Process Address Spaces Using Paging

- Paging allows non-contiguous regions to be mapped in Virtual Address (VA) space and Physical Address (PA) space.
- To map a region a VA space, the page directory and page table entries at the corresponding offsets should be mapped.
- To map a region of PA space, the values in the page table entries should point to the corresponding physical pages.
- When paging is enabled, all addresses (including EIP, ESP, direct-addressing, etc.) go through the paging hardware.
- Each entry in the Page Directory (PDE - page directory entry) and in the Page Table (PTE - page table entry) is 32-bit wide. The top 20 bits store a pointer (using physical address) to the page table or the page itself. The last 12 bits contains flags: `present`, `writable`, and `user-accessible`.
- The final access permissions are determined by looking at the stricter of the permissions specified by the flags in the corresponding PDE and PTE.
- CR3, PDE, and PTE contain physical addresses

- Each process has a separate page table, so a different address is loaded into the CR3 register on every context switch.
- Also, each page table also maps the kernel into its address space above a certain address. This address is 0x80000000 (2GB) on xv6.
- Thus the kernel is mapped at the same addresses in every page table. So, copies of the same PDEs/PTEs are present in the page directory and page tables of all process to map the kernel at these addresses. Let's call this address space region, the kernel's address space. Another way to put this is that each process has two halves: kernel half and user half. The kernel half is shared among all processes. The user half is separate. Thus every user process (separate address spaces) also acts as a kernel thread (shared address space).

- On xv6, the kernel maps the entire physical memory into its address space starting at 0x80000000. Thus, virtual address `(0x80000000+x)` always translates to physical address `x`.
- The kernel's address space holds the kernel's code and the kernel's data. All the other space is managed as a heap (using `malloc` and `free` for example).
- The kernel's heap is used to allocate memory for its own data structures (e.g., PCBs), processes' kernel stacks, processes' address spaces, among other things.
- For example, if a kernel wants to allocate an address space for a process, it simply mallocs some space from its heap, converts the returned pointer to its physical address, and then creates a mapping into the process's user-side address space (by creating entries in its page table) so the process can access it in future.

- Thus every page that is accessible by the user also has a mapping in the kernel's address space. Thus two entries in the page table point to the same physical page (one in the *kernel-space*, and another in the *user-space*).
- The Interrupt Descriptor Table (IDT) is setup to hold kernel pointers, i.e., the CS:EIP entries are setup such that the handler runs in priviliged mode (last two bits of CS are zero), and EIP points to a kernel address (above `0x80000000`).
- On a trap (due to an interrupt, exception, or system call), execution control transfers to the kernel's handler running in privileged mode. Because the kernel is mapped in every process's address space, the pointers to the handler (and the kernel stack of the process) in the kernel address space are always valid. Thus, the handler can start executing immediately on a trap, without requiring any address-space switch.
- The kernel's handler executes the necessary logic. This sharing of address space between the process user-space and the kernel also allows very fast communication between the kernel and the user spaces. For example, if the user wants to provide a string argument to a system call, it can simply pass a pointer to the string stored in its own address space. The kernel can de-reference that pointer, and because the user's address space is still mapped, the de-referencing of the pointer will result in the desired data (as supplied by the user). Thus communication from user-space to kernel-space can be done only by sending a pointer - this is very fast. Recall that kernel can always access user pages (the user bit in PDE/PTE only prevents the user from accessing a kernel page).
- This fast communication between the user and the kernel is the primary reason, why the kernel maps itself entirely in the process page table. Linux maps itself starting at `0xc0000000` (3GB) and Windows maps itself starting at `0x80000000` (2GB), but can be configured to map itself starting at `0xc0000000`. As already discussed, xv6 maps itself starting at `0x80000000`.
- Because xv6 maps the entire available physical memory in the kernel space, there is a limit to the size of physical memory that it can support (less than 2GB). Full operating systems (like Linux/Windows) handle this by mapping a part of the physical memory at all times (esp. the parts which contain kernel's code and data). For the other parts of the physical memory, the kernel maps and unmaps those regions into a VA space, to access them, depending on what needs to get accessed.
- The page directory itself is allocated from the kernel address space (per process), and its corresponding physical address is stored in the `cr3` register when that process is running. Security is ensured by disallowing a process from accessing its own page table (by mapping the pages containing the page tables and page directory only in the kernel address space and not in the user address space).
- Compare this organization to a kernel which only uses segmentation. In that case, a trap would switch the `CS` register (and the associated base and limit values of the descriptor) and thus the kernel would be executing in a different address space. All other segment registers will also be loaded with the kernel's base, so they can access the kernel's data. However, if the kernel wants to read an argument from the user-space (passed as a pointer), it needs to switch one of its segment registers to the user's descriptor before de-referencing that pointer through that segment register.

  Notice that the kernel cannot simply de-reference a pointer supplied by the user in this case, as the address space is now different - the kernel needs to switch to user address space to de-reference that pointer.

- Compare this organization to another paging-based organization of the kernel, where the entire kernel is not mapped into the process address space, but only a small slice of the kernel address space (which stores the trap handler and the kernel stack) is mapped in the process address space. In this case, the trap handler will switch to the kernel's page table and then execute the kernel's logic in a different

address space (by switching to a different page table). However, if the kernel needs to de-reference a user pointer, it cannot do so. In this case, the trap handler should also de-reference all user pointers and copy them into its space before switching to the kernel's address space.

Because transitions between user and kernel require page table switches, and because the kernel cannot access user memory directly in this organization (the trap handler needs to copy portions of user memory for the kernel to read), this organization is relatively more complex and less performant.

Bootup : Executing the first instruction after power-on

Switching on your computer, makes it start from a clean state, where memory contents are completely uninitialized. Only the disk contains state that persists across power cycles. The x86 architecture specifies that when a computer is powered-on, the first block (or sector) on disk will be read and its contents pasted at address `0x7c00`, and control will be transferred to the instruction at its first byte (address `0x7c00`). Recall that the x86 architecture boots in 16-bit mode, with no paging. Also, the segmentation hardware in 16-bit mode simply multiplies the segment register's value by 16 and adds it to the virtual address to obtain a physical address.

A disk block (or sector) is sized at 512 bytes. Thus the machine reads the 512 bytes in the first block and pastes them at addresses `0x7c00-0x7e00` and transfers control to `0x7c00`. This code, which must fit in 512 bytes then loads the kernel from the disk (it must know the location of the kernel on disk) and pastes it into memory (it must know the location in memory where it needs to be pasted), before transferring control to it.