Subject: Re: Checkpoint/restart (was Re: [PATCH 0/4] - v2 - Object creation with a specified id)
Posted by ebiederm on Thu, 10 Jul 2008 01:58:55 GMT
View Forum Message <> Reply to Message

Oren Laadan <orenl@cs.columbia.edu> writes:

>> Consider more intimate kernel states like:
>> a. task statistics
>> b. task start time
>> c. load average
>> d. skb state and it's data.
>> e. mount tree.
>>
>> If you think over, e.g. (b) is a bad thing. It was used to be accounted in
> jiffies, then in timespec.
>> (a) is another example of dataset which we can't predict. task statistics
> change over a time.
>> Why bother with such intimate data in user-space at all?
>> Why the hell user-space should know about it and be ABLE to modify it?
>
> Agreed.

Almost agreed. The reason we care is that the data is visible to user space in some form. So we need to save it, but hopefully not in it's internal kernel representation. If we don't care at all we should not save it.

>> My personal vision is that:
>> 1. user space must initialize checkpointing/restore state via some system
> call,
>> supply file descriptor from where data can be read/written to.
>> 2. must call the syscall asking kernel to restore/save different subsytems one
> by one.
>> 3. finalize cpt/restore state via the syscall
>> But user-space MUST NOT bother about data content. At least not about the data
> supplied by the kernel.
>> It can add additional sections if needed, e.g. about iptables state.
>
> I mostly agree with the vision that checkpoint/restart is probably best
> implemented as a black box:

I would claim an atomic unit. Roughly like a coredump is today. We know what a core dump does and we don't care how it does it.

> * First, much of the work required to restore the state of a process
> as well as the state of its resources, requires kernel interfaces that
> are lower than the ones available to user-space. Working in user-space
> will require that we design new complex interfaces for this purpose only.

Yes.

> * Second, much of the state that needs to be saved was not, is not, and
> should probably never be exported to user-space (e.g. interval socket
> buffers, t->did_exec and many more). It is only accessible to kernel
> code, so an in-kernel module (for checkpoint/restart) makes sense. It is
> that sort of internals that may (and will) change as the kernel evolves
> - precisely because it is not visible to user-space and not bound to it.

No. If the state can be inferred from user space it is visible to user
space. However there is state visible to user space like did_exec
that is not directly manipulatable by user space.

In the worst case today we can restore a checkpoint by replaying all of
the user space actions that took us to get there. That is a tedious
and slow approach.

> That said, we should still attempt to reuse existing kernel interfaces
> and mechanisms as much as possible to save - and restore - the state of
> processes, and prefer that over handcrafting special code. This is
> especially true for restart: in checkpoint one has to _capture_ the
> state by probing it in a passive manner; in contrast, in restart one
> has to actively _construct_ new resources and ensure that their state
> matches the saved state.

> For instance, it is possible to create the process tree in a container
> during restart from user-space reusing clone() (I'd argue that it's
> even better to do so from user-space). Likewise, it is possible to redo
> an open file by opening the file then using dup2() syscall to adjust
> the target fd if necessary. The two differ in that the latter allows
> to adjust the (so called) resources identifier to the desired value
> (because it is privately visible), while the former does not - it gives
> a globally visible identifier. And this is precisely why this thread
> had started: how to determine the resource identifier when requesting
> to allocate a resource (in this example, the pid).

The limiting factor to me appears to be live migration.

As I understand the concept. Live migration is where you take you
first take a snapshot of a running system, and restore that snapshot
on another system and don't start it.

Then you repeat with an incremental snapshot and you do an incremental
restore.

Until ideally the changes are small and you can afford to have the
system paused for the amount of time it takes to transfer and restore

the last incremental snapshot.

I expect a design that allows for multiple cpus to work on the
checkpoint/restore paths and that allows for incremental and
thus live migration are going to be the limiting factors in a good
interface design.

Pushing the work into the kernel with an atomic syscall style approach
so that the normal people maintaining the kernel can have visibility
of the checkpoint/restore code paths and can do some of the work
seems healthy.

>> Having all this functionality in a signle syscall we specifically CLAIM a
> black box,
>> and that no one can use this interfaces for something different from
> checkpoint/restore.
>
> True, except for what can be done (and is easier to actually that way)
> in user space; the most obvious example being clone() and setsid() -
> which are a pain to adjust after the fact. In particular, everything
> that is private in the kernel now (un-exported) should remain that way,
> unless there is an (other) compelling reason to expose it.
> (see http://www.ncl.cs.columbia.edu/publications/usenix2007_fordist.pdf)

Yes. A very good example of something that is user visible but should
not be user settable (except during restore) is the start time for a
monotonic clock. If you allow someone to set it arbitrarily it is no
longer a monotonic clock and it looses all value.

So here is one suggestion:
sys_processtree_isolate(container_init_pid);
sys_checkpoint(old_dir_fd, new_dir_fd, container_init_pid);
sys_processtree_unisolate(container_init_pid);

container_init_pid = sys_restore(dir_fd, old_container_init_pid);
sys_processtree_unisolate(container_init_pid);

Then save the different components in different files. And in the non-trivial cases
have tagged data in the files. The tags allow us to have different data types
easily.

For data that already have or should have an interface for
persistence. Filesystems being the primary example we don't handle
this way. Instead we use the already existing techniques to
backup/snapshot the data and then to restore the data.

Isolation should be used instead of freezing if we can, because

isolation is a much cheaper concept, and it allows for multi-machine synchronized checkpoints. On the big scale if I unplug a machines network cable (isolating it) then I don't care when between the time I isolate the machine and the time I plug the cable back in. It all looks the same to the outside world. But just the comunications of the machine were frozen. Likewise for processes. SIGSTOP is good but I suspect it may be excessive, so denies us optimization opportunities (at least in the interface).

So in summary we want an interface that allows for.
- Existing persistent kernel state to using the preexisting mechanisms.
- The persistence code to be looked after by the people working on the kernel subsystem.
- Migration between kernel versions, (possibly with user space intervention).
- Live migration
- Minimal intervention of the processes being checkpointed.
- Denying persistence to applications that need it.
- Checkpoints coordinated between multiple containers or real machines.
- Documented in terms of isolation rather then freezing, as it is the isolation that is the important property.
- Restricted access to user space visible but immutable state.
- Ideally sufficiently general that debuggers can take advantage of the checkpoints.

Eric

_____