Paging

Mythili Vutukuru
CSE, IIT Bombay

Recap: Paging and page table

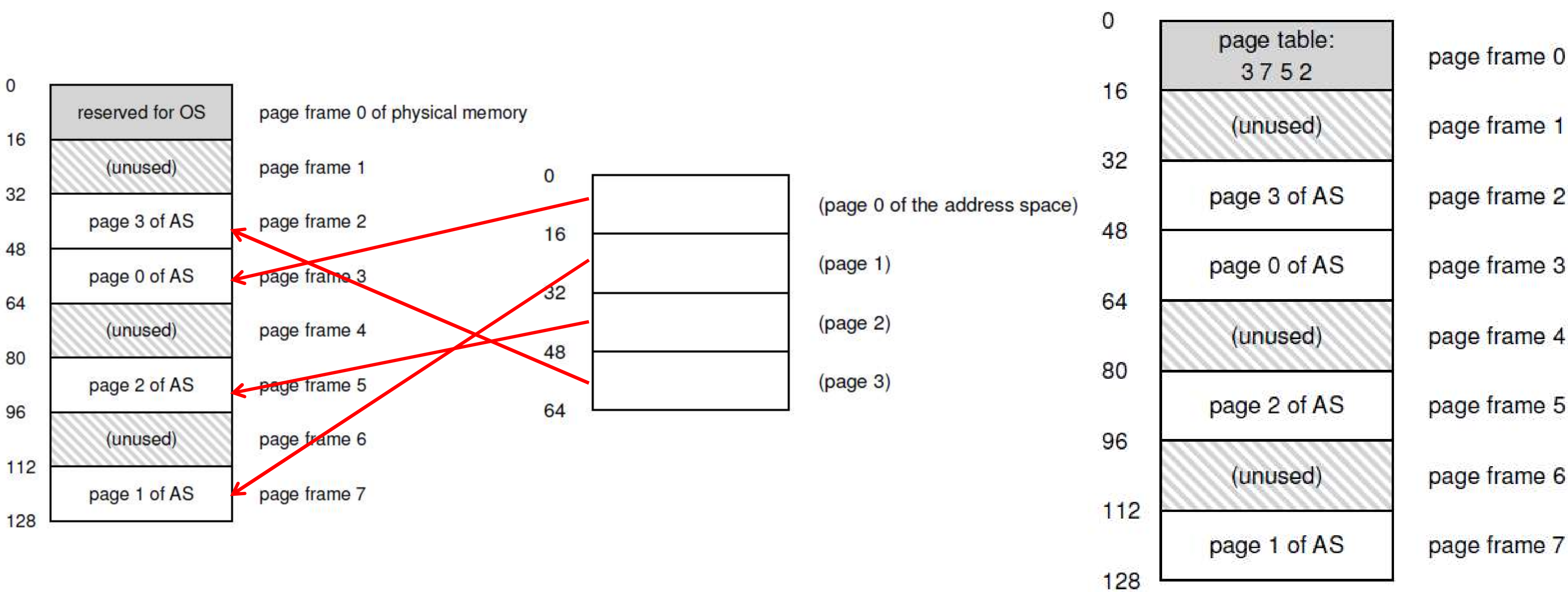


Image credit: OSTEP

Page table entry

- Page table is array of page table entries, one per page of process
- i-th page table entry (PTE) contains physical frame number and other details (permissions, status, ..) of i-th page of process
  - Valid: is this page in use by process (not all virtual addresses are used by process)
  - Various permission bits (more later)
  - Other status bits: present, dirty, accessed (more later)

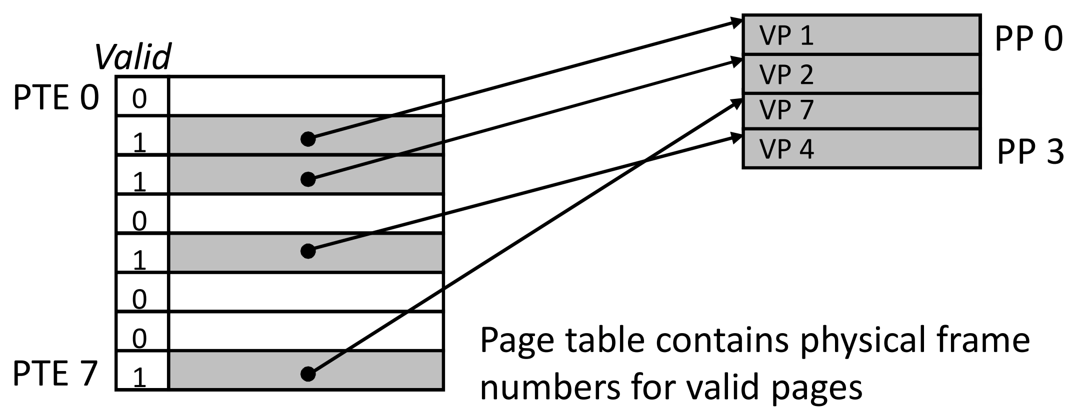


Image credit: CSAPP

Address translation in MMU

- MMU stores starting (physical) address of page table array in CPU register called page table base register
- Page size determines number of bits in offset
  - 4KB pages need log2(4K) = 12 bits as offset within page
- Remaining most significant bits give VPN
  - For 32-bit machines and 4KB pages, 20 bit VPN
- MMU uses VPN as index into page table array, accesses PTE, gets PFN, adds offset bits to get PA
- If no valid PTE found, MMU traps to OS

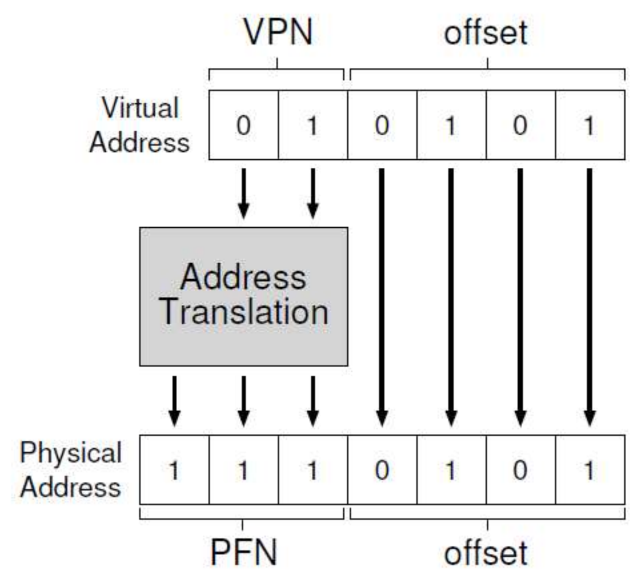


Image credit: OSTEP

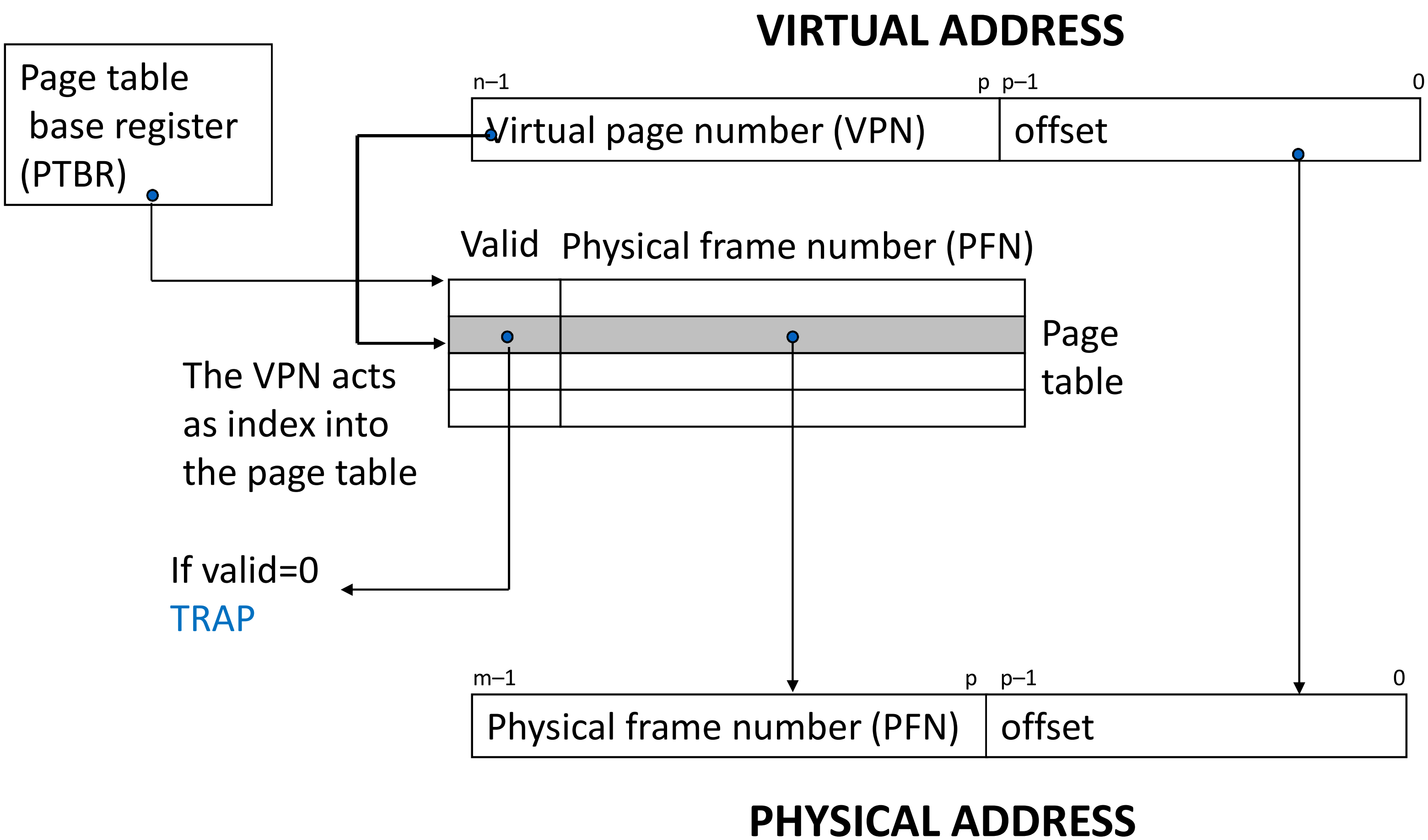


Image credit: CSAPP

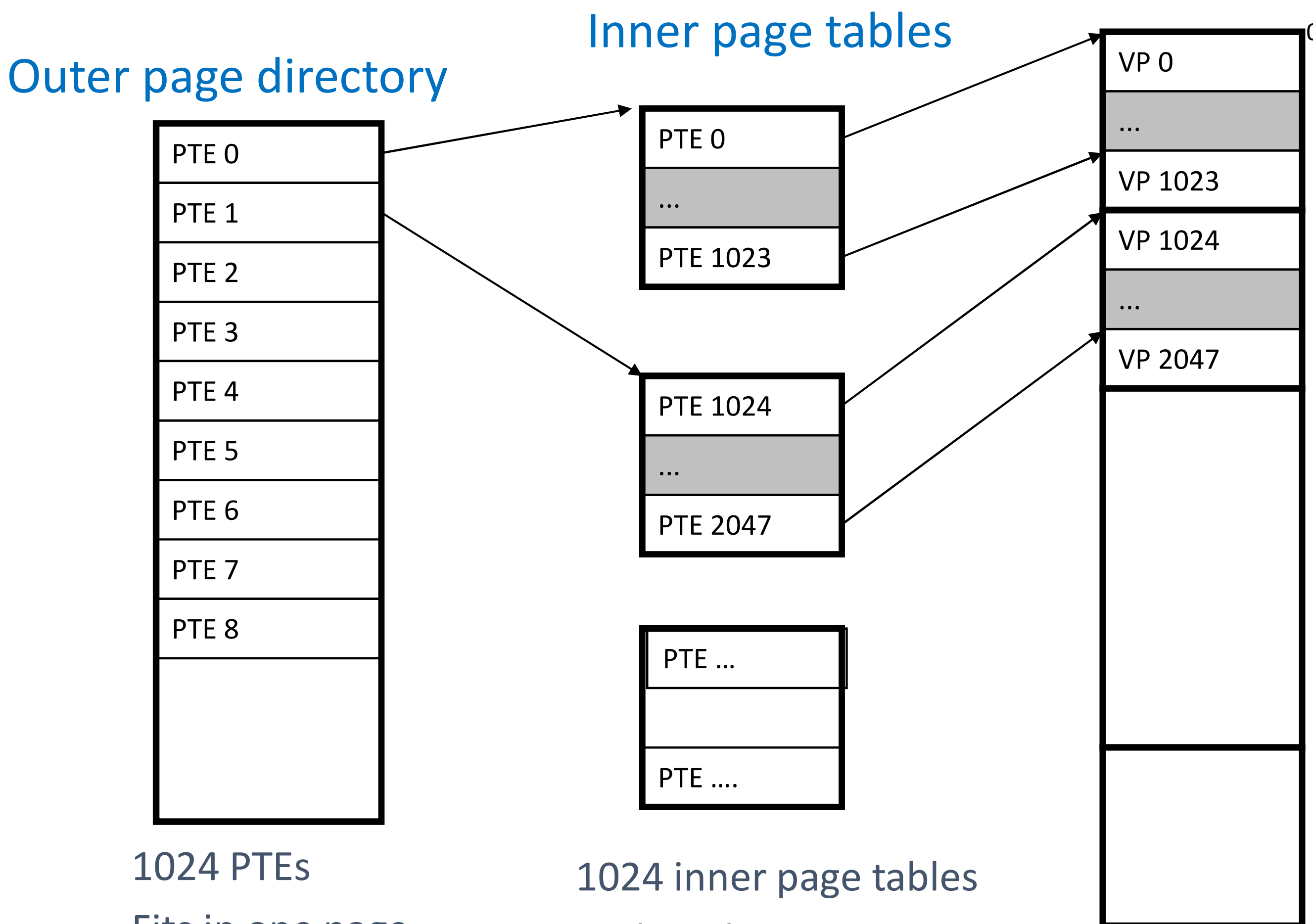
1K = 2^10 = 1024
1M = 2^20 = 1024*1024
1G = 2^30 = 1024 * 1024 * 1024
B = byte, b = bit

Size of page tables

- What is typical size of page table in a 32-bit system?
- 2^32 = 4GB virtual address space
- Assume page size = 4KB = 2^12
- Number of PTEs = number of pages in virtual address space = (2^32/2^12) = 2^20 = 1M
- If each PTE is 4 bytes, page table size = 4 bytes * 1M entries = 4MB
- How are page tables stored in memory?
  - All memory is only allocated in 4KB chunks, so how to store 4MB?
- Solution: split page table into pages (much like memory image), use another page table to keep track of original page table!

Two-level page table in 32-bit systems

- 4MB page table split into 1024 chunks of 4KB each (to fit in page)
- 1M PTEs split across 1024 pages, each containing 1024 PTEs
- Physical frame numbers of these 1024 chunks stored in an outer page table or page directory
  - 4 byte page table entry each, so outer page directory fits in one page here
- Page table now has two levels
  - Outer page table (page directory) has physical frame numbers of 1024 "inner" page table pages
  - Each inner page table has physical frame numbers (PTEs) of 1024 pages of the process virtual address space



Virtual address space can have 4GB memory = 1M pages

1024 PTEs

Fits in one page

1024 inner page tables

Each with 1024 PTEs

Each PTE contains PFN of one page of process

Image credit: CSAPP

Inner page tables on demand

- Note: not all inner page tables need to be created always, only those with at least one valid entry needed
- Example: Process with 2K pages of code+data, 6K + 1023 unallocated pages in address space, then one page allocated for stack
  - First two inner page tables are allocated, hold the 2K valid PTEs
  - Next 6 inner page tables are not created, the corresponding entries in outer page directory are invalid / null
  - In next inner page table, 1023 invalid entries and one valid PTE containing frame number of stack page
  - Remaining inner page tables not created, corresponding outer page directory entries are invalid

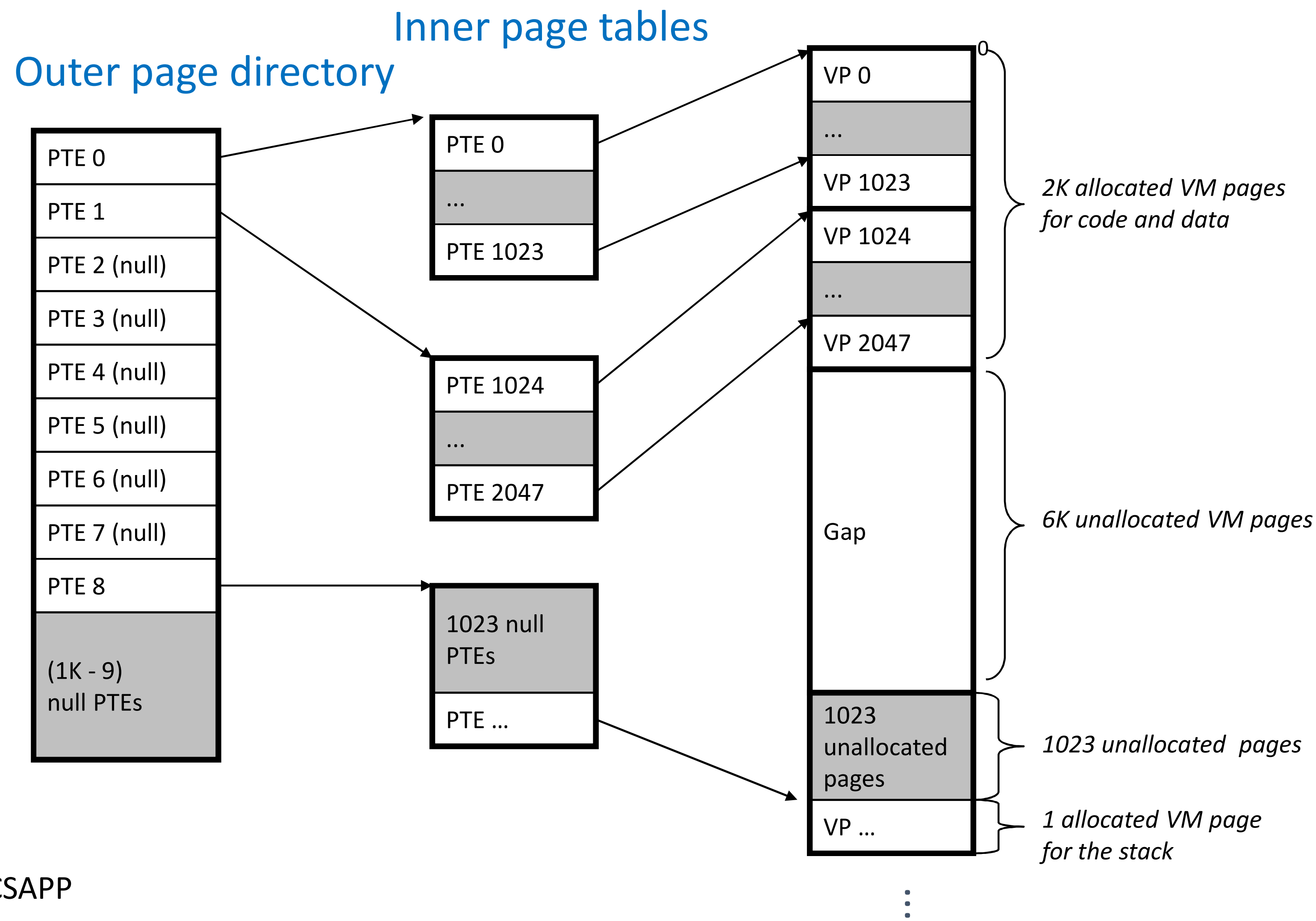


Image credit: CSAPP

Address translation in 2-level page table

- Virtual address of 32 bits = 20 bit page number + 12 bit offset
- 20 bits index into a single page table is now used as
  - Most significant 10 bits index into page directory, locate PTE of one of the 1024 inner page tables contain our desired address
  - Next 10 bits index into inner page table to locate PTE of page
- Locate PTE, computer physical address using frame number and 12-bit offset into page
- MMU "walks" the multiple levels of the page table to translate virtual addresses

Page table/directory base register

- Single level: MMU stores starting address of page table in page table base register
- Multi-level: MMU stores starting address of outer page directory in page directory base register (CR3 register in x86)

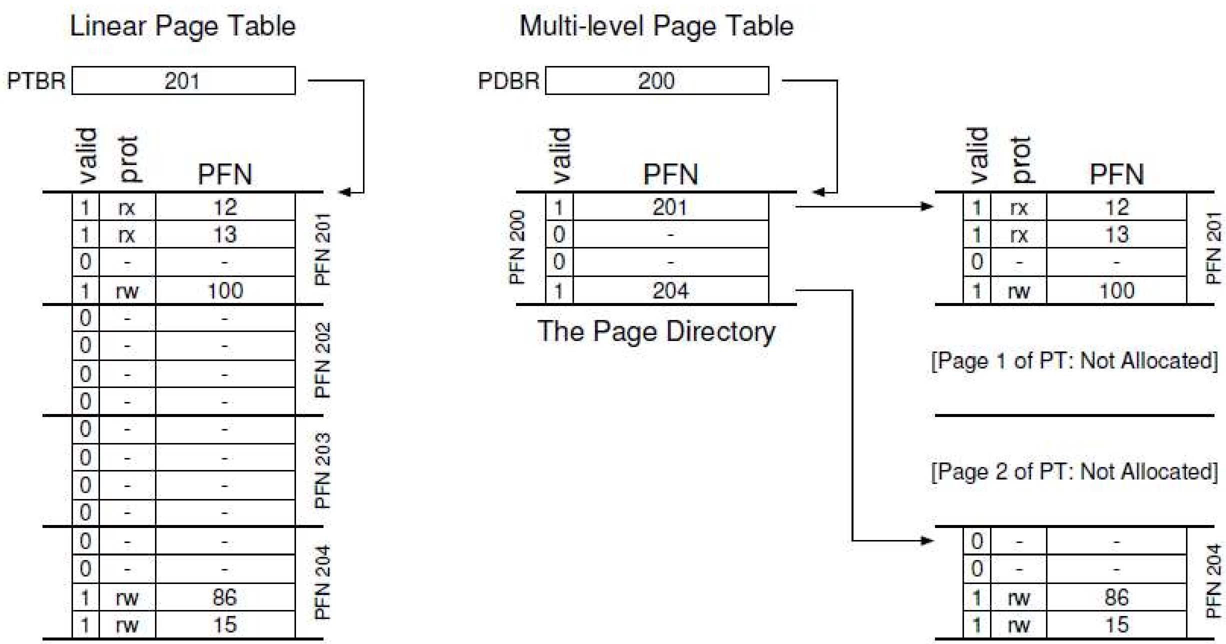


Image credit: OSTEP

Multi-level page tables

- What if outer page directory does not fit into one page?
- Store page directory across many pages, use yet another page table to store frame numbers of page directory pages
- This can go on until outermost page table fits in one page
- Example: 48-bit CPU, 4KB pages, 8 byte page table entries
  - 2^48 bytes in virtual address space = 2^36 pages for each process
  - Each page can store 4KB/8 = 2^9 = 512 page table entries
  - Innermost level (actual page table) has 2^36 page table entries = needs 2^27 pages
  - Innermost page table split into multiple pages = 2^27 page table entries to track innermost page table pages
  - Next level of page table stores 2^27 page table entries = needs 2^18 pages
  - Next level stores 2^18 page table entries = needs 2^9 = 512 pages
  - Outermost level can store all 512 page table entries in 1 page

Address translation with 4-level page table

- Example: 48-bit CPU, 4KB pages, 8 byte page table entries
  - 4 level page table required
  - Outermost page directory has 512 entries, containing frame numbers of next level page table pages, each of those contain frame numbers of next level page table, …
  - Page table at i-th level has frame numbers of 512 (i+1)-th level page table pages
- How to translate VA to PA?
  - 48-bit VA = 36 bits + 12 bit offset
  - 36 bits = 9 bit offset into each of the 4 levels of page table
- If TLB miss, MMU has to access 4 different memory locations for 4 levels of page table, in order to translate one VA to PA
- MMU page table walks become even longer, TLB hit rate is critical

Address translation with multi-level page table

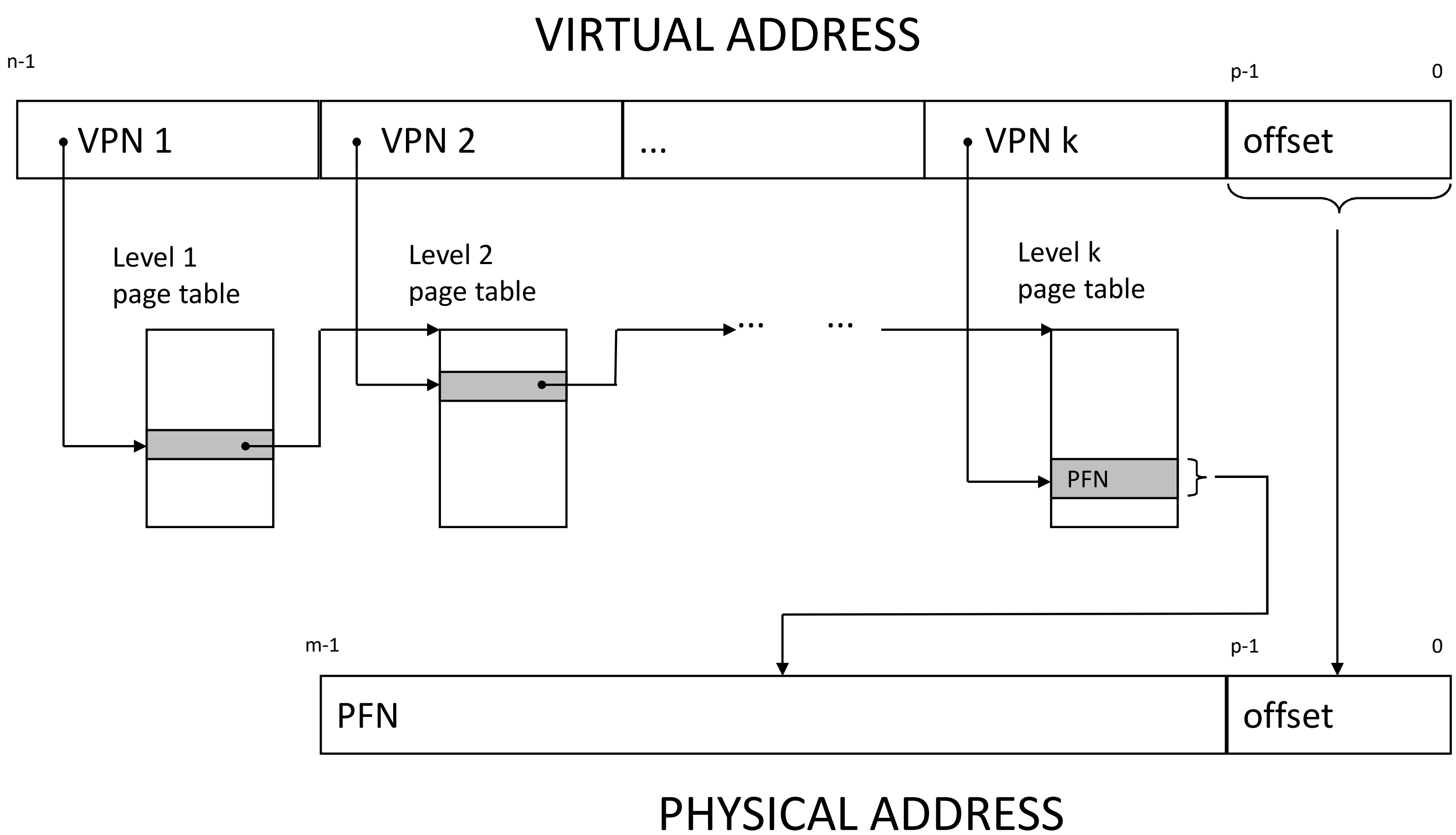


Image credit: CSAPP

Revisiting process virtual address space

- What should virtual address space/page table of process have? Any memory that the process needs to access during its execution
  - Its own memory image: code, data, stack, heap
  - Other common memory it needs to access: shared language libraries, OS
- Why? MMU allows access to memory only via virtual addresses
  - Can only access physical memory mapped in page table at some virtual address
  - So all physical memory needed by process should be mapped into address space
- OS binary image (kernel code, data) is mapped into the virtual address space of every process at addresses not used by process (high VA)
- Why is this done? Easy to jump to OS code during a trap

A subtle point

- OS is not a separate process with its own address space
- Instead, OS code is part of the address space of every process
- A process sees OS as part of its code (e.g., like a library)
- During trap, process jumps to high virtual addresses and executes OS code

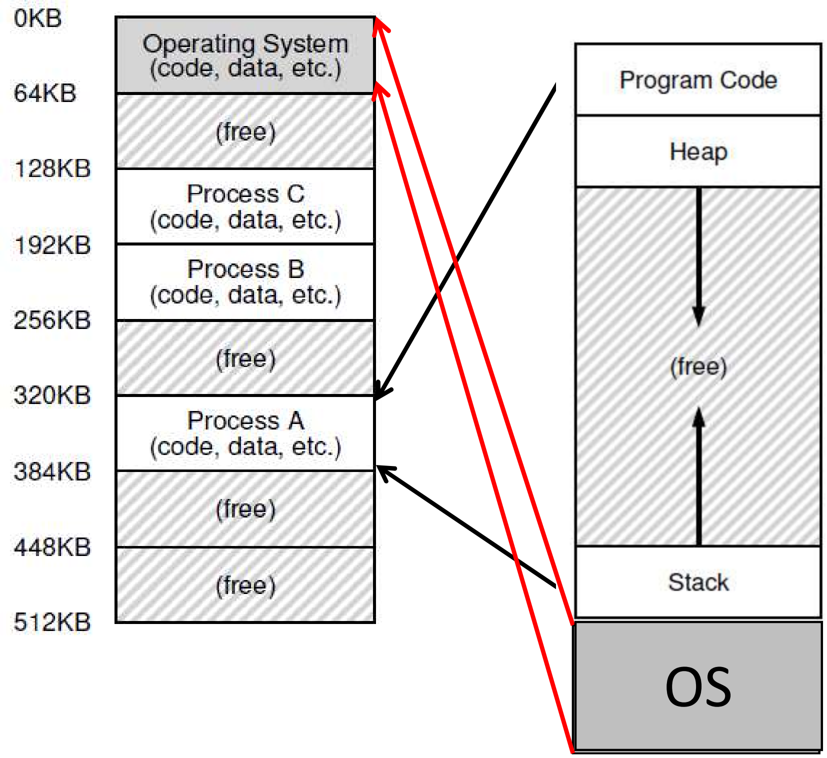


Image credit: OSTEP

OS is part of address space of every process

- OS code/data assigned virtual addresses
  - Compiler ensures high virtual addresses not used by user code
- OS virtual addresses are mapped to physical addresses of OS via page table entries of every process
- There is only one copy of OS code/data in RAM
  - Loaded into RAM at low physical addresses during system bootup
- Page tables of all processes have mappings to same OS physical addresses
  - Same high virtual addresses map to same physical addresses of OS code

Page-level isolation and security

- How is OS code/data protected from illegal access by user?
- Page table has permissions for every memory page
  - Whether read/write or read-only (code pages are read-only)
  - Whether page can be accessed in user mode or kernel mode
- Page table mappings for OS code are protected to allow access only when CPU is in kernel mode
  - CPU in user mode cannot access high virtual addresses of OS code
  - CPU in kernel mode (after trap instruction) can access OS code/data
- MMU traps to OS if any violation detected during memory access, ensures user programs can only access memory they are permitted to access

Example: page-level protection using page tables

- Example: process P1 and P2 each have one read-only page, one read-write page, and one page with OS code accessible in kernel mode

P1

	kernel	READ	WRITE	Address
VP 0:	No	Yes	No	PFN 2
VP 1:	No	Yes	Yes	PFN 4
VP 2:	Yes	Yes	Yes	PFN 0

Page tables with permission bits

P2

	kernel	READ	WRITE	Address
VP 0:	No	Yes	No	PFN 11
VP 1:	No	Yes	Yes	PFN 6
VP 2:	Yes	Yes	Yes	PFN 0

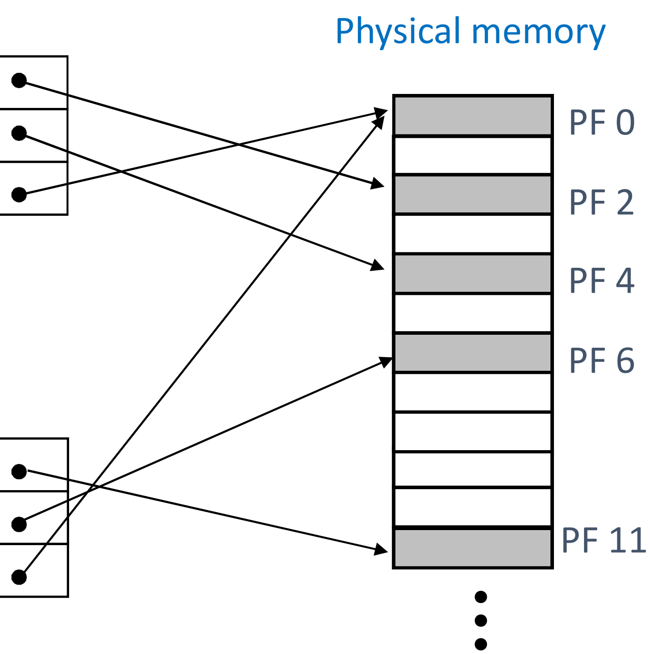


Image credit: CSAPP

Memory management in xv6

- 32-bit OS, so 2^32=4GB virtual address space for every process
- 4KB pages, so 32 bit VA = 20 bit page number + 12 bit offset
- Each PTE has 20 bit physical frame number, and some flags
  - PTE_P indicates if page is valid/present (if not set, access will cause page fault)
  - PTE_W indicates if writeable (if not set, only reading is permitted)
  - PTE_U indicates if user page (if not set, only kernel can access the page)
- Address translation: use page number (top 20 bits of virtual address) to index into page table, find physical frame number, add 12-bit offset

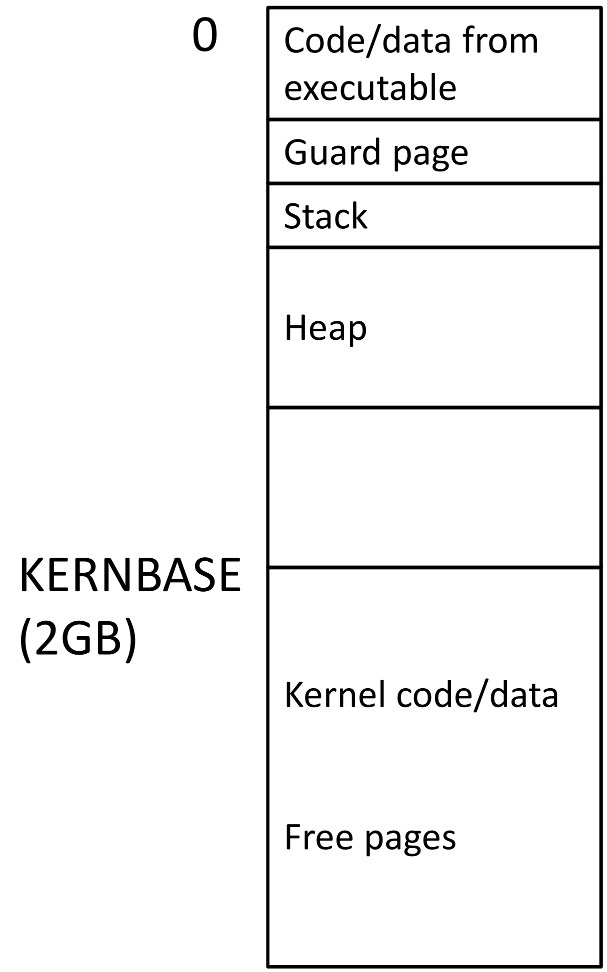
Two level page table in xv6

- xv6 has two-level page table
  - 1024 “inner” page table pages, each with 1024 PTEs
  - Outer page directory stores PTE-like references to 1024 inner page table pages
  - Physical address of outer page directory is stored in CPU’s cr3 register, used by MMU during address translation
- 32 bit virtual address = 10 bits index into page directory, next 10 bits index into inner page table, last 12 bits are offset within page
  - PFN from PTE + offset = physical address

```
// A virtual address 'la' has a three-part structure as follows:
//
// +--------10------+-------10-------+---------12----------+
// | Page Directory |   Page Table   | Offset within Page  |
// |      Index     |      Index     |                     |
// +----------------+----------------+---------------------+
//  \--- PDX(va) --/ \--- PTX(va) --/
```

Virtual address space in xv6

- Virtual address space [0, 4GB]
- Physical address space [0, PHYSTOP] where PHYSTOP is max physical memory that can be used
- Virtual address space contains
  - Low virtual addresses: user code/data, guard page, stack, expandable heap
  - High virtual address starting at KERNBASE (2GB): kernel code/data, free pages that OS assigns to user processes, memory reserved for I/O devices, …



Page table mappings

- Page table contains two sets of PTEs
- User entries: low VA to PA used to process for code, data, stack, heap
- Kernel entries: high VA to PA containing OS code/data/free pages
    - [KERNBASE, KERNBASE+PHYSTOP] mapped to [0, PHYSTOP]
- Kernel page table entries identical across all processes