

Challenges in Preserving Virtual Networking State and dmabuf across Kexec Live Updates

Zhu Yanjun Sep. 6, 2026



Agenda

- Background
- RXE liveupdate architecture(Complete)
- netkit liveupdate architecture(Complete)
- Dmabuf liveupdate architecture(WIP)
- Problems
 - Dependency Ordering, State Preservation and Recovery
Coordination, Hardware State Synchronization
- Discussion



Background

The architecture consists of four main building blocks:

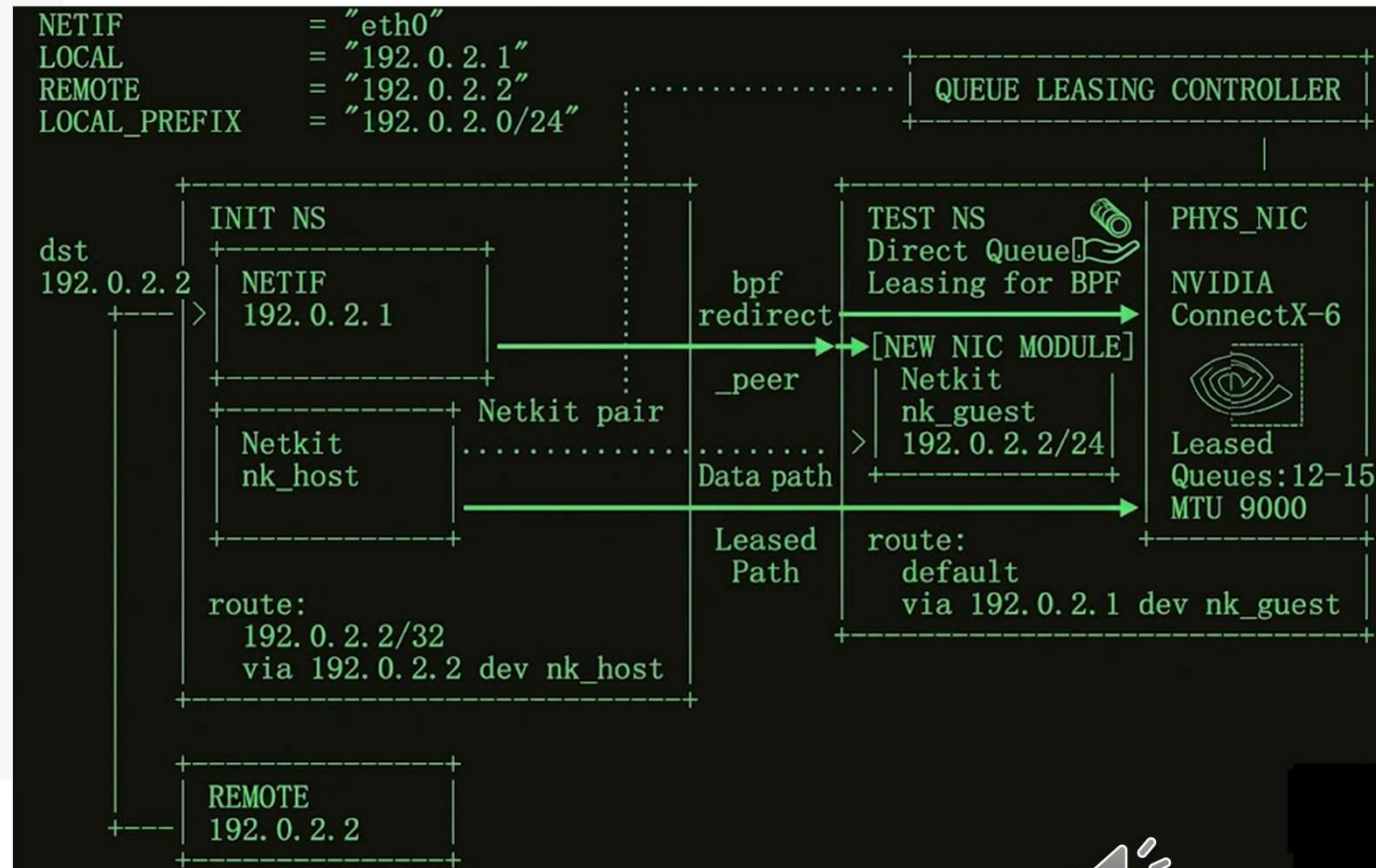
INIT NS (Host/Root Namespace): Contains the root network interface NETIF (192.0.2.1) and the host side of the Netkit pair (nk_host).

TEST NS (Target Container/Test Namespace): Represents the isolated container environment running a [NEW NIC MODULE] with the guest side of the Netkit pair (nk_guest, 192.0.2.2/24).

QUEUE LEASING CONTROLLER: A control-plane component that dynamically manages, isolates, and allocates specific hardware queues from the physical NIC to target namespaces.

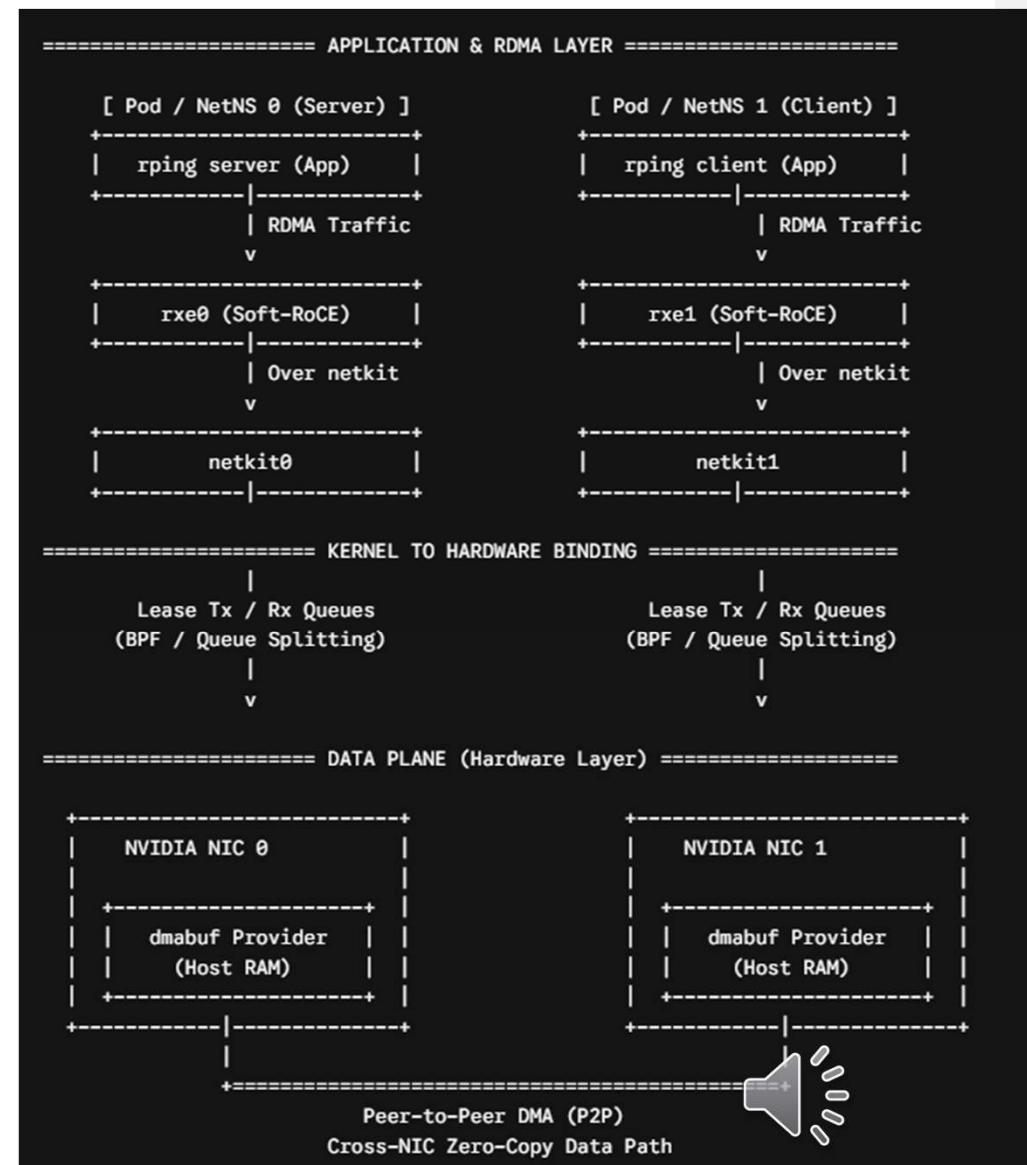
PHYS_NIC (Physical Network Interface): Represented here by an NVIDIA ConnectX-6 NIC, which supports hardware partitioning.

Network Topology



Architecture Introduction

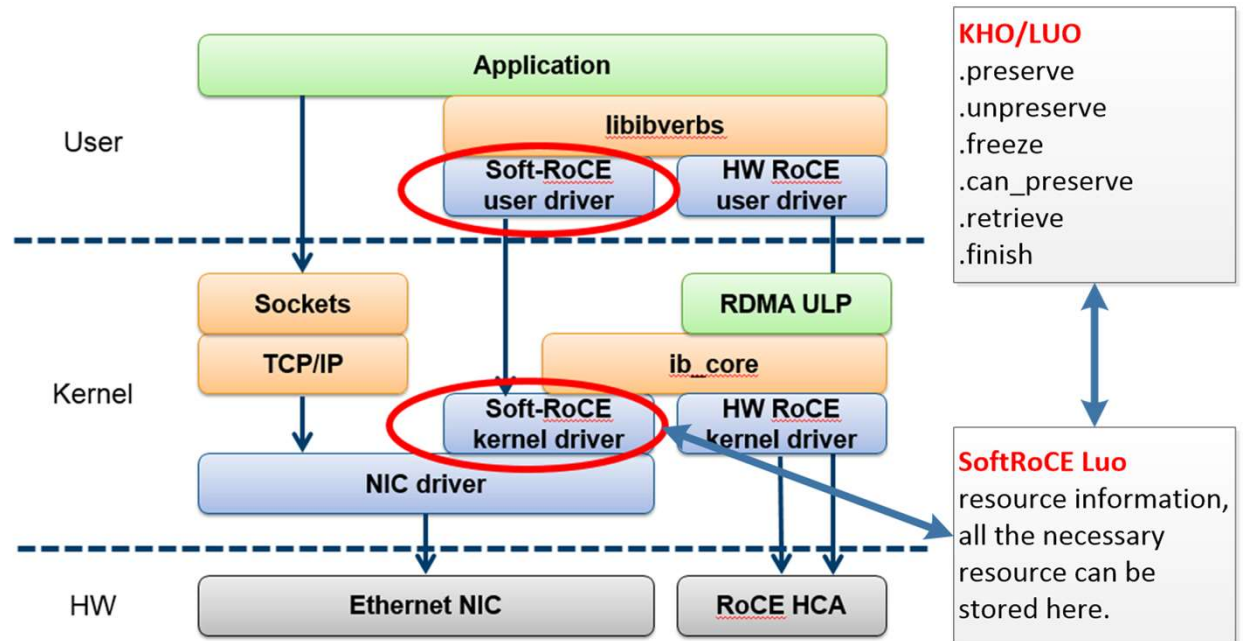
This diagram illustrates our end-to-end logical data path. On the left, traffic originates from the **rping server (App)** inside **NetNS 0**. It passes through the Soft-RoCE driver **rxex0**, enters the **netkit0** interface, and leverages BPF-based Queue Leasing to bypass standard bottlenecks directly to **NVIDIA NIC 0**. At the hardware layer, data flows across NICs via dmabuf-backed Peer-to-Peer DMA for zero-copy transfer to **NIC 1**, then travels up through **netkit1** and **rxex1** to reach the client app in **NetNS 1**. Our goal is to evaluate and achieve state preservation across Live Update / KHO (Kexec Handover). Specifically, when a kernel reboot or live update occurs, we want this entire topology—spanning the RDMA stack, netkit bindings, and devmem/dmabuf hardware data paths—to remain intact and functional post-reboot. To implement this, we break the topology down into three distinct layers: **RDMA**, **netkit**, and **devmem/dmabuf**, addressing state retention and handover for each independently.



RXE Live update Architecture

How the **Soft-RoCE (rx)** module integrates with the **Live Update** framework? When the rx kernel driver initializes, it registers itself directly with the Live Update system. As RDMA links and resources are instantiated, rx maintains and tracks all necessary state contexts inside internal xarray data structures.

The core state handover is driven by the KHO and LUO framework lifecycle hooks—specifically **.preserve**, **.unpreserve**, **.freeze**, **.canpreserve**, **.retrieve**, and **.finish**. Just before the live_update execution, LUO triggers the preservation sequence, dumping the xarray resource metadata straight into KHO-reserved **CMA** buffers to ensure memory persistence across the kexec boundary. Upon boot, the new kernel reads back these rx resources directly from the **CMA** memory region via the **.retrieve** and **.finish** stages, automatically reconstructing the virtual RDMA networking topology without breaking state continuity. The complete implementation and RFC patches for this workflow are available on GitHub for community review.



RXE live update test result

As shown in this log, we have successfully validated the Soft-RoCE/RXE driver under the Live Update framework using `kselftest`. During this run, the full suite of LUO callback functions—including `.can_preserve`, `.preserve`, and `.unpreserve`—were invoked and executed as expected. As you can see from the console output, the test case `liveupdate_device.preserve_rxe_fd` completed with a 100% pass rate and zero errors, confirming that the state preservation and restoration logic for RXE during live updates is fully functional. The complete selftest implementation is available on GitHub under the `kselftest` directory for anyone who wants to review or replicate the results.

```
++ tools/testing/selftests/liveupdate/luo_test_rxe
TAP version 13
1..1
# Starting 1 tests from 1 test cases.
# RUN liveupdate_device.preserve_rxe_fd ...
insmod /lib/modules/7.3.0-rc1-rxe-luo-10+/kernel/net/ipv4/udp_tunnel.ko
insmod /lib/modules/7.3.0-rc1-rxe-luo-10+/kernel/net/ipv6/ip6_udp_tunnel.ko
insmod /lib/modules/7.3.0-rc1-rxe-luo-10+/kernel/drivers/infiniband/core/ib_core.ko
insmod /lib/modules/7.3.0-rc1-rxe-luo-10+/kernel/drivers/infiniband/sw/rxe/rdma_rxe.ko
link rxe0/1 state ACTIVE physical_state LINK_UP netdev lo
rmmod rdma_rxe
rmmod udp_tunnel
rmmod ip6_udp_tunnel
rmmod ib_core
# OK liveupdate_device.preserve_rxe_fd
ok 1 liveupdate_device.preserve_rxe_fd
# PASSED: 1 / 1 tests passed.
# Totals: pass:1 fail:0 xfail:0 xpass:0 skip:0 error:0
++ set +x
```



Netkit Live update Architecture

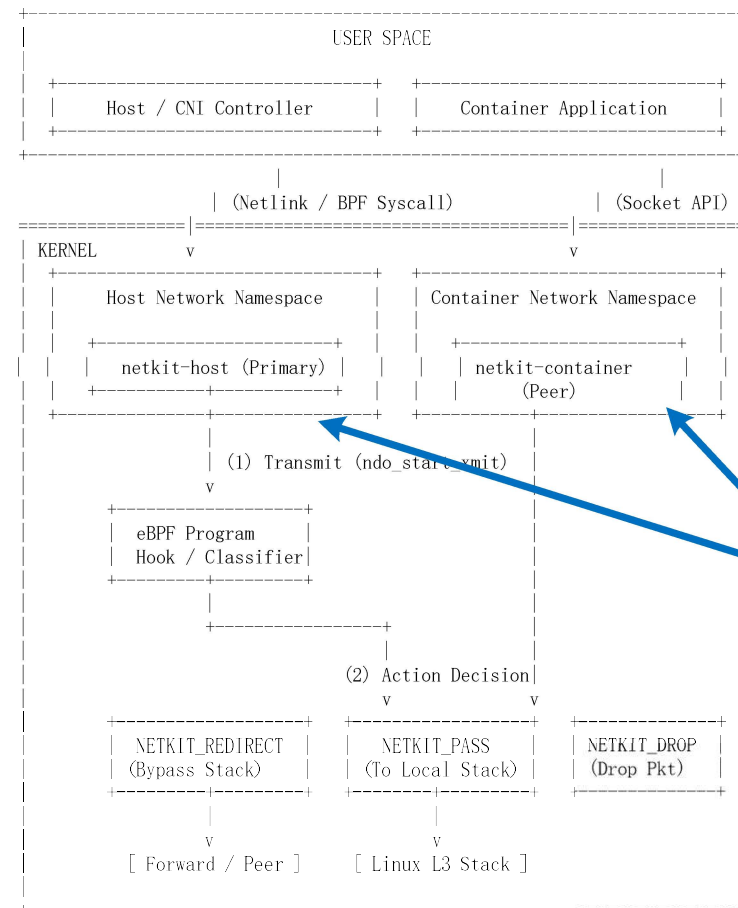
Let us walk through how **netkit** integrates with the Live Update framework. During initialization, the **netkit** kernel driver registers directly with the Live Update system. As ip links and resources are instantiated, **netkit** tracks their state inside internal **xarray** data structures.

The core state handover relies on KHO and LUO lifecycle hooks—specifically **.can_preserve**, **.preserve**, **.unpreserve**, **.freeze**, **.retrieve**, and **.finish**.

Right before a live update, LUO triggers the preservation sequence, dumping the **xarray** metadata into KHO-reserved CMA memory buffers. This ensures complete memory persistence across the **kexec** boundary.

Once the new kernel boots, it reads the **netkit** resources back from CMA memory using the **.retrieve** and **.finish** stages, seamlessly restoring the virtual network topology without dropping state.

The complete implementation and RFC patches are up on GitHub for community review.



KHO/LUO
.preserve
.unpreserve
.freeze
.can_preserve
.retrieve
.finish

Netkit Luo resource information, all the necessary resource can be stored here.



Netkit live update test result

As shown in this log, we have successfully validated the netkit driver under the Live Update framework using kselftest. During this run, the full suite of LUO callback functions—including `.can_preserve`, `.preserve`, and `.unpreserve`—were invoked and executed as expected. As you can see from the console output, the test case `liveupdate_device.preserve_netkit_fd` completed with a 100% pass rate and zero errors, confirming that the state preservation and restoration logic for netkit during live updates is fully functional. The complete selftest implementation is available on GitHub under the kselftest directory for anyone who wants to review or replicate the results.

```
[root@localhost linux]# ./tools/testing/selftests/liveupdate/luo_test_netkit
TAP version 13
1..1
# Starting 1 tests from 1 test cases.
# RUN      liveupdate_device.preserve_netkit_fd ...
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enpls0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEFAULT group default qlen 1000
   link/ether 52:54:00:47:8a:13 brd ff:ff:ff:ff:ff:ff
   altname enx525400478a13
5: netkit1@netkit0: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
   link/ether 7a:83:8d:be:db:54 brd ff:ff:ff:ff:ff:ff
6: netkit0@netkit1: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
   link/ether aa:12:c7:57:52:63 brd ff:ff:ff:ff:ff:ff
#      OK      liveupdate_device.preserve_netkit_fd
ok 1 liveupdate_device.preserve_netkit_fd
# PASSED: 1 / 1 tests passed.
# Totals: pass:1 fail:0 xfail:0 xpass:0 skip:0 error:0
```



Questions

1. Based on our LUO/KHO integration experience with `softroce/rxe` and `netkit`, the current approach requires custom development for each driver. Specifically, every driver must implement set callback ops including `.can_preserve`, `.preserve`, `.unpreserve`, `.freeze`, `.retrieve`, and `.finish`. Can we extract these core features into a generic framework to avoid redundant engineering across drivers? Rather than forcing drivers to manage complex callback lifecycles, can we implement a standalone state management layer? Any driver or kernel subsystem could simply register with this layer, stash its state before kexec, and retrieve it upon boot. Is this architectural simplification feasible?

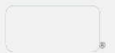
Traditional

```
/* Drivers must define custom ops, serialize structs, and build DTB nodes */
static struct kho_device_ops rxe_kho_ops = {
    .serialize    = rxe_kho_serialize,
    .deserialize  = rxe_kho_deserialize,
};
```

Proposed

```
/* Save Phase (Pre-Kexec / Freeze) */
kho_state_put("rxe_qp_table", &qp_data, sizeof(qp_data), KHO_FLAG_MEM_PIN);

/* Restore Phase (Post-Kexec / Probe) */
qp_data = kho_state_get("rxe_qp_table", &size);
if (qp_data) {
    /* Restore QP state directly */
}
```



Dmabuf live update architecture Development

1. Currently dmabuf liveupdate is based on pci liveupdate v9, VFIOv5 and iommu phase1.

The links are as below:

<https://github.com/dmatlack/linux/tree/liveupdate/pci/base/v9>

<https://lore.kernel.org/kvm/20260714151505.3466855-1-vipinsh@google.com/>

<https://lore.kernel.org/linux-iommu/20260921004834.2601285-1-skhawaja@google.com/>

The above are WIP. The dmabuf liveupdate is based on the above. It is also WIP.

2. Pci liveupdate status: Currently PCI Core Live Update is ready. It enables PCI devices to preserve state and maintain uninterrupted DMA transactions across kexec kernel upgrades via KHO. It automatically handles upstream bridges, bus numbers, and ACS configurations. The series has passed tests on both QEMU and bare-metal, and will be merged through the liveupdate.git tree.
3. VFIO liveupdate status: PCIe function (non-VF) is preserved across Live Update without changing its BDF, and this has been validated on both bare-metal hardware and QEMU.
4. IOMMU liveupdate status: Keep DMA translations and IOMMU context active across a kexec reboot to achieve zero-downtime updates for VM-assigned devices. Supports only file-backed DMA mappings (sealed memfd); devices using ATS (Address Translation Services) are currently rejected.



Dmabuf live update preparations

1. Enforcing Quiescence and Fence Cleanup

- 1) Clear Pending Fences: Unfinished dma_fence objects (such as active rendering tasks on a GPU) may still be attached to the dma-buf. During the upgrade preparation phase, all existing fences transition should be in the Signaled state. Transferring a dma-buf with pending fences across kernels is strictly prohibited.()
- 2) Suspend Writes from Non-Preserved Devices: Pause access to the dma-buf from all hardware modules that are not participating in the Live Update. This avoids undefined write contention at the exact moment the old kernel exits.

Question: To clear pending fences in a dma_buf, PCI and IOMMU must stay fully active. How do we ensure PCI and IOMMU are not disabled or frozen before fence cleanup completes during Live Update? What mechanism handles this?

2. Physical Memory De-fragmentation

- 1) Consolidate Scattered Pages (Contiguous Chunking): If the physical pages backing the dma-buf are excessively fragmented (spread across thousands of distinct pages), the KHO serialization structures will become abnormally large, increasing the parsing overhead during early boot.
- 2) Preparation Requirements: When allocating a dma-buf that supports Live Update, try to use CMA (Contiguous Memory Allocator) or Huge Pages to keep the number of dmabuf_kho_chunk instances under control.

Question: Which is better for physical memory defragmentation: CMA or Huge Pages?



Dmabuf live update preparations – Cont.

1. Driver and Userspace Protocol Alignment

- 1) Exporter Driver Adaptation: Drivers exporting the dma-buf (such as DRM or System Heap) must implement the KHO Restore callback functions, allowing them to reconstruct the struct dma_buf directly from KHO metadata without reallocating physical memory.
- 2) Userspace File Descriptor (FD) Tracking: Userspace management components (such as the LUO Client) must record in advance which FDs correspond to which dma-buf objects, and maintain the capability to regain control of these FDs after the upgrade using pidfd_getfd() or Unix Sockets.

Questions: Hardware & DMA Out-of-Sync: Reconstructing struct dma_buf via KHO only restores kernel metadata. If the buffer is active in DMA (e.g., GPU, SmartNIC/RDMA), hardware state like IOMMU page tables, ring buffers, and device contexts can become stale. Failing to remap device state after kernel restore risks IOMMU faults or memory corruption. How to solve this problem?

Dmabuf Steps

1) Outgoing Phase (Preparation)

1. Userspace uses FLB to mark the dma-buf file descriptor (FD).
2. The exporter retrieves the underlying physical pages (sg_table) and packages them into a dmabuf_kho_chunk. Simultaneously, the IOMMU reads the current virtual address (IOVA) and packages it into a dmabuf_kho_attachment.
3. Metadata is registered into the KHO (Kexec HandOver) region.

Question: During dmabuf information collection, PCI, VFIO, and IOMMU must be fully operational; however, these devices operate asynchronously. Is there a comprehensive mechanism in Live Update to ensure that dmabuf is preserved first, followed by PCI, VFIO, and IOMMU?



Dmabuf live update Steps – Cont.

2) Kexec Transition

1. PCI devices maintain Bus Mastering enabled, allowing background DMA operations to continuously write data into the physical memory corresponding to chunks[]. **Currently No questions if IOMMU liveupdate works well**

3) Incoming Phase (New Kernel Recovery)

1. **Memory Protection (Bootmem/Memblock):** At the very early stage of booting, the new kernel parses KHO and calls memblock_reserve() to directly lock the physical memory in chunks[], preventing the Page Allocator from mistakenly allocating it.

2. **IOMMU Mapping Inheritance:** The IOMMU driver looks up the requester_id and iova to directly inherit the existing hardware page tables without clearing or resetting the IOMMU Domain.

3. **Exporter Reconstruction:** The Exporter driver matches the exporter_name to reconstruct the struct dma_buf.

Question: DMA-BUF, PCI, VFIO, and IOMMU operations are asynchronous. How can we ensure DMA-BUF starts only after all of them complete, and how should DMA-BUF be notified?

4) Re-binding Phase

1. The PCI driver re-establishes the association via dma_buf_attach_preserved(), directly reusing the existing IOVA.

2. The LUO daemon assigns the restored struct dma_buf to a new userspace FD and injects it back into the target process (e.g., QEMU). **Question: PCI Device Context and HW State Resynchronization: The presence of a dma_buf does not imply that the PCI device's hardware state is ready or synchronized after kexec. When dma_buf_attach_preserved() restores the buffer in the incoming kernel, the driver must also restore and synchronize the device's ring pointers, interrupts, doorbells, and other hardware context with the preserved buffer state; otherwise, the device may access incorrect buffer locations, resulting in packet loss or data corruption.**



Dmabuf live update problems

1. Roadmap for PCI core liveupdate, VFIO liveupdate and IOMMU liveupdate since dmabuf is based on the above 3 components.
2. Dmabuf is based on PCI core, VFIO and iommu liveupdate, in preserve phase, dma liveupdate should check the state of PCI core, VFIO and iommu state?
If yes, does any interface in the above 3 components exist for lookup the current status?
If no, if one of the above 3 components fails, is it necessary to preserve the state of dmabuf?



Problems – Cont.

1. Dependency

In my setup, the hardware NIC performs queue leasing to netkit, and rxe is then configured on top of netkit. This creates a strict dependency chain where the hardware NIC must be configured first, followed by netkit leasing the queues, and finally rxe being instantiated. Consequently, rxe depends on netkit, and netkit depends on the hardware NIC. Is there a mechanism available during live updates to ensure this dependency chain is maintained?

iommu-liveupdate is based on pci-liveupdate and vfio-liveupdate, vfio-liveupdate is based on pci-liveupdate. When kexec reboots kernel, how to make pci-liveupdate restores firstly, vfio-liveupdate restores secondly, then iommu-liveupdate restores finally?

If a new kernel module depends on one of pci, vfio and iommu live update, how to control boot order?



Thanks for your attention!

