System State Changes

Some users are really reluctant to reboot a system. This brings the need to provide more livepatches and maintain some compatibility between them.

Maintaining more livepatches is much easier with cumulative livepatches. Each new livepatch completely replaces any older one. It can keep, add, and even remove fixes. And it is typically safe to replace any version of the livepatch with any other one thanks to the atomic replace feature.

The problems might come with shadow variables and callbacks. They might change the system behavior or state so that it is no longer safe to go back and use an older livepatch or the original kernel code. Also any new livepatch must be able to detect what changes have already been done by the already installed livepatches.

This is where the livepatch system state tracking gets useful. It allows to:

- store data needed to manipulate and restore the system state
- define compatibility between livepatches using a change id and version

1. Livepatch system state API

The state of the system might get modified either by several livepatch callbacks or by the newly used code. Also it must be possible to find changes done by already installed livepatches.

Each modified state is described by struct klp_state, see include/linux/livepatch.h.

Each livepatch defines an array of struct klp_states. They mention all states that the livepatch modifies.

The livepatch author must define the following two fields for each struct klp_state:

- *id*
  - Non-zero number used to identify the affected system state.
- *version*
  - Number describing the variant of the system state change that is supported by the given livepatch.

The state can be manipulated using two functions:

- klp_get_state()
  - Get struct klp_state associated with the given livepatch and state id.
- klp_get_prev_state()
  - Get struct klp_state associated with the given feature id and already installed livepatches.

2. Livepatch compatibility

The system state version is used to prevent loading incompatible livepatches. The check is done when the livepatch is enabled. The rules are:

- Any completely new system state modification is allowed.
- System state modifications with the same or higher version are allowed for already modified system states.
- Cumulative livepatches must handle all system state modifications from already installed livepatches.
- Non-cumulative livepatches are allowed to touch already modified system states.

3. Supported scenarios

Livepatches have their life-cycle and the same is true for the system state changes. Every compatible livepatch has to support the following scenarios:

- Modify the system state when the livepatch gets enabled and the state has not been already modified by a livepatches that are being replaced.
- Take over or update the system state modification when is has already been done by a livepatch that is being replaced.
- Restore the original state when the livepatch is disabled.
- Restore the previous state when the transition is reverted. It might be the original system state or the state modification done by livepatches that were being replaced.
- Remove any already made changes when error occurs and the livepatch cannot get enabled.

4. Expected usage

System states are usually modified by livepatch callbacks. The expected role of each callback is as follows:

pre_patch()

- Allocate *state->data* when necessary. The allocation might fail and *pre_patch()* is the only callback that could stop loading of the livepatch. The allocation is not needed when the data are already provided by previously installed livepatches.
- Do any other preparatory action that is needed by the new code even before the transition gets finished. For example, initialize *state->data*.

  The system state itself is typically modified in *post_patch()* when the entire system is able to handle it.
- Clean up its own mess in case of error. It might be done by a custom code or by calling *post_unpatch()* explicitly.

post_patch()

- Copy *state->data* from the previous livepatch when they are compatible.
- Do the actual system state modification. Eventually allow the new code to use it.
- Make sure that *state->data* has all necessary information.
- Free *state->data* from replaces livepatches when they are not longer needed.

pre_unpatch()

- Prevent the code, added by the livepatch, relying on the system state change.
- Revert the system state modification..

post_unpatch()

- Distinguish transition reverse and livepatch disabling by checking *klp_get_prev_state()*.
- In case of transition reverse, restore the previous system state. It might mean doing nothing.
- Remove any not longer needed setting or data.

> **Note**
>
> *pre_unpatch()* typically does symmetric operations to *post_patch()*. Except that it is called only when the livepatch is being disabled. Therefore it does not need to care about any previously installed livepatch.
>
> *post_unpatch()* typically does symmetric operations to *pre_patch()*. It might be called also during the transition reverse. Therefore it has to handle the state of the previously installed livepatches.