6.823 Computer System Architecture

Nested Paging

http://csg.csail.mit.edu/6.823/

This handout explores the architectural impact of running a process in a virtual machine, as it relates to virtual memory and address translation.

Platform virtualization (or simply virtualization) is a technique to allow running multiple operating systems over the same physical hardware. Each of these operating systems, called a *guest operating system*, believes it has access to a full physical machine (with a CPU, memory, disks, and other devices). But in fact these resources are emulated—they constitute a *virtual machine*, rather than a physical one, as shown in Figure 1.

The actual physical hardware is in control of the *host operating system*. The host OS in turn can run one or more virtual machines, each with its own emulated resources and guest OS. The host OS abstracts each physical resource and arbitrates access to it among the guest OSs.

Much like an operating system brings the benefits of protection among processes and abstraction of the underlying machine's resources, virtualization achieves protection and abstraction at the level of full virtual machines. Therefore, virtualization has many applications. For example, it can be used to consolidate multiple legacy systems on the same physical hardware; provide stronger levels of security; make it easier to rent physical hardware, as is done in cloud computing; migrate virtual machines among physical machines, etc.

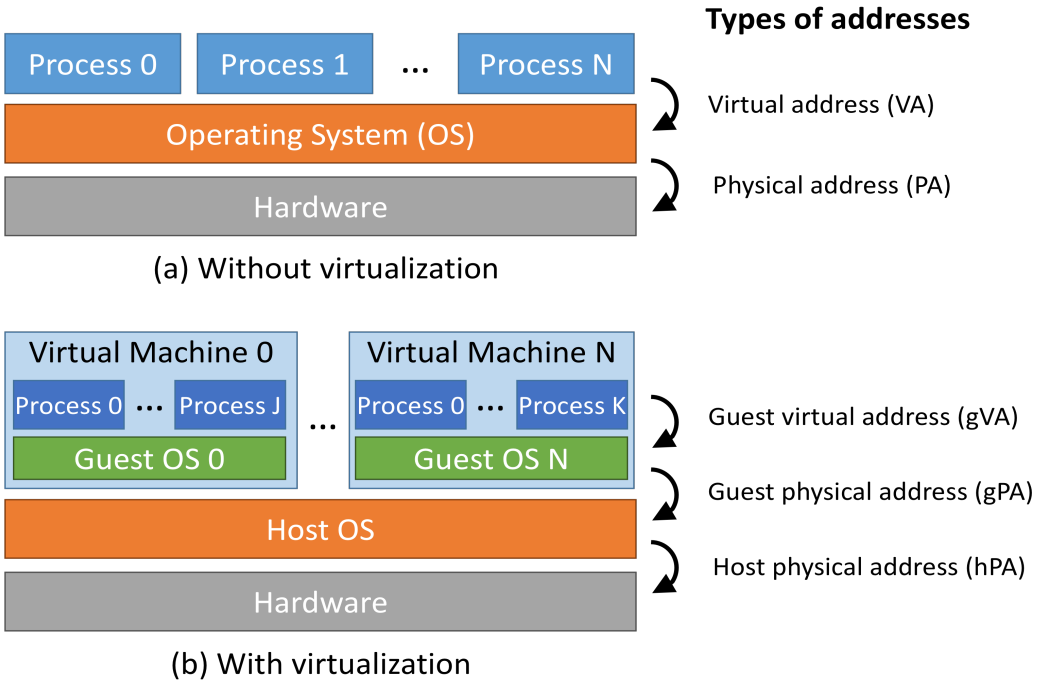

Figure 1: Key elements in systems (a) without and (b) with platform virtualization, and corresponding layers of address translation.

Virtualization requires emulating each of the physical machine's resources. Here we focus on memory. We will discuss other facets of virtualization later in the course.

Each guest OS has access to its *guest memory space*, distinct from the *host memory space*. Processes running on the guest issue loads and stores using virtual addresses, and these are translated to physical addresses, just as we have seen before. However, because the guest memory is distinct from the host memory, this translation alone is insufficient for the host to serve the data to the guest from its real physical memory. In fact, two layers of address translation are required, as shown in Figure 1:

- **gVA=>gPA**: guest virtual address to guest physical address translation.
- **gPA=>hPA**: guest physical address to host physical address translation.

Every guest physical address appears to the host as a host virtual address.

Nested *paging* is a common design to perform these two address translations. Figure 2 shows an implementation of nested paging using 2-level hierarchical page tables. Below, the host performs `gPA=>hPA` translations through a per-VM 2-level hierarchical page table that **resides in host physical memory**, and whose **PTEs use host physical addresses.**

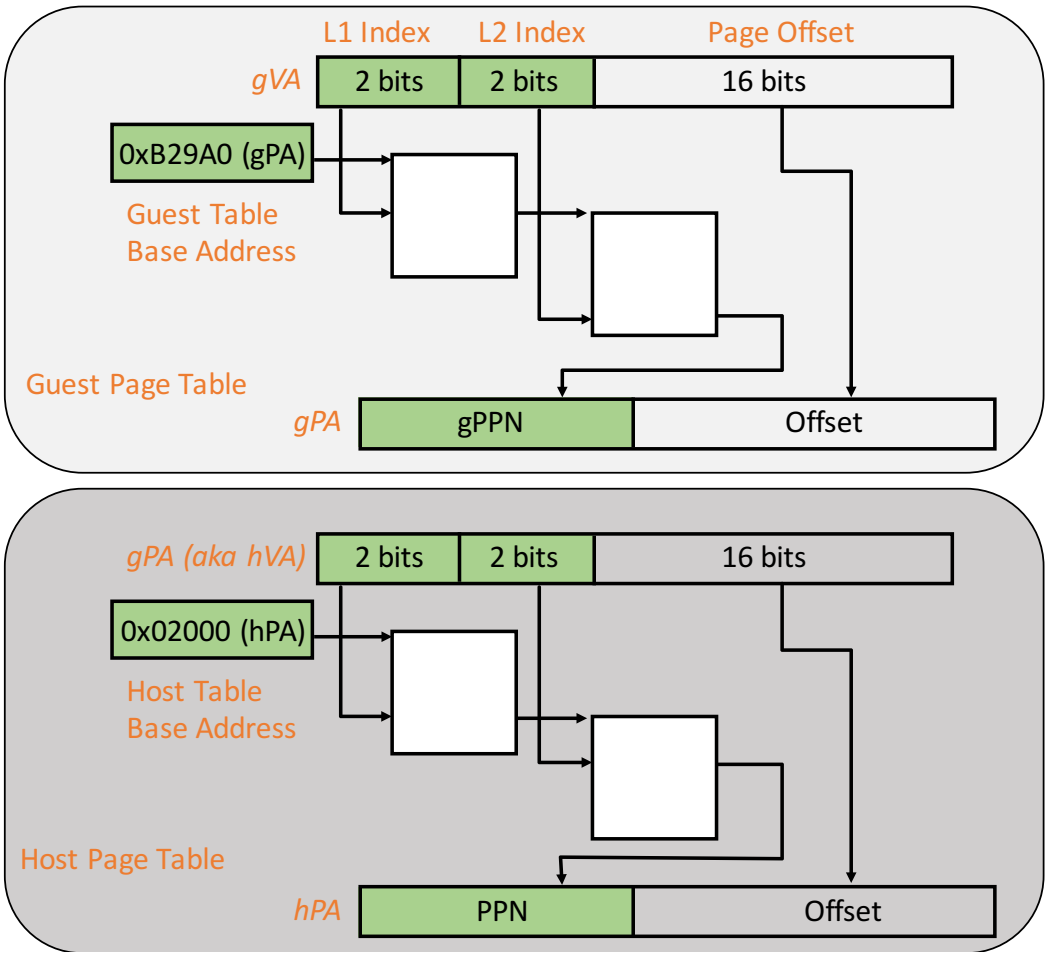

Figure 2: Implementation of *nested paging* with 2-level hierarchical page tables.

The guest also has a 2-level hierarchical guest page table for the gVA=>gPA translation, whose **PTEs use guest physical addresses**. We assume the guest L1 and L2 page tables reside in host physical memory. This means that every guest table lookup (which uses a gPA) requires an additional gPA=>hPA translation. This complexity exists because the guest is unaware it does not run on a real processor!

To make this process efficient, modern systems have hardware support for nested paging. The processor's MMU has two page table base address registers, for the host and guest page tables. The host table base address register holds an hPA, and the guest table base address register holds a gPA.

The processor's TLB caches only the full gVA=>hPA translations. On a TLB miss, the MMU translation performs the nested page table walk. Notice that the lookup of a guest L1 table entry (i.e. 0xB29A0 + gVA's L1index) itself requires a (nested) translation of guest physical address into a host physical address, in order to fetch the relevant guest L1 entry from host physical memory.