

---

Subject: Re: [PATCH 4/5] c/r: checkpoint and restart pids objects  
Posted by [Sukadev Bhattiprolu](#) on Sat, 05 Feb 2011 21:43:18 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

Oren:

I am still reviewing this patchset, but have a few questions/comments below on this patch.

| From: Oren Laadan <[orenl@cs.columbia.edu](mailto:orenl@cs.columbia.edu)>  
| Subject: [PATCH 4/5] c/r: checkpoint and restart pids objects

| 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 kernel `_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 namespaces 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 handle that. So only linear pid-nesting works well and not arbitrary tree.

| DISCLAIMER: 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 the tasks. Entries are of fixed size, and hold the objref tags to the shared pids objects rather than actual pid numbers.

(the old `vuids_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 namespace are disallowed. 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 pids-ns, because it is quite complicated to find out 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 different 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 container 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 | 22 +-
include/linux/checkpoint_types.h | 10 +-
kernel/checkpoint/checkpoint.c | 440 ++++++-----
kernel/checkpoint/process.c | 108 +-----
kernel/checkpoint/restart.c | 551 ++++++-----
kernel/checkpoint/sys.c | 5 -
6 files changed, 773 insertions(+), 363 deletions(-)
```

diff --git a/include/linux/checkpoint\_hdr.h b/include/linux/checkpoint\_hdr.h

index 922eff0..c0a548a 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 +360,32 @@ struct ckpt\_hdr\_container {

\*/

} \_\_attribute\_\_((aligned(8)));

+/ \* pids array \*/

+struct ckpt\_hdr\_pids {

+ struct ckpt\_hdr h;

+ \_\_u32 nr\_pids;

+ \_\_u32 nr\_vpids;

```
| +} __attribute__((aligned(8)));
```

For consistency can we call this ckpt\_hdr\_pids\_tree ?

```
| +
| +struct ckpt_pids {
| + __u32 depth;
| + __s32 numbers[1];
| +} __attribute__((aligned(8)));
| +
```

This actually corresponds to `_one_ 'struct pid'` right ? Can we rename to `'struct ckpt_pid'` or `ckpt_struct_pid` ?

```
| /* task tree */
| struct ckpt_hdr_tree {
| struct ckpt_hdr h;
| - __s32 nr_tasks;
| + __u32 nr_tasks;
| } __attribute__((aligned(8)));
```

And this to, `ckpt_hdr_task_tree` ?

```
|
| -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 42de30a..f351f49 100644
--- a/kernel/checkpoint/checkpoint.c
+++ b/kernel/checkpoint/checkpoint.c
@@ -313,133 +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;
| @@ -546,31 +419,321 @@ static int build_tree(struct ckpt_ctx *ctx)
|     return 0;
| }
|
| +static int checkpoint_pids_build(struct ckpt_ctx *ctx,
| + struct flex_array *pids_arr,
| + int *nr_pids, int *nr_vpids)
| +{
| + struct pid_namespace *root_pidns;
| + struct pid *pid, *tgid, *pgrp, *session;
| + struct pid *root_session;
| + struct task_struct *task;
| + int i = 0, depth = 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();
| +
| + /*
| +  * 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) {
| +     depth += pid->level - root_pidns->level;

```

'depth' here was a bit confusing to me. We are really counting of the number of vuids. So, can you rename 'depth' to nr\_pids ?

(i.e if you find a process with pid and tgid two levels deep, it initially appeared that the depth would be 4. But the depth is still 2 and the number of vuids is 4 right ?)

```

| + 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) {
| +     depth += 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) {
| +     depth += 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) {
| +   depth += session->level - root_pidns->level;
| +   ret = flex_array_put(pids_arr, i++, session, GFP_KERNEL);
| + }
| +
| + put_pid(pid);
| + put_pid(tgid);
| + put_pid(pgrp);
| + put_pid(session);

```

We save the pid pointers in the flex\_array right ? If we put the references here, the pointers in flex\_array don't have a reference, so the pid pointer access in checkpoint\_pids\_dump() is unsafe ?

Or is it that the process tree is frozen so the pid won't go away ? If so do we need the get\_pid() and put\_pid() in this function ?

```

| +
| + if (ret < 0)
| +   break;
| + }
| +
| + *nr_pids = i;
| + *nr_vpids = depth;
| +
| + ckpt_debug("nr_pids = %d, nr_vpids = %d\n", i, depth);
| + return ret;
| +}
| +
| +static int checkpoint_pids_dump(struct ckpt_ctx *ctx,
| +  struct flex_array *pids_arr,
| +  int nr_pids, int nr_vpids)
| +{
| +  struct ckpt_hdr_pids *hh;
| +  struct ckpt_pids *h;
| +  struct pid *pid;
| +  char *buf;
| +  int root_level;
| +  int len, pos;
| +  int depth = 0;

```

Here too, using 'depth' to count nr\_vpids is a bit confusing :-)

```

| + int i, n = 0;
| + int ret;
| +
| + hh = ckpt_hdr_get_type(ctx, sizeof(*hh), CKPT_HDR_PIDS);
| + if (!hh)
| + return -ENOMEM;
| +
| + hh->nr_pids = nr_pids;
| + hh->nr_vpids = nr_vpids;
| +
| + ret = ckpt_write_obj(ctx, &hh->h);
| + ckpt_hdr_put(ctx, hh);
| + if (ret < 0)
| + return 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 (1) {
| + pid = flex_array_get(pids_arr, n);
| + len = sizeof(*h) + pid->level * sizeof(__s32);

```

Hmm. pid->level is the global level here right ? So if we checkpoint a container 2 levels deep, we don't need to save the vpids for levels 0,1. do we ? Or should we s/pid->level/(pid->level - root->level)/ (like we do for h->depth below ?

```

| +
| + /* need to flush current buffer ? */
| + if (pos + len > PAGE_SIZE || n == nr_pids)
| + break;
| +
| + h = (struct ckpt_pids *) &buf[pos];
| + h->depth = pid->level - root_level;
| + for (i = 0; i <= h->depth; i++)

```

```
| + h->numbers[i] = pid->numbers[pid->level + i].nr;
| + depth += h->depth;
| + pos += len;
| + n++;
| + }
| + rcu_read_unlock();
| +
| + /* something must have changed since last count... */
| + if (depth > nr_vpids) {
| + ret = -EBUSY;
| + break;
| + }
| +
| + ret = ckpt_kwrite(ctx, buf, pos);
| + if (ret < 0)
| + break;
```

Do we need to memset(buf, 0, sizeof(buf)) here ? Specially if we expect to fill 0s in ancestor pid namespaces (in the above example of checkpointing a container 2 levels deep, do we want to write zeros for the pid in levels 0,1) ?

Suka

---

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

---

---

Subject: Re: [PATCH 4/5] c/r: checkpoint and restart pids objects  
Posted by [Oren Laadan](#) on Sat, 05 Feb 2011 22:21:43 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

Suka,

Thanks for the review.

On 02/05/2011 04:43 PM, Sukadev Bhattiprolu wrote:

> Oren:

>

> I am still reviewing this patchset, but have a few questions/comments  
> below on this patch.

>

> | From: Oren Laadan <[orenl@cs.columbia.edu](mailto:orenl@cs.columbia.edu)>

> | Subject: [PATCH 4/5] c/r: checkpoint and restart pids objects

> |

> | 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 kernel _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 namespaces we need to report with each pid the entire list of
> | pid numbers, not only a single pid. While currently 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 handle that. So only linear pid-nesting works well and
> | not arbitrary tree.
> |
> | DISCLAIMER: this patch is big and intrusive! Here is a summary of the
> | changes that it makes:
```

```
[...]
```

```
> | diff --git a/include/linux/checkpoint_hdr.h b/include/linux/checkpoint_hdr.h
> | index 922eff0..c0a548a 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 +360,32 @@ struct ckpt_hdr_container {
> |  */
> | } __attribute__((aligned(8)));
> |
> | +/* pids array */
> | +struct ckpt_hdr_pids {
> | + struct ckpt_hdr h;
> | + __u32 nr_pids;
> | + __u32 nr_vpids;
> | +} __attribute__((aligned(8)));
> |
> | > For consistency can we call this ckpt_hdr_pids_tree ?
```

'struct ckpt\_hdr\_pids' and 'struct ckpt\_pids' are related, and do not provide information about the process tree. See also, for example, 'struct ckpt\_eventpoll\_item' and the ..\_hdr\_.. one.

```
>
> | +
> | +struct ckpt_pids {
> | + __u32 depth;
> | + __s32 numbers[1];
> | +} __attribute__((aligned(8)));
> | +
>
> This actually corresponds to _one_ 'struct pid' right ? Can we rename to
> 'struct ckpt_pid' or ckpt_struct_pid ?
```

Yes, it corresponds to a single pid object, which has `_multiple_` (rather than a single) pids numbers --> hence the name.

```
>
> | /* task tree */
> | struct ckpt_hdr_tree {
> | struct ckpt_hdr h;
> | - __s32 nr_tasks;
> | + __u32 nr_tasks;
> | } __attribute__((aligned(8)));
>
> And this to, ckpt_hdr_task_tree ?
```

Ok.

[...]

```
> | + 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();
> | +
> | + /*
> | + * 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) {
> |+     depth += pid->level - root_pidns->level;
> |+
> |+ 'depth' here was a bit confusing to me. We are really counting of the
> |+ number of vpids. So, can you rename 'depth' to nr_pids ?

```

So either 'vpids', or 'levels'. The problem with 'nr\_pids' is that it's ambiguous: could be number of pid-objects, or pid-numbers.

```

>
> (i.e if you find a process with pid and tgid two levels deep, it initially
> appeared that the depth would be 4. But the depth is still 2 and the number
> of vpids is 4 right ?)

```

Yes, it is summing the depths.

```

> |+ 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) {
> |+     depth += 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) {
> | +   depth += 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) {
> | +   depth += session->level - root_pidns->level;
> | +   ret = flex_array_put(pids_arr, i++, session, GFP_KERNEL);
> | + }
> | +
> | + put_pid(pid);
> | + put_pid(tgid);
> | + put_pid(pgrp);
> | + put_pid(session);
>
> We save the pid pointers in the flex_array right ? If we put the references
> here, the pointers in flex_array don't have a reference, so the pid pointer
> access in checkpoint_pids_dump() is unsafe ?
>
> Or is it that the process tree is frozen so the pid won't go away ? If
> so do we need the get_pid() and put_pid() in this function ?

```

We get a reference inside the rcu\_read\_lock() so that we could safely access them after we drop the lock. Then we add each (new) pid to the objhash - which will take another reference to it. Finally we drop the local reference no longer needed.

I'll a comment to make it clear.

```

> | +
> | + if (ret < 0)
> | +   break;
> | + }
> | +
> | + *nr_pids = i;
> | + *nr_vpids = depth;

```



```

> | +
> | + ckpt_debug("nr_pids = %d, nr_vpids = %d\n", i, depth);
> | + return ret;
> | +}
> | +
> | +static int checkpoint_pids_dump(struct ckpt_ctx *ctx,
> | + struct flex_array *pids_arr,
> | + int nr_pids, int nr_vpids)
> | +{
> | + struct ckpt_hdr_pids *hh;
> | + struct ckpt_pids *h;
> | + struct pid *pid;
> | + char *buf;
> | + int root_level;
> | + int len, pos;
> | + int depth = 0;
>
> Here too, using 'depth' to count nr_vpids is a bit confusing :-)

```

Ok - will change as above.

```

>
> | + int i, n = 0;
> | + int ret;
> | +
> | + hh = ckpt_hdr_get_type(ctx, sizeof(*hh), CKPT_HDR_PIDS);
> | + if (!hh)
> | + return -ENOMEM;
> | +
> | + hh->nr_pids = nr_pids;
> | + hh->nr_vpids = nr_vpids;
> | +
> | + ret = ckpt_write_obj(ctx, &hh->h);
> | + ckpt_hdr_put(ctx, hh);
> | + if (ret < 0)
> | + return 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 (1) {
> | + pid = flex_array_get(pids_arr, n);
> | + len = sizeof(*h) + pid->level * sizeof(__s32);
>
> Hmm. pid->level is the global level here right ? So if we checkpoint a
> container 2 levels deep, we don't need to save the vpids for levels 0,1.
> do we ? Or should we s/pid->level/(pid->level - root->level)/ (like
> we do for h->depth below ?

```

The latter. Good catch !

```

>
> | +
> | + /* need to flush current buffer ? */
> | + if (pos + len > PAGE_SIZE || n == nr_pids)
> | + break;
> | +
> | + h = (struct ckpt_pids *) &buf[pos];
> | + h->depth = pid->level - root_level;
> | + for (i = 0; i <= h->depth; i++)
> | + h->numbers[i] = pid->numbers[pid->level + i].nr;
> | + depth += h->depth;
> | + pos += len;
> | + n++;
> | + }
> | + rcu_read_unlock();
> | +
> | + /* something must have changed since last count... */
> | + if (depth > nr_vpids) {
> | + ret = -EBUSY;
> | + break;
> | + }
> | +
> | + ret = ckpt_kwrite(ctx, buf, pos);
> | + if (ret < 0)
> | + break;
>
> Do we need to memset(buf, 0, sizeof(buf)) here ? Specially if we expect
> to fill 0s in ancestor pid namespaces (in the above example of
> checkpointing a container 2 levels deep, do we want to write zeros for
> the pid in levels 0,1) ?

```

We shouldn't need it - assuming we fix the above as noted.

Thanks,

Oren.

---

Containers mailing list

Containers@lists.linux-foundation.org

<https://lists.linux-foundation.org/mailman/listinfo/containers>

---