Device Passthrough to Driver Domain in Xen

Passthrough. List of terms.

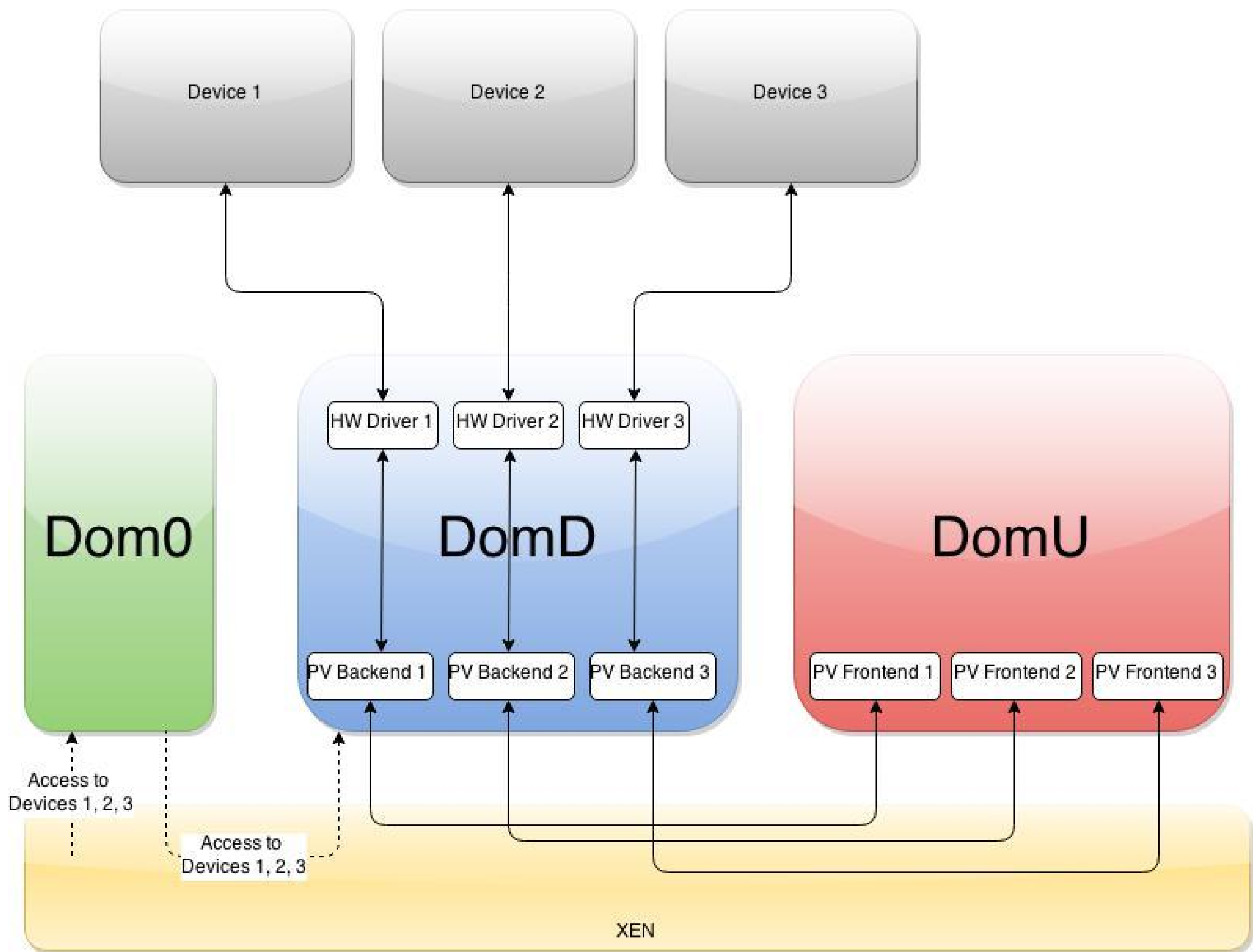
- Passthrough: the process of passing access to DomD via Dom0
- Dom0: Domain 0, a privileged domain
- DomD: Driver Domain, a domain with hardware drivers that has access to real devices

Passthrough: why do we need it?

As we all know, one of the most common sources of OS crashes are hardware drivers and the issues with them. On systems with visualization, it seems logical to create a separate domain and place hardware drivers (or at least the buggiest of them) there. All other domains that need access to these devices should use special "frontend drivers" that do not have direct access to the hardware but instead connect to "backends" in the driver domain. These backends manage access and redirect all requests via hardware drivers to the appropriate devices.

A system with this type of configuration becomes much more stable. If panic occurs in the driver domain, it gets rebooted. All other parts of a system stay functional. Although the system may lose some hardware functionality during the driver domain reboot, this functionality gets restored when the DomD with backends is online again. Xen provides examples of creating a Network Domain (i.e., a domain with network drivers and access to network devices) and a Block Device Domain (i.e., a domain with block device drivers and access to block devices). Also, PCI devices can passthrough to PV domains.

One of the most significant tasks of creating a system with such a driver domain is to correctly provide it with resources (e.g., IO memory, IRQs). For example, in order to create a driver domain with all (or almost all) peripheral devices of ARM SoC for an automotive solution, we have to grant DomD with access to IO memories and IRQs of about 50-100 devices.



Example of System with Driver Domain based on Xen Hypervisor

Passthrough: what do we want?

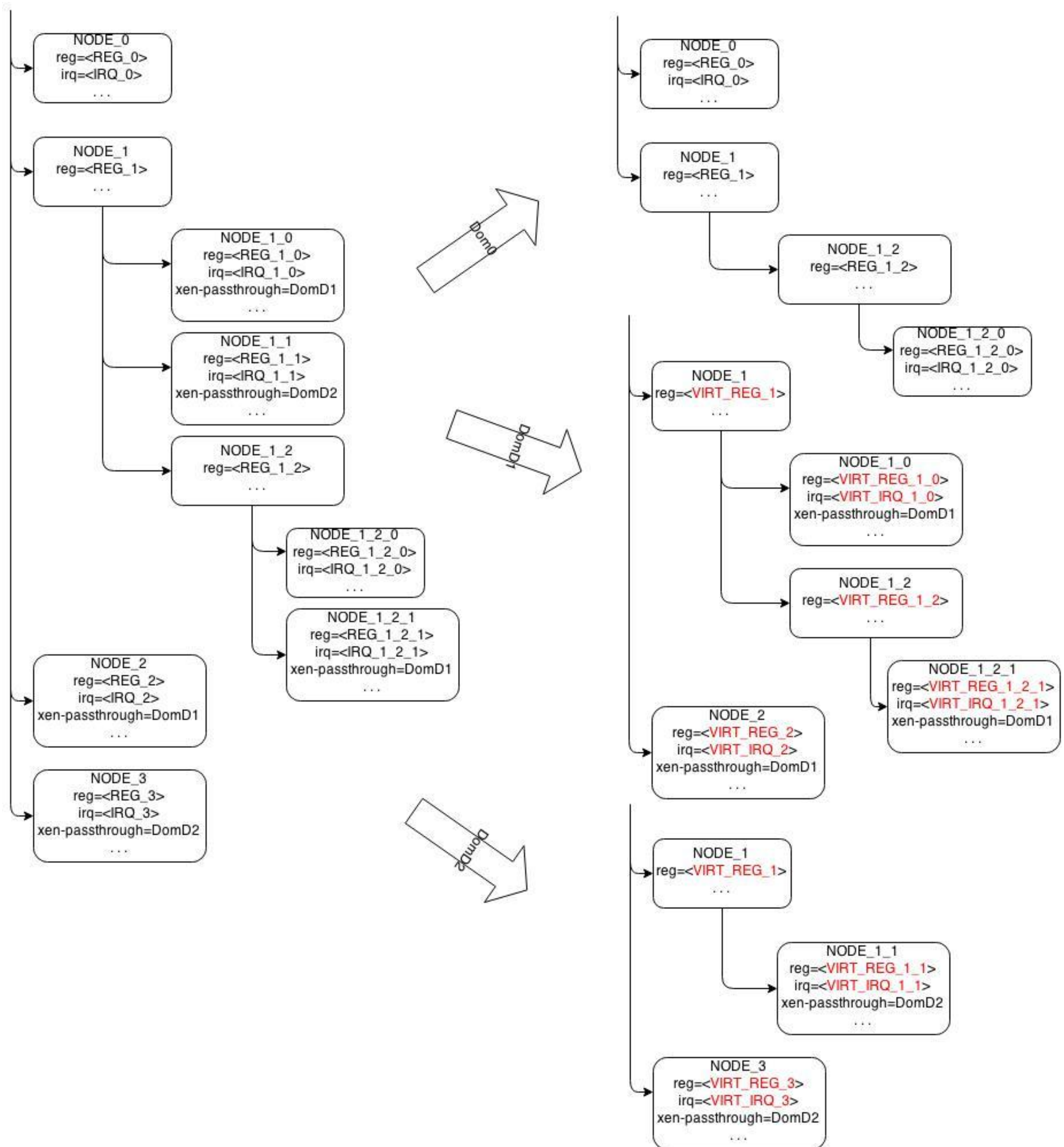
The main idea of passthrough is to grant access for DomD through Dom0. The best solution is just to mark which device should be assigned to which domain. It can be implemented by adding a special field to device tree nodes. Then, while constructing Dom0, Xen leaves nodes without this field unchanged and grants IO-mems and IRQs to them. It removes nodes (with all fields and child nodes) with a "xen-passthrough" field from a device tree and just grants them access to Dom0. When Dom0 initiates DomD construction, the initial device tree is parsed again, and all nodes that don't have a "xen-passthrough" field with the name of an appropriate domain and don't have such children are removed.

In this way, all redundant nodes are filtered. IO-mems and IRQs are then mapped for any nodes that remain after filtering. It is important to note that DomD resources mapping should be not one-to-one. New virtual addresses and IRQs are substituted in the "reg" and "irq" fields of the appropriate device nodes.

There may be some places in the kernel where access to a device is provided via hardcoded addresses rather than values from the device tree. You should change such

places and use the right way (i.e., via a device tree). It is a fee for pure virtualization. Alternatively, you can just directly map such regions via the "iomem=[]" command in the DomD configuration file.

ARM architectural restriction means we can manage memory access only in the context of memory pages. In other words, if we want to give access to DomD to some memory region that is unaligned to page size, we have to give access to the whole page(s) that contains this region. This is a security violation regardless of how you give access to DomD, either via the "iomem=[]" command in the configuration file or via passthrough. The below figure demonstrates how you can divide a single device tree into separate device trees using Xen during domains construction.



Passthrough: how does it work?

At GlobalLogic, we leverage an automotive solution called Nautilus that is based on dra7xx board (TI dra7 ARM SoC) and Xen 4.5 on-board. This solution has a successfully integrated driver domain, Linux kernel 3.12 as Dom0, and Linux kernel 3.8 as DomD.

After Dom0 starts, it creates DomD and grants IO-mems and IRQs to it via a domain configuration file by means of "iomem=[]" and "irqs=[]" commands. A device tree for Dom0 is cleaned up, and only devices that are dedicated to Dom0 stay there. A device tree for DomD is not constructed by Xen; instead, it is built and attached to the kernel image during compilation. It contains only devices dedicated to DomD. In the DomD kernel, there are requests to hardcoded IO-mems (not listed as a device in the device tree). Such IO-mems should also be listed in a domain configuration file.

Moreover, there is one more Xen 4.5 security requirement. Dom0 can grant to other domains only IO-mems and IRQs that are granted to Dom0. In other words, Xen gives access to IO-mems and IRQs to Dom0. Then Dom0 gives access to these IO-mems and IRQs to other domains. In Nautilus, it is implemented by adding fake nodes to the Dom0 device tree for devices that should be granted to DomD, with only fields "reg" and "irq." This way, Xen will automatically grant access to IO-mems and IRQs while creating Dom0. Then Dom0 can create DomD and grant access to these resources.

Still, this method has a lot of disadvantages. First of all, we need two separate device trees for Dom0 and DomD. Both of them significantly differ from the original device tree, which is used for a system without Xen. This means you need to separate and modify a device for each domain. Furthermore, after an original device tree has been updated, changes will probably not be automatically applied to separated device trees. You will likely need to separate the new original device tree or manually change a device tree for each domain.

Secondly, this method contradicts the "one place source" principle. If you need to add some IO-mems or IRQ for DomD, you should do it at several places: add a node to the DomD device tree, add it to the DomD configuration file in "iomem" or "irqs," and create a fake node in the Dom0 device tree. The same theory applies to modifying or deleting, which makes it a potentially buggy place. Finally, DomD has one-to-one mappings for IO-mems and IRQs, which goes against the very ideology of virtualization. Nevertheless, the main advantage of this method is that it successfully works.

Passthrough: how is it implemented?

There are two main tasks to implement passthrough for systems with DomD: (1) separating a common device tree into several device trees for each domain and (2) mapping domain resources (i.e., IRQs and IO-mems).

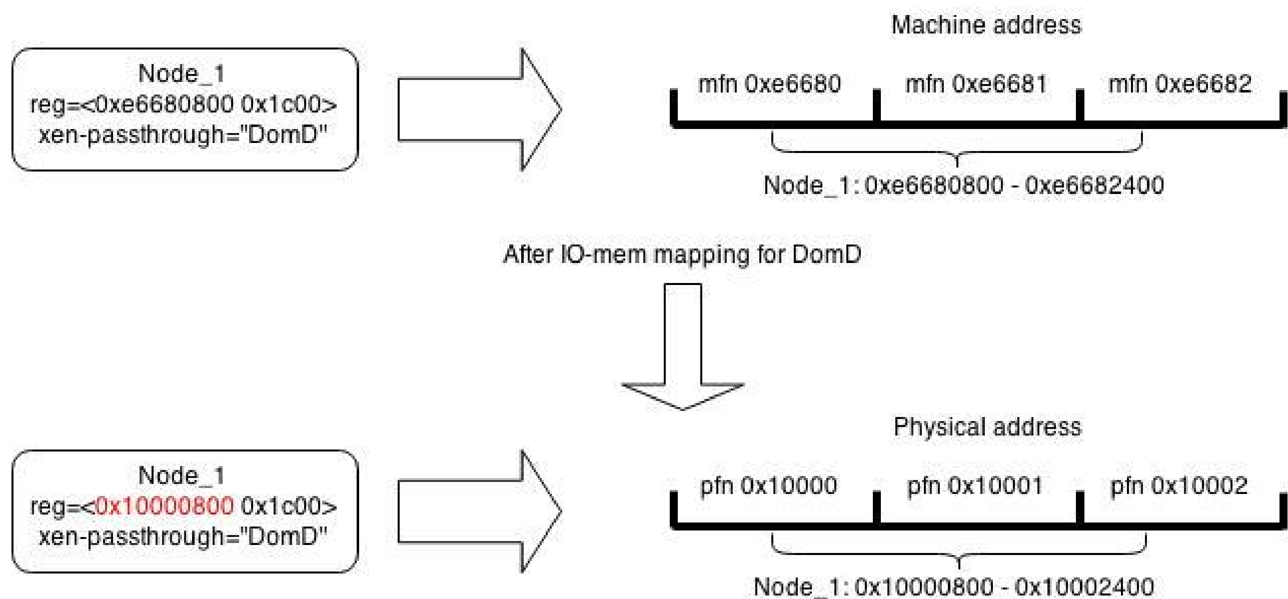
There are two basic ways to divide a device tree. The first method is to copy a common device tree to a target and then traverse through it, deleting the odd nodes that don't have a “xen-passthrough” field and don't have child nodes with such options. The second

approach is to create an empty target device tree and then traverse through a common tree, copying the necessary nodes (i.e., the ones with a “xen-passthrough” field with their parents) to the target. For both methods, we need libfdt from xen/common. All nodes should be copied with all their fields and child nodes. Additional investigations are needed to understand which method has better performance metrics and is easier to implement.

Since Xen now maps resources for Dom0 automatically from a device tree, not much work is needed to provide access for Dom0 with passthrough. An idea for IRQ and IO-mems mapping for DomD is to do the same operations as the ones done via a domain configuration file, but automatically and more elegantly.

While creating a domain with libxl, the library parses the configuration file and maps resources by means of libxc (xc_physdev_map_pirq() and xc_domain_irq_permission() for IRQs; xc_domain_iomem_permission() and xc_domain_memory_mapping() for IO-mems). Projecting to passthrough, after (or while) creating a device tree for a domain, we have to look through nodes for “interrupts” and “reg” fields and map them. Since we have access to a target device tree, we can use virtual IRQs and addresses of IO-mems (they are returned from libxc) for a domain, replacing the original values in “interrupts” and “reg” fields with a virtual one. Ranges of virtual memory and IRQs can be taken from special Xen nodes, or -- if they are absent in a device tree -- a predefined one can be used.

The below figure demonstrates an example of an unaligned memory region mapping. As you can see, all pages covered by IO-mem reg should be mapped. The node's reg field is replaced with a domain's physical address of a first page plus a reg offset inside page.



A lot of questions about mapping remain open, such as “What should we do if several nodes use the same mfn? Should we use different mapping for each node, or should we save domain mapping and use the same pfn for all such nodes?” Another problem is a hidden security violation when we map unaligned regs (due to ARM architectural restrictions). However, if we implement passthrough functionality in Xen, we will make great progress in creating stable, successful systems with DomD based on a Xen hypervisor.