Memory Management

Computer Science 432 — Lecture 7 — Duane Bailey

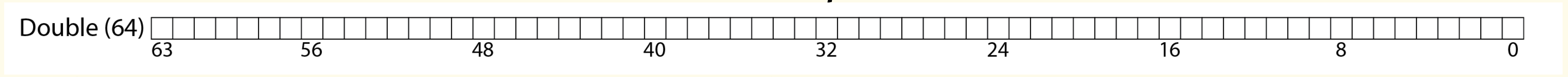
February 23, 2022

Announcements

- Chapter 4 (System Calls) is due on Monday.
- Small group meetings today and tomorrow.
  - Tell me if you need me to re-pull Lab 2.
- Questions on Lab 3?

The Memory Model

- *Memory* is simply a linear, contiguous arrangement of bytes in a *store*.
  - The store is typically "main memory" or "DRAM". (DRAM is actually a technology.)
  - The offset from the beginning of the store is the "address".
  - The store is typically byte-addressable. Every byte has a unique address.
  - We often think of these addresses as *pointers*.



- Although memory is byte-sized, machines and OSs may prefer to manipulate larger units.
  - E.g. The stack-pointer is *quad-word* aligned. Many reasons why this might be.
    Stack pointers end in 4 binary zeros: ….0000.
  - E.g. Memory is often logically segmented into *blocks* or *pages*. 4K bytes = 1 block.
    Block pointers end in 12 binary zeros: ….0000 0000 0000

- At the OS level, *everything* is all about pages. *And why not?!*

Virtual Memory Translation

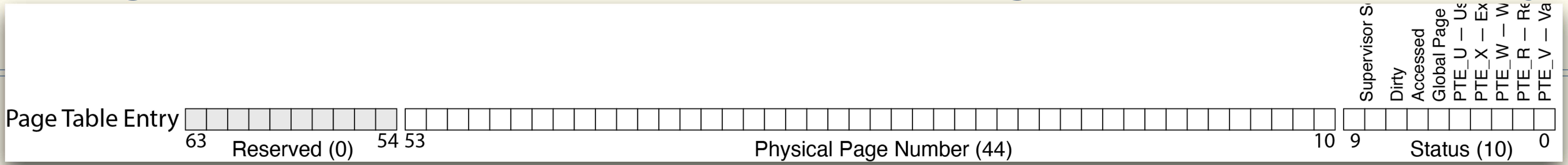
- Advanced, modern processors all reference memory through a system of *virtual addresses.*
  - Virtual addresses allows processes to act as if they're using a large contiguous memory.
  - The virtual memory is typically larger than what is actually available in the machine.
  - Ideally, the virtual memory is *fast*.

- Memory management units.
  - Physical memory is a physical device, often sized differently, not contiguous. Slow.
  - The *memory management unit* is a device adjacent to the processor that interfaces the two models of memory:
    - Includes devices: *caches (Li$)*, a *page table walker*, and a *translation lookaside buffer* (TLB). (Note to Mom: "Studying TRANSLATION LOOKASIDE BUFFERS. Aren't you proud?")
    - Includes protocols: approaches to *translation*, *cache coherency*, and *exception handling*.
  - If successful, the complexity of this unit will rival that of the processor, itself.

Who, what, when, where, and how.

- The processor ships out virtual addresses.
- The memory—by definition—accepts only physical addresses.
- The memory hierarchy must interface/translate between these warring parties…
  - Goal: Do not touch offsets within a page. Performance across a page should be uniform.
  - Goal: Replace the higher bits of the virtual address, the *virtual page number*, with a different number of bits, representing a *physical page number.*
  - Goal: Find page number translations in the TLB while looking for the data in the L1$.
    - First level caches are *virtually* addressed, but *physically* tagged. Cache starts searching for a virtual address, but verifies the tag against the higher bits of the co-translated physical address. Higher level caches: *"We're all physical."*
  - Goal: If TLB misses, "Walk the page table" (a Johnny Cash ditty?) and cache the translation.
  - Goal: If page table walking fails, *get the OS to Just Fix the Problem.* Page fault handling.

Storing a Translation: The Page Table Entry



- Somewhere, there's a *page table entry* that solves your virtual page number translation problem
  - RISC-V PTE: takes up 64-bits (8-bytes) describing a single translation:
    - A 44-bit *physical page number.* It is the translation for the *virtual page number* bits that got you here.
    - *Metadata:* 10 status bits describing page entry (*valid*) or the page itself (*read, write, execute, user, dirty, et al.*)

- A *page table* is simply a list of page table entries, indexed by *virtual page number*.
  - On RISC-V, virtual page numbers are 27-bits long.
  - Page table has up to 2^{27} entries, taking 1Gb of memory to store.

Wowza! Yikes!

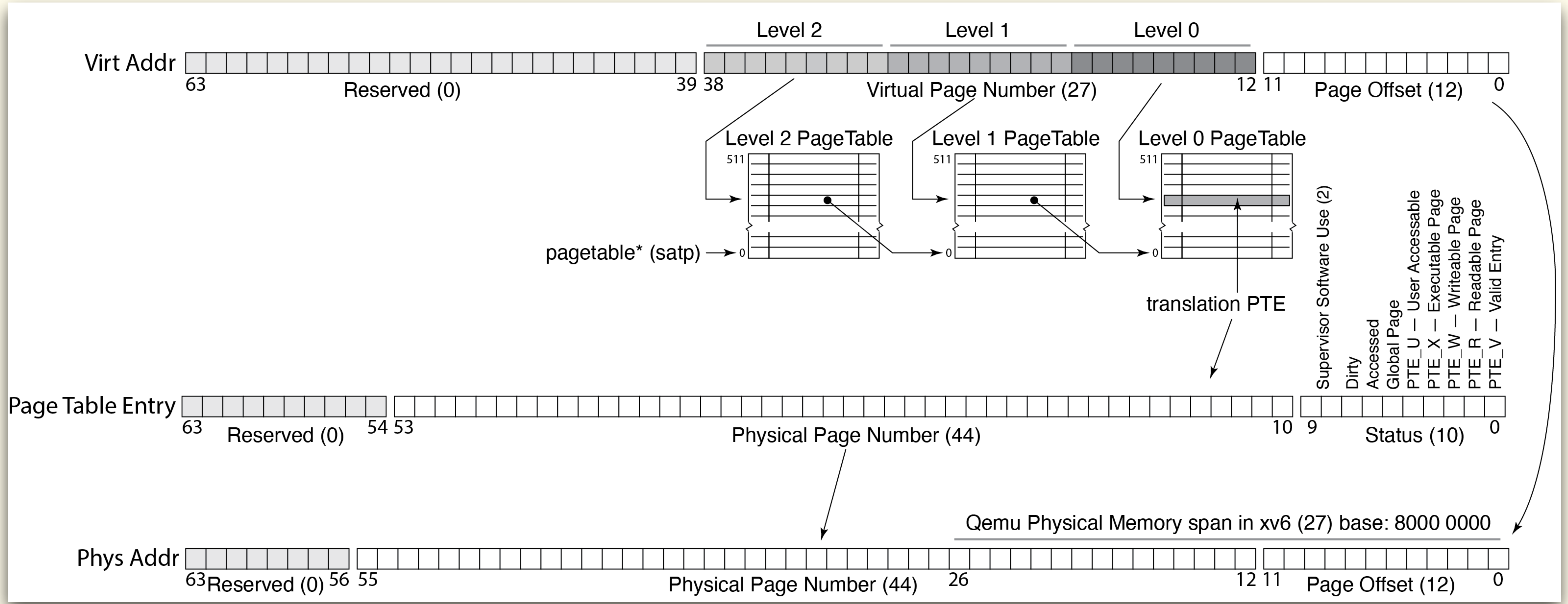
Common Solution: Hierarchical Page Table

- Observation: 1 block (4096 bytes) can store 512 64-bit values, indexed by a 9-bit address.
    - E.g. 64-bit *page table entry*
    - E.g. 64-bit *pointer* to another block (a pointer whose low 12 bits are zero!)
- Observation: Virtual memory is often sparse, not contiguous.

- Store the (sparse, non-contiguous) page table as a 3-level, 512-ary tree:
    - Leaf blocks of page table tree (all at bottom level) are collections of *page table entries*.
    - Interior blocks (two levels) are collections of *lower level block pointers*—***physical addresses***.
    - Blocks at each level are indexed by 9 bits from the virtual block address.

    - If any entry is zero, it's not a useful entry.
    - Only page table entries with the "valid bit" set (bit 0) actually store translations.

Common Solution: Hierarchical Page Table



A Symbiotic Relationship

- Only the operating system can make a tree:
  - The page table is constructed by the OS:
    - As the machine boots, the kernel's page table is built in `kvminit` (kernel/vm.c)
    - As processes are constructed, page tables are built in `proc_pagetable` (kernel/proc.c)
  - Page tables are maintained by the OS:
    - Memory allocation leads to new virtual memory addresses for allocated physical page.
    - Sets PTE status bits: e.g. when loading a new executable.

- The hardware uses the tree to form translations
  - The `satp` CSR register holds the physical address of the root block of the tree.
  - Hardware will (1) assume the tree is in memory and (2) will walk the tree to form the translations. *Ideally*: everything's in memory, and cached in L1$; done in 3 reads.
  - Caches any translations (and metadata…like, uh, what?) in the TLB.

Failure is not an option.

Failure by the hardware to form a translation generates a *page fault*.

404: Page faults are not fun.

(Millions of instructions (and several chapters) later) the OS will *Just Fix the Problem*.

xv6 kernel memory mapping

- Rule 1: The kernel manages all of physical memory.
- Rule 2: The kernel can always access physical memory using an identical virtual memory address.
  - Called direct addressing. Simplifies many things: e.g. user page table can be thought of as translating user virtual addresses into kernel virtual addresses. *Cool!*
  - Only meaningful if the virtual page numbers are as long as physical page numbers. Not true. But "practically true".
- Rule 3: Top page of virtual memory is a shared block of mode-switching code called the *trampoline. (Note to Mom: "Playing with TRAMPOLINES. Please pay health insurance.") Also* directly mapped!
- **[Huh: Several processes may be executing code in the kernel, from different cores.]**
- Rule 4: Kernel shares one block (the *trap frame*) with *each process;* writeable by trampoline.
- Rule 5: Kernel has one block of stack *for each process.*
- Rule 6: Kernel places *guard pages* between critical areas where rogue processes may fall.

- *Not* Orwell: Break any of these rules sooner than build anything outright barbarous.

xv6 kernel memory layout.

- Kernel has full, direct access to physical memory;
  virt. addr. = phys. addr.

- Everything in the computer is stored somewhere in the physical memory highlighted on the right

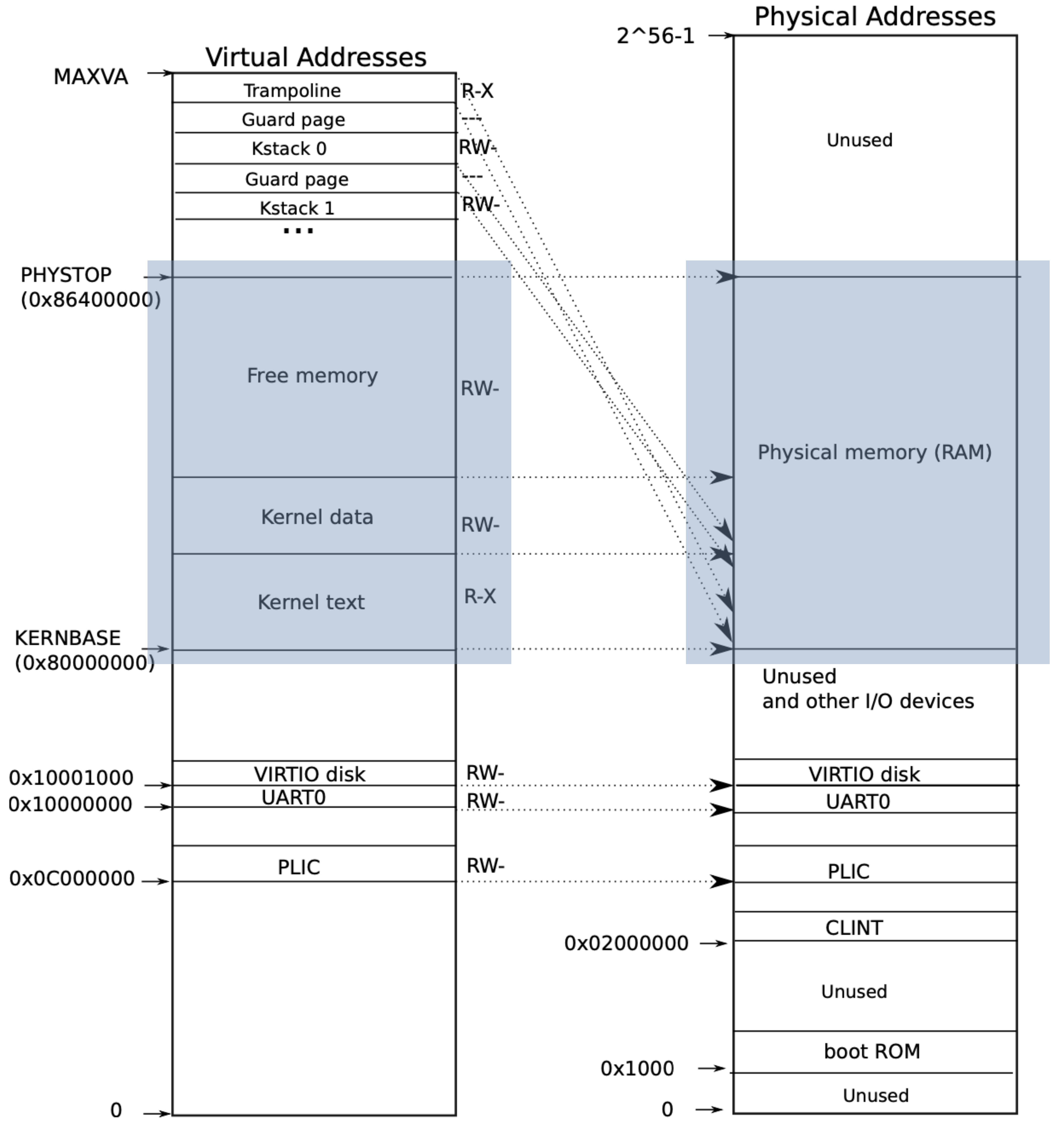


Figure 3.3: On the left, xv6's kernel address space. RWX refer to PTE read, write, and execute permissions. On the right, the RISC-V physical address space that xv6 expects to see.

xv6 kernel memory layout.

- Some pages are virtually mapped; these pages can be accessed *via* two addresses.
- Here, the Trampoline is mapped to the highest virtual block, but is also available through direct mapping.

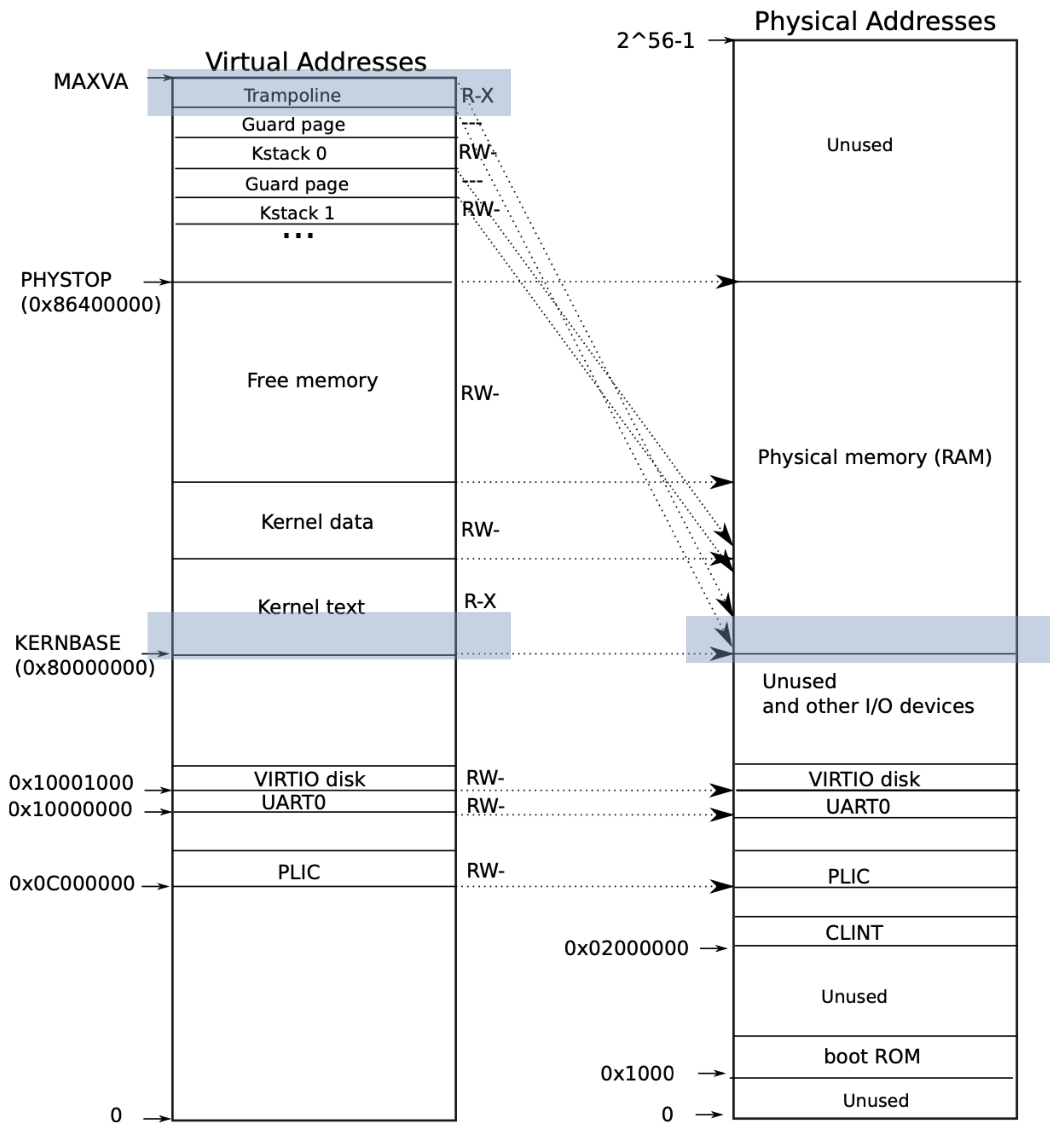


Figure 3.3: On the left, xv6's kernel address space. RWX refer to PTE read, write, and execute permissions. On the right, the RISC-V physical address space that xv6 expects to see.

xv6 kernel memory layout.

- Code in the kernel uses a stack that is dedicated to the current process. The stack is in high memory.
- Some areas, including the kernel stack, are bracketed by *guard pages* that are are not accessible. Notice this does not reduce the amount of physical memory available.

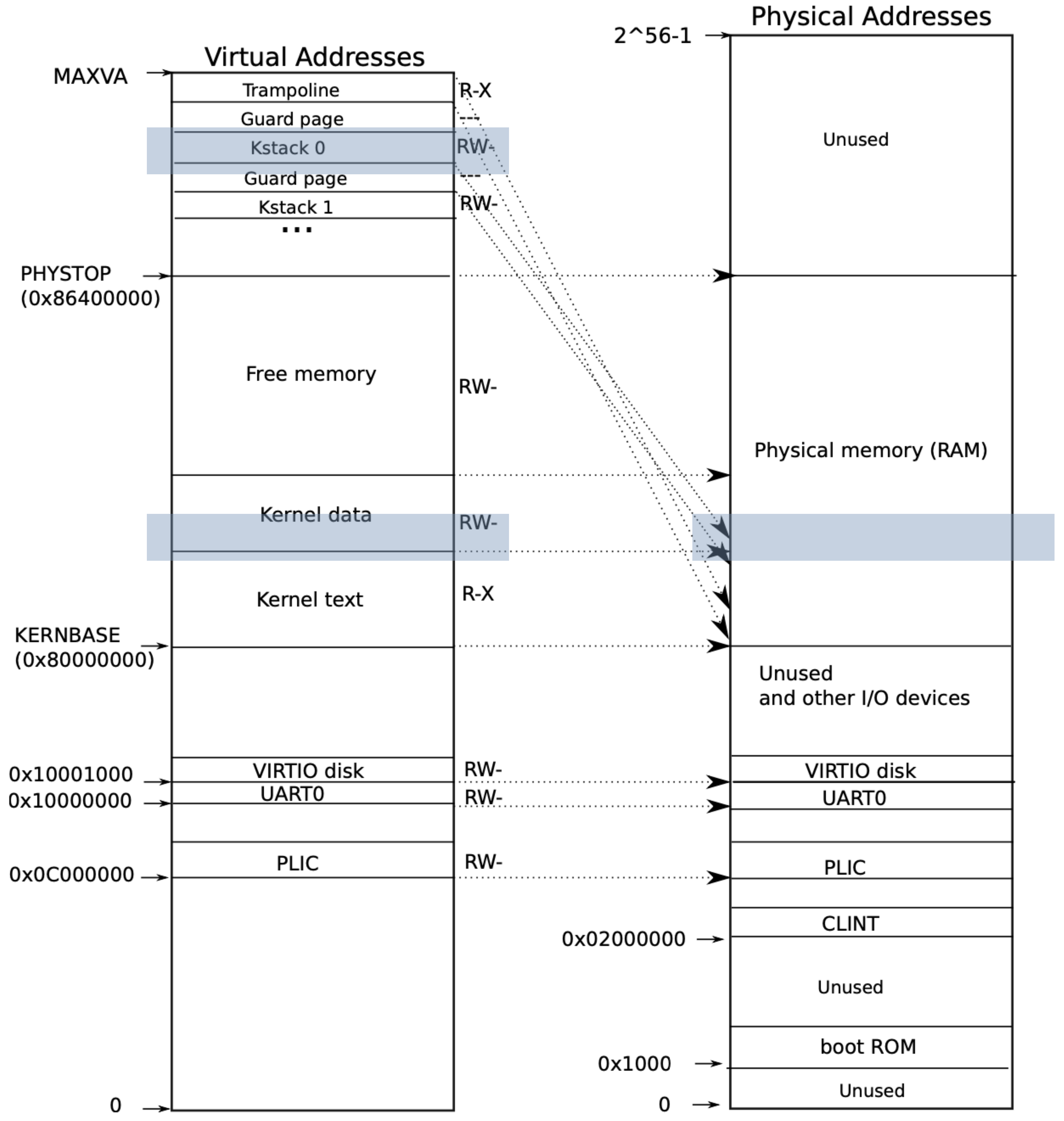
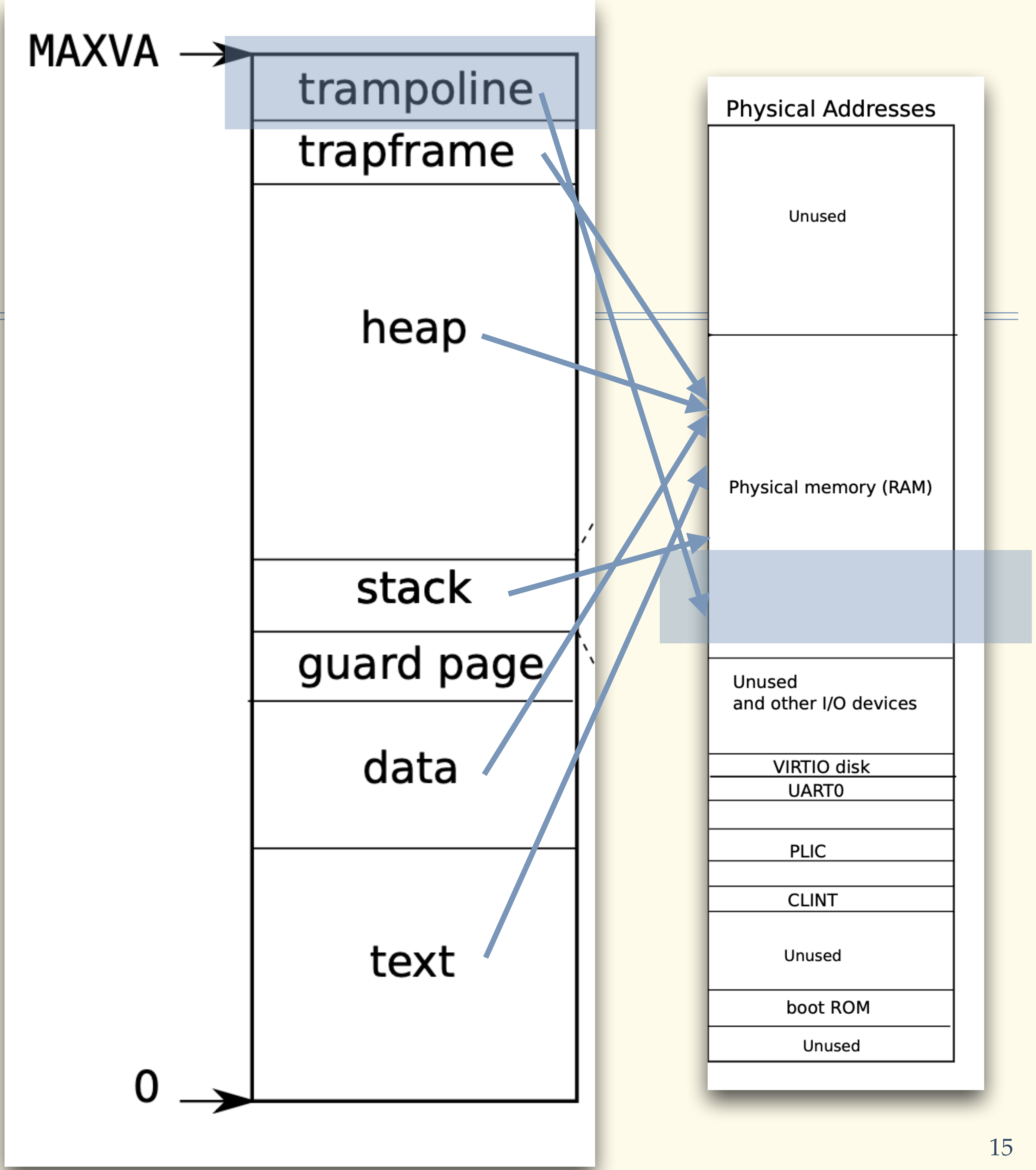


Figure 3.3: On the left, xv6's kernel address space. RWX refer to PTE read, write, and execute permissions. On the right, the RISC-V physical address space that xv6 expects to see.

xv6 userland memory layout.

- There is, effectively, no relationship between virtual addresses for user processes and their mapping to physical memory. Allocation of process pages is controlled by the kernel memory allocator (kalloc).
- The trampoline page (blue) is mapped exactly like it is in the kernel. Why? The page table changes in the middle of trampoline execution.


MAXVA
trampoline
trapframe
heap
stack
guard page
data
text
0
Physical Addresses
Unused
Physical memory (RAM)
Unused
and other I/O devices
VIRTIO disk
UART0
PLIC
CLINT
Unused
boot ROM
Unused