out;
+ }
+
+ while (n < nr_tasks) {
+  len = min((int)(PAGE_SIZE / sizeof(*h)), nr_tasks - n);
+  ret = ckpt_kread(ctx, h, len * sizeof(*h));
+  if (ret < 0)
+   break;
+
+  p = 0;
+
+  ret = -EINVAL;
+  while (n < nr_tasks && p < len) {
+   task = restore_validate_task(ctx, &h[p]);
+   if (IS_ERR(task)) {
+    ret = PTR_ERR(task);
+    goto out;
+   }
+   ctx->tasks_arr[n] = task;
+   p++;
+   n++;
+  }
+ }
+
+ if (n == nr_tasks)
+  ret = 0;
+
+#ifdef CONFIG_CHECKPOINT_DEBUG
+ ckpt_debug("restoring tasks.....\n");
+ for (n = 0; n < nr_tasks; n++)
+  ckpt_debug("task[%d]: pid %d (%d)\n", n,
+      task_pid_vnr(ctx->tasks_arr[n]),
+      task_pid_nr_ns(ctx->tasks_arr[n],
+       ctx->root_nsproxy->pid_ns));
```

```
+ ckpt_debug("......................\n");
+#endif
+
+ out:
+ _ckpt_hdr_put(ctx, h, PAGE_SIZE);
  return ret;
 }

 static inline int all_tasks_activated(struct ckpt_ctx *ctx)
 {
- return (ctx->active_pid == ctx->nr_pids);
+ return (ctx->active_task == ctx->nr_tasks);
 }

-static inline pid_t get_active_pid(struct ckpt_ctx *ctx)
+static inline struct task_struct *get_active_task(struct ckpt_ctx *ctx)
 {
- int active = ctx->active_pid;
- return active >= 0 ? ctx->pids_arr[active].vpid : 0;
+ int active = ctx->active_task;
+
+ /*
+  * This is safe even for (active == ctx->nr)_tasks because the
+  * ctx->tasks_arr[] array has an extra (NULL) slot at the end,
+  * This may happen if due to bad input a restarting task is
+  * not in the ctx->tasks_arr, but calls wait_task_active().
+  * It may then wake up and call us with active_task already
+  * maxed out...
+  */
+ BUG_ON(active > ctx->nr_tasks);
+ return active >= 0 ? ctx->tasks_arr[active] : NULL;
 }

-static inline int is_task_active(struct ckpt_ctx *ctx, pid_t pid)
+static inline int is_task_active(struct ckpt_ctx *ctx, struct task_struct *task)
 {
- return get_active_pid(ctx) == pid;
+ return get_active_task(ctx) == task;
 }

 /*
@@ -857,36 +1218,17 @@ static void restore_task_done(struct ckpt_ctx *ctx)
 {
  if (atomic_dec_and_test(&ctx->nr_total))
   complete(&ctx->complete);
- BUG_ON(atomic_read(&ctx->nr_total) < 0);
 }
```

```
 static int restore_activate_next(struct ckpt_ctx *ctx)
 {
- struct task_struct *task;
- pid_t pid;
-
- ctx->active_pid++;
+ ctx->active_task++;

- BUG_ON(ctx->active_pid > ctx->nr_pids);
+ BUG_ON(ctx->active_task > ctx->nr_tasks);

  if (!all_tasks_activated(ctx)) {
   /* wake up next task in line to restore its state */
-  pid = get_active_pid(ctx);
-
-  rcu_read_lock();
-  task = find_task_by_pid_ns(pid,
-     task_active_pid_ns(ctx->root_task));
-  /* target task must have same restart context */
-  if (task && task->checkpoint_ctx == ctx)
-   wake_up_process(task);
-  else
-   task = NULL;
-  rcu_read_unlock();
-
-  if (!task) {
-   ckpt_err(ctx, -ESRCH, "task %d not found\n", pid);
-   return -ESRCH;
-  }
+  wake_up_process(get_active_task(ctx));
  } else {
   /* wake up ghosts tasks so that they can terminate */
   wake_up_all(&ctx->ghostq);
@@ -897,15 +1239,14 @@ static int restore_activate_next(struct ckpt_ctx *ctx)

 static int wait_task_active(struct ckpt_ctx *ctx)
 {
- pid_t pid = task_pid_nr_ns(current, task_active_pid_ns(ctx->root_task));
  int ret;

- ckpt_debug("pid %d waiting\n", pid);
+ ckpt_debug("pid %d waiting\n", ckpt_task_vnr(ctx, current));
  ret = wait_event_interruptible(ctx->waitq,
-           is_task_active(ctx, pid) ||
+           is_task_active(ctx, current) ||
            ckpt_test_error(ctx));
  ckpt_debug("active %d < %d (ret %d, errno %d)\n",
-     ctx->active_pid, ctx->nr_pids, ret, ctx->errno);
```

```
+     ctx->active_task, ctx->nr_tasks, ret, ctx->errno);
  if (ckpt_test_error(ctx))
   return ckpt_get_error(ctx);
  return 0;
@@ -1140,29 +1481,36 @@ static int __prepare_descendants(struct task_struct *task, void
*data)
  */
 static int prepare_descendants(struct ckpt_ctx *ctx, struct task_struct *root)
 {
- int nr_pids;
+ int nr_total;

- nr_pids = walk_task_subtree(root, __prepare_descendants, ctx);
- ckpt_debug("nr %d/%d\n", ctx->nr_pids, nr_pids);
- if (nr_pids < 0)
-  return nr_pids;
+ nr_total = walk_task_subtree(root, __prepare_descendants, ctx);
+ ckpt_debug("nr %d/%d\n", ctx->nr_tasks, nr_total);
+ if (nr_total < 0)
+  return nr_total;

   /*
-  * Actual tasks count may exceed ctx->nr_pids due of 'dead'
-  * tasks used as place-holders for PGIDs, but not fall short.
+  * Normally, ctx->nr_total holds the total descendants, and
+  * decremented by each descendant that completes the restart
+  * or terminates (e.g. ghost/zombie). If a descendant dies
+  * _here_ (after setting ->checkpoint_ctx but before setting
+  * ctx->nr_total) it decrements it in exit_checkpoint(), so we
+  * need to _add_ the total (no set it). If all of them die
+  * here, then none will call complete() and the coordinator
+  * will hang - so test for zero already.
    */
- if (nr_pids < ctx->nr_pids)
-  return -ESRCH;
+        if (atomic_add_return(nr_total, &ctx->nr_total) == 0) {
+  ckpt_err(ctx, -EAGAIN, "descendant(s) terminated unexpectedly");
+  return -EAGAIN;
+ }

- atomic_set(&ctx->nr_total, nr_pids);
- return nr_pids;
+ return nr_total;
 }

 static int wait_all_tasks_finish(struct ckpt_ctx *ctx)
 {
  int ret;
```

```
- BUG_ON(ctx->active_pid != -1);
+ BUG_ON(ctx->active_task != -1);
  ret = restore_activate_next(ctx);
  if (ret < 0)
   return ret;
@@ -1206,6 +1554,8 @@ static int init_restart_ctx(struct ckpt_ctx *ctx, pid_t pid)
   * occurs then ckpt_ctx_free() is eventually called.
   */

+ ckpt_debug("restore context: root pid %d\n", pid);
+
  if (!choose_root_task(ctx, pid))
   return -ESRCH;

@@ -1219,7 +1569,8 @@ static int init_restart_ctx(struct ckpt_ctx *ctx, pid_t pid)
  if (!nsproxy)
   return -ESRCH;

- ctx->active_pid = -1; /* see restore_activate_next, get_active_pid */
+ ctx->active_task = -1; /* see restore_activate_next, get_active_task */
+ atomic_set(&ctx->nr_total, 0);

  return 0;
 }
@@ -1246,8 +1597,13 @@ static int do_restore_coord(struct ckpt_ctx *ctx, pid_t pid)
  ret = restore_debug_task(ctx, RESTART_DBG_COORD);
  if (ret < 0)
   return ret;
+
  restore_debug_running(ctx);

+ ret = init_restart_ctx(ctx, pid);
+ if (ret < 0)
+  return ret;
+
  ret = restore_read_header(ctx);
  ckpt_debug("restore header: %d\n", ret);
  if (ret < 0)
@@ -1256,22 +1612,6 @@ static int do_restore_coord(struct ckpt_ctx *ctx, pid_t pid)
  ckpt_debug("restore container: %d\n", ret);
  if (ret < 0)
   return ret;
- ret = restore_read_tree(ctx);
- ckpt_debug("restore tree: %d\n", ret);
- if (ret < 0)
-  return ret;
-
```

```
- ret = restore_slurp_vpids(ctx);
- ckpt_debug("read vpids %d\n", ret);
- if (ret < 0)
-  return ret;
-
- if ((ctx->uflags & RESTART_TASKSELF) && ctx->nr_pids != 1)
-  return -EINVAL;
-
- ret = init_restart_ctx(ctx, pid);
- if (ret < 0)
-  return ret;

  /*
   * Populate own ->checkpoint_ctx: if an ancestor attempts to
@@ -1284,7 +1624,7 @@ static int do_restore_coord(struct ckpt_ctx *ctx, pid_t pid)
    * We are a bad-behaving descendant: an ancestor must
    * have prepare_descendants() us as part of a restart.
    */
-  ckpt_debug("coord already has checkpoint_ctx\n");
+  ckpt_debug("coord/self already has checkpoint_ctx\n");
   return -EBUSY;
  }

@@ -1294,6 +1634,26 @@ static int do_restore_coord(struct ckpt_ctx *ctx, pid_t pid)
   * subtree that we marked for restart - see below.
   */

+ if (!(ctx->uflags & RESTART_TASKSELF)) {
+  /* prepare descendants' t->checkpoint_ctx point to coord */
+  ret = prepare_descendants(ctx, ctx->root_task);
+  ckpt_debug("restore prepare: %d\n", ret);
+  if (ret < 0)
+   goto out;
+ }
+
+ ret = restore_read_pids(ctx);
+ ckpt_debug("restore pids: %d\n", ret);
+ if (ret < 0)
+  goto out;
+ ret = restore_read_tree(ctx);
+ ckpt_debug("restore tree: %d\n", ret);
+ if (ret < 0)
+  goto out;
+
+ if ((ctx->uflags & RESTART_TASKSELF) && ctx->nr_tasks != 1)
+  return -EINVAL;
+
  if (ctx->uflags & RESTART_TASKSELF) {
```

```
   ret = pre_restore_task();
   ckpt_debug("pre restore task: %d\n", ret);
@@ -1304,11 +1664,6 @@ static int do_restore_coord(struct ckpt_ctx *ctx, pid_t pid)
   if (ret < 0)
    goto out;
  } else {
-  /* prepare descendants' t->checkpoint_ctx point to coord */
-  ret = prepare_descendants(ctx, ctx->root_task);
-  ckpt_debug("restore prepare: %d\n", ret);
-  if (ret < 0)
-   goto out;
   /* wait for all other tasks to complete do_restore_task() */
   ret = wait_all_tasks_finish(ctx);
   ckpt_debug("restore finish: %d\n", ret);
@@ -1459,12 +1814,13 @@ void exit_checkpoint(struct task_struct *tsk)

  /* no one else will touch this, because @tsk is dead already */
  ctx = tsk->checkpoint_ctx;
+ tsk->checkpoint_ctx = NULL;

  /* restarting zombies will activate next task in restart */
- if (tsk->flags & PF_RESTARTING) {
-  BUG_ON(ctx->active_pid == -1);
+ if (tsk->flags & PF_RESTARTING)
   restore_task_done(ctx);
- }
+ else
+  ckpt_set_error(ctx, -EAGAIN);

  ckpt_ctx_put(ctx);
 }
diff --git a/kernel/checkpoint/sys.c b/kernel/checkpoint/sys.c
index 2383db9..0e59f18 100644
--- a/kernel/checkpoint/sys.c
+++ b/kernel/checkpoint/sys.c
@@ -241,8 +241,6 @@ static void ckpt_ctx_free(struct ckpt_ctx *ctx)
  if (ctx->tasks_arr)
   task_arr_free(ctx);

- if (ctx->coord_pidns)
-  put_pid_ns(ctx->coord_pidns);
  if (ctx->root_nsproxy)
   put_nsproxy(ctx->root_nsproxy);
  if (ctx->root_task)
@@ -252,8 +250,6 @@ static void ckpt_ctx_free(struct ckpt_ctx *ctx)

  free_page((unsigned long) ctx->scratch_page);
```

```
- kfree(ctx->pids_arr);
-
 #ifdef CONFIG_NETNS_CHECKPOINT
  sock_listening_list_free(&ctx->listen_sockets);
 #endif
@@ -274,7 +270,6 @@ static struct ckpt_ctx *ckpt_ctx_alloc(int fd, unsigned long uflags,
  ctx->uflags = uflags;
  ctx->kflags = kflags;
  ctx->ktime_begin = ktime_get();
- ctx->coord_pidns = get_pid_ns(current->nsproxy->pid_ns);

  atomic_set(&ctx->refcount, 0);
  INIT_LIST_HEAD(&ctx->pgarr_list);
diff --git a/kernel/signal.c b/kernel/signal.c
index 3842f5d..718b940 100644
--- a/kernel/signal.c
+++ b/kernel/signal.c
@@ -3218,12 +3218,12 @@ static int checkpoint_signal(struct ckpt_ctx *ctx, struct task_struct *t)

  /* tty */
  if (signal->leader) {
-  h->tty_old_pgrp = ckpt_pid_nr(ctx, signal->tty_old_pgrp);
+  h->tty_old_pgrp = ckpt_pid_vnr(ctx, signal->tty_old_pgrp);
   tty = tty_kref_get(signal->tty);
   if (tty) {
    /* irq is already disabled */
    spin_lock(&tty->ctrl_lock);
-   h->tty_pgrp = ckpt_pid_nr(ctx, tty->pgrp);
+   h->tty_pgrp = ckpt_pid_vnr(ctx, tty->pgrp);
    spin_unlock(&tty->ctrl_lock);
    tty_kref_put(tty);
   }
@@ -3236,7 +3236,7 @@ static int checkpoint_signal(struct ckpt_ctx *ctx, struct task_struct *t)
   * and set tty_objref = 0. It will not be explicitly restored,
   * but rather inherited from parent pid-ns at restart time.
   */
- if (tty && ckpt_pid_nr(ctx, tty->session) > 0) {
+ if (tty && ckpt_pid_vnr(ctx, tty->session) > 0) {
   h->tty_objref = checkpoint_obj(ctx, tty, CKPT_OBJ_TTY);
   if (h->tty_objref < 0)
    ret = h->tty_objref;
@@ -3385,7 +3385,7 @@ static int restore_signal(struct ckpt_ctx *ctx)
  /* tty - tty_old_pgrp */
  if (current->signal->leader && h->tty_old_pgrp != CKPT_PID_NULL) {
   rcu_read_lock();
-  pgrp = get_pid(_ckpt_find_pgrp(ctx, h->tty_old_pgrp));
+  pgrp = NULL;  /* temp until next patch */
   rcu_read_unlock();
```

```
   if (!pgrp)
    goto out;
--
1.7.1
```

_____

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