Livepatch

This document outlines basic information about kernel livepatching.

1. Motivation

There are many situations where users are reluctant to reboot a system. It may be because their system is performing complex scientific computations or under heavy load during peak usage. In addition to keeping systems up and running, users want to also have a stable and secure system. Livepatching gives users both by allowing for function calls to be redirected; thus, fixing critical functions without a system reboot.

2. Kprobes, Ftrace, Livepatching

There are multiple mechanisms in the Linux kernel that are directly related to redirection of code execution; namely: kernel probes, function tracing, and livepatching:

- The kernel probes are the most generic. The code can be redirected by putting a breakpoint instruction instead of any instruction.
- The function tracer calls the code from a predefined location that is close to the function entry point. This location is generated by the compiler using the '-pg' gcc option.
- Livepatching typically needs to redirect the code at the very beginning of the function entry before the function parameters or the stack are in any way modified.

All three approaches need to modify the existing code at runtime. Therefore they need to be aware of each other and not step over each other's toes. Most of these problems are solved by using the dynamic ftrace framework as a base. A Kprobe is registered as a ftrace handler when the function entry is probed, see CONFIG_KPROBES_ON_FTRACE. Also an alternative function from a live patch is called with the help of a custom ftrace handler. But there are some limitations, see below.

3. Consistency model

Functions are there for a reason. They take some input parameters, get or release locks, read, process, and even write some data in a defined way, have return values. In other words, each function has a defined semantic.

Many fixes do not change the semantic of the modified functions. For example, they add a NULL pointer or a boundary check, fix a race by adding a missing memory barrier, or add some locking around a critical section. Most of these changes are self contained and the function presents itself the same way to the rest of the system. In this case, the functions might be updated independently one by one.

But there are more complex fixes. For example, a patch might change ordering of locking in multiple functions at the same time. Or a patch might exchange meaning of some temporary structures and update all the relevant functions. In this case, the affected unit (thread, whole kernel) need to start using all new versions of the functions at the same time. Also the switch must happen only when it is safe to do so, e.g. when the affected locks are released or no data are stored in the modified structures at the moment.

The theory about how to apply functions a safe way is rather complex. The aim is to define a so-called consistency model. It attempts to define conditions when the new implementation could be used so that the system stays consistent.

Livepatch has a consistency model which is a hybrid of kGraft and kpatch: it uses kGraft's per-task consistency and syscall barrier switching combined with kpatch's stack trace switching. There are also a number of fallback options which make it quite flexible.

Patches are applied on a per-task basis, when the task is deemed safe to switch over. When a patch is enabled, livepatch enters into a transition state where tasks are converging to the patched state. Usually this transition state can complete in a few seconds. The

same sequence occurs when a patch is disabled, except the tasks converge from the patched state to the unpatched state.

An interrupt handler inherits the patched state of the task it interrupts. The same is true for forked tasks: the child inherits the patched state of the parent.

Livepatch uses several complementary approaches to determine when it's safe to patch tasks:

1. The first and most effective approach is stack checking of sleeping tasks. If no affected functions are on the stack of a given task, the task is patched. In most cases this will patch most or all of the tasks on the first try. Otherwise it'll keep trying periodically. This option is only available if the architecture has reliable stacks (HAVE_RELIABLE_STACKTRACE).
2. The second approach, if needed, is kernel exit switching. A task is switched when it returns to user space from a system call, a user space IRQ, or a signal. It's useful in the following cases:
    a. Patching I/O-bound user tasks which are sleeping on an affected function. In this case you have to send SIGSTOP and SIGCONT to force it to exit the kernel and be patched.
    b. Patching CPU-bound user tasks. If the task is highly CPU-bound then it will get patched the next time it gets interrupted by an IRQ.
3. For idle "swapper" tasks, since they don't ever exit the kernel, they instead have a klp_update_patch_state() call in the idle loop which allows them to be patched before the CPU enters the idle state.

    (Note there's not yet such an approach for kthreads.)

Architectures which don't have HAVE_RELIABLE_STACKTRACE solely rely on the second approach. It's highly likely that some tasks may still be running with an old version of the function, until that function returns. In this case you would have to signal the tasks. This especially applies to kthreads. They may not be woken up and would need to be forced. See below for more information.

Unless we can come up with another way to patch kthreads, architectures without HAVE_RELIABLE_STACKTRACE are not considered fully supported by the kernel livepatching.

The /sys/kernel/livepatch/<patch>/transition file shows whether a patch is in transition. Only a single patch can be in transition at a given time. A patch can remain in transition indefinitely, if any of the tasks are stuck in the initial patch state.

A transition can be reversed and effectively canceled by writing the opposite value to the /sys/kernel/livepatch/<patch>/enabled file while the transition is in progress. Then all the tasks will attempt to converge back to the original patch state.

There's also a /proc/<pid>/patch_state file which can be used to determine which tasks are blocking completion of a patching operation. If a patch is in transition, this file shows 0 to indicate the task is unpatched and 1 to indicate it's patched. Otherwise, if no patch is in transition, it shows -1. Any tasks which are blocking the transition can be signaled with SIGSTOP and SIGCONT to force them to change their patched state. This may be harmful to the system though. Sending a fake signal to all remaining blocking tasks is a better alternative. No proper signal is actually delivered (there is no data in signal pending structures). Tasks are interrupted or woken up, and forced to change their patched state. The fake signal is automatically sent every 15 seconds.

Administrator can also affect a transition through /sys/kernel/livepatch/<patch>/force attribute. Writing 1 there clears TIF_PATCH_PENDING flag of all tasks and thus forces the tasks to the patched state. Important note! The force attribute is intended for cases when the transition gets stuck for a long time because of a blocking task. Administrator is expected to collect all necessary data (namely stack traces of such blocking tasks) and request a clearance from a patch distributor to force the transition. Unauthorized usage may cause harm to the system. It depends on the nature of the patch, which functions are (un)patched, and which functions the blocking tasks are sleeping in (/proc/<pid>/stack may help here). Removal (rmmod) of patch modules is permanently disabled when the force feature is used. It cannot be guaranteed there is no task sleeping in such module. It implies unbounded reference count if a patch module is disabled and enabled in a loop.

Moreover, the usage of force may also affect future applications of live patches and cause even more harm to the system. Administrator should first consider to simply cancel a transition (see above). If force is used, reboot should be planned and no more live patches applied.

3.1 Adding consistency model support to new architectures

For adding consistency model support to new architectures, there are a few options:

1. Add CONFIG_HAVE_RELIABLE_STACKTRACE. This means porting objtool, and for non-DWARF unwinders, also making sure there's a way for the stack tracing code to detect interrupts on the stack.
2. Alternatively, ensure that every kthread has a call to klp_update_patch_state() in a safe location. Kthreads are typically in an infinite loop which does some action repeatedly. The safe location to switch the kthread's patch state would be at a designated point in the loop where there are no locks taken and all data structures are in a well-defined state.

    The location is clear when using workqueues or the kthread worker API. These kthreads process independent actions in a generic loop.

    It's much more complicated with kthreads which have a custom loop. There the safe location must be carefully selected on a case-by-case basis.

    In that case, arches without HAVE_RELIABLE_STACKTRACE would still be able to use the non-stack-checking parts of the consistency model:

a. patching user tasks when they cross the kernel/user space boundary; and
b. patching kthreads and idle tasks at their designated patch points.

This option isn't as good as option 1 because it requires signaling user tasks and waking kthreads to patch them. But it could still be a good backup option for those architectures which don't have reliable stack traces yet.

4. Livepatch module

Livepatches are distributed using kernel modules, see samples/livepatch/livepatch-sample.c.

The module includes a new implementation of functions that we want to replace. In addition, it defines some structures describing the relation between the original and the new implementation. Then there is code that makes the kernel start using the new code when the livepatch module is loaded. Also there is code that cleans up before the livepatch module is removed. All this is explained in more details in the next sections.

4.1. New functions

New versions of functions are typically just copied from the original sources. A good practice is to add a prefix to the names so that they can be distinguished from the original ones, e.g. in a backtrace. Also they can be declared as static because they are not called directly and do not need the global visibility.

The patch contains only functions that are really modified. But they might want to access functions or data from the original source file that may only be locally accessible. This can be solved by a special relocation section in the generated livepatch module, see Documentation/livepatch/module-elf-format.rst for more details.

4.2. Metadata

The patch is described by several structures that split the information into three levels:

- struct klp_func is defined for each patched function. It describes the relation between the original and the new implementation of a particular function.

  The structure includes the name, as a string, of the original function. The function address is found via kallsyms at runtime.

  Then it includes the address of the new function. It is defined directly by assigning the function pointer. Note that the new function is typically defined in the same source file.

  As an optional parameter, the symbol position in the kallsyms database can be used to disambiguate functions of the same name. This is not the absolute position in the database, but rather the order it has been found only for a particular object (vmlinux or a kernel module). Note that kallsyms allows for searching symbols according to the object name.

- struct klp_object defines an array of patched functions (struct klp_func) in the same object. Where the object is either vmlinux (NULL) or a module name.

  The structure helps to group and handle functions for each object together. Note that patched modules might be loaded later than the patch itself and the relevant functions might be patched only when they are available.

- struct klp_patch defines an array of patched objects (struct klp_object).

  This structure handles all patched functions consistently and eventually, synchronously. The whole patch is applied only when all patched symbols are found. The only exception are symbols from objects (kernel modules) that have not been loaded yet.

  For more details on how the patch is applied on a per-task basis, see the "Consistency model" section.

5. Livepatch life-cycle

Livepatching can be described by five basic operations: loading, enabling, replacing, disabling, removing.

Where the replacing and the disabling operations are mutually exclusive. They have the same result for the given patch but not for the system.

5.1. Loading

The only reasonable way is to enable the patch when the livepatch kernel module is being loaded. For this, klp_enable_patch() has to be called in the module_init() callback. There are two main reasons:

First, only the module has an easy access to the related struct klp_patch.

Second, the error code might be used to refuse loading the module when the patch cannot get enabled.

5.2. Enabling

The livepatch gets enabled by calling klp_enable_patch() from the module_init() callback. The system will start using the new implementation of the patched functions at this stage.

First, the addresses of the patched functions are found according to their names. The special relocations, mentioned in the section "New functions", are applied. The relevant entries are created under /sys/kernel/livepatch/<name>. The patch is rejected when any above operation fails.

Second, livepatch enters into a transition state where tasks are converging to the patched state. If an original function is patched for the first time, a function specific struct klp_ops is created and an universal ftrace handler is registered[1]. This stage is indicated by a value of '1' in /sys/kernel/livepatch/<name>/transition. For more information about this process, see the "Consistency model" section.

Finally, once all tasks have been patched, the 'transition' value changes to '0'.

[1] Note that functions might be patched multiple times. The ftrace handler is registered only once for a given function. Further patches just add an entry to the list (see field *func_stack*) of the struct klp_ops. The right implementation is selected by the ftrace handler, see the "Consistency model" section.

That said, it is highly recommended to use cumulative livepatches because they help keeping the consistency of all changes. In this case, functions might be patched two times only during the transition period.

5.3. Replacing

All enabled patches might get replaced by a cumulative patch that has the .replace flag set.

Once the new patch is enabled and the 'transition' finishes then all the functions (struct klp_func) associated with the replaced patches are removed from the corresponding struct klp_ops. Also the ftrace handler is unregistered and the struct klp_ops is freed when the related function is not modified by the new patch and func_stack list becomes empty.

See Documentation/livepatch/cumulative-patches.rst for more details.

5.4. Disabling

Enabled patches might get disabled by writing '0' to /sys/kernel/livepatch/<name>/enabled.

First, livepatch enters into a transition state where tasks are converging to the unpatched state. The system starts using either the code from the previously enabled patch or even the original one. This stage is indicated by a value of '1' in /sys/kernel/livepatch/<name>/transition. For more information about this process, see the "Consistency model" section.

Second, once all tasks have been unpatched, the 'transition' value changes to '0'. All the functions (struct klp_func) associated with the to-be-disabled patch are removed from the corresponding struct klp_ops. The ftrace handler is unregistered and the struct klp_ops is freed when the func_stack list becomes empty.

Third, the sysfs interface is destroyed.

5.5. Removing

Module removal is only safe when there are no users of functions provided by the module. This is the reason why the force feature permanently disables the removal. Only when the system is successfully transitioned to a new patch state (patched/unpatched) without being forced it is guaranteed that no task sleeps or runs in the old code.

6. Sysfs

Information about the registered patches can be found under /sys/kernel/livepatch. The patches could be enabled and disabled by writing there.

/sys/kernel/livepatch/<patch>/force attributes allow administrator to affect a patching operation.

See Documentation/ABI/testing/sysfs-kernel-livepatch for more details.

7. Limitations

The current Livepatch implementation has several limitations:

- Only functions that can be traced could be patched.

  Livepatch is based on the dynamic ftrace. In particular, functions implementing ftrace or the livepatch ftrace handler could not be patched. Otherwise, the code would end up in an infinite loop. A potential mistake is prevented by marking the problematic functions by "notrace".

- Livepatch works reliably only when the dynamic ftrace is located at the very beginning of the function.

  The function need to be redirected before the stack or the function parameters are modified in any way. For example, livepatch requires using -fentry gcc compiler option on x86_64.

  One exception is the PPC port. It uses relative addressing and TOC. Each function has to handle TOC and save LR before it could call the ftrace handler. This operation has to be reverted on return. Fortunately, the generic ftrace

code has the same problem and all this is handled on the ftrace level.

- Kretprobes using the ftrace framework conflict with the patched functions.

  Both kretprobes and livepatches use a ftrace handler that modifies the return address. The first user wins. Either the probe or the patch is rejected when the handler is already in use by the other.

- Kprobes in the original function are ignored when the code is redirected to the new implementation.

  There is a work in progress to add warnings about this situation.