Atomic Replace & Cumulative Patches

There might be dependencies between livepatches. If multiple patches need to do different changes to the same function(s) then we need to define an order in which the patches will be installed. And function implementations from any newer livepatch must be done on top of the older ones.

This might become a maintenance nightmare. Especially when more patches modified the same function in different ways.

An elegant solution comes with the feature called "Atomic Replace". It allows creation of so called "Cumulative Patches". They include all wanted changes from all older livepatches and completely replace them in one transition.

Usage

The atomic replace can be enabled by setting "replace" flag in struct klp_patch, for example:

```
static struct klp_patch patch = {
        .mod = THIS_MODULE,
        .objs = objs,
        .replace = true,
};
```

All processes are then migrated to use the code only from the new patch. Once the transition is finished, all older patches are automatically disabled.

Ftrace handlers are transparently removed from functions that are no longer modified by the new cumulative patch.

As a result, the livepatch authors might maintain sources only for one cumulative patch. It helps to keep the patch consistent while adding or removing various fixes or features.

Users could keep only the last patch installed on the system after the transition to has finished. It helps to clearly see what code is actually in use. Also the livepatch might then be seen as a "normal" module that modifies the kernel behavior. The only difference is that it can be updated at runtime without breaking its functionality.

Features

The atomic replace allows:

- Atomically revert some functions in a previous patch while upgrading other functions.
- Remove eventual performance impact caused by core redirection for functions that are no longer patched.
- Decrease user confusion about dependencies between livepatches.

Limitations:

- Once the operation finishes, there is no straightforward way to reverse it and restore the replaced patches atomically.

  A good practice is to set .replace flag in any released livepatch. Then re-adding an older livepatch is equivalent to downgrading to that patch. This is safe as long as the livepatches do _not_ do extra modifications in (un)patching callbacks or in the module_init() or module_exit() functions, see below.

  Also note that the replaced patch can be removed and loaded again only when the transition was not forced.

- Only the (un)patching callbacks from the _new_ cumulative livepatch are executed. Any callbacks from the replaced patches are ignored.

  In other words, the cumulative patch is responsible for doing any actions that are necessary to properly replace any older patch.

  As a result, it might be dangerous to replace newer cumulative patches by older ones. The old livepatches might not provide the necessary callbacks.

  This might be seen as a limitation in some scenarios. But it makes life easier in many others. Only the new cumulative livepatch knows what fixes/features are added/removed and what special actions are necessary for a smooth transition.

  In any case, it would be a nightmare to think about the order of the various callbacks and their interactions if the callbacks from all enabled patches were called.

- There is no special handling of shadow variables. Livepatch authors must create their own rules how to pass them from one cumulative patch to the other. Especially that they should not blindly remove them in module_exit() functions.

  A good practice might be to remove shadow variables in the post-unpatch callback. It is called only when the

livepatch is properly disabled.