Virtual Memory: The Page Table

COMP 321

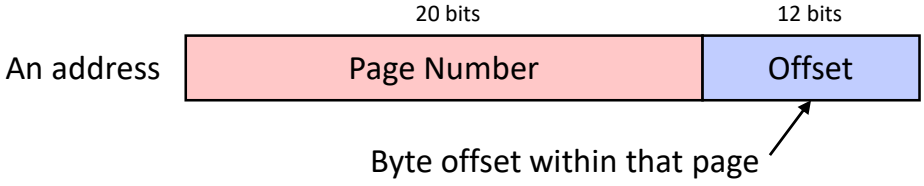
Dave Johnson

1

Review: Virtual Memory Using Paging

An addition to the hardware, managed by the OS kernel

- Conceptually divide memory into fixed-size (power of 2 in size) ***pages***
- Each page is allocated and managed as an indivisible unit
- All pages here are the same size (common is 4096 = 0x1000)

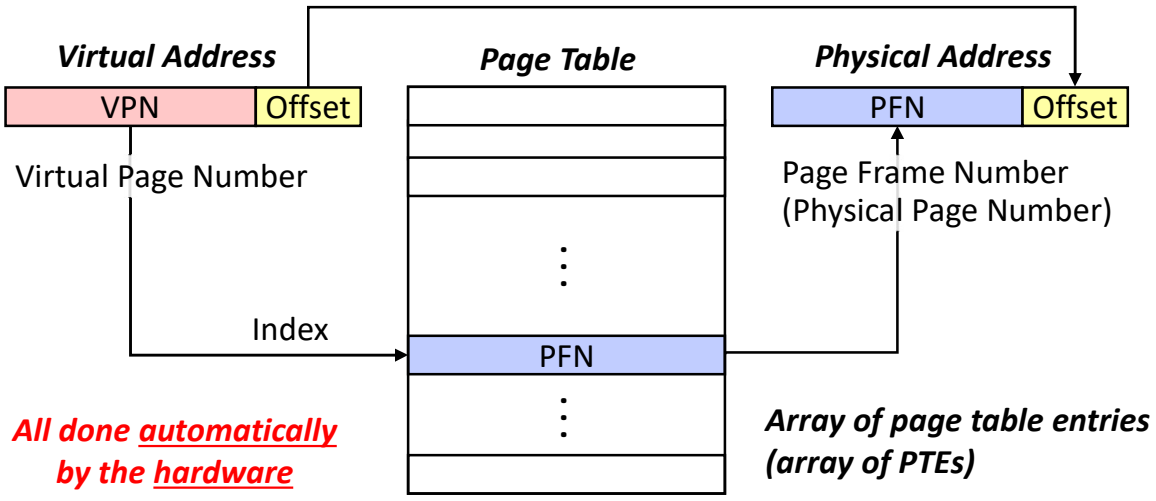


Example: 4096 page size (0x1000)

- Offset is 0…4095 (12 bits for offset)

2

Converting Virtual Address to Physical Address



Page Table Entry Format

Each PTE contains a number of separate fields

- Valid (V): If 0, no access to page is allowed (page "doesn't exist")
  - Hardware causes an exception on attempt to access that virtual page
- Protection (prot): Usually ***separate*** bits for ***user*** vs. ***kernel*** mode – example:
  - R (0x1 = 001): ***Reading*** the contents of the page is allowed
  - W (0x2 = 010): ***Writing*** on the contents of the page is allowed
  - X (0x4 = 100): ***Executing*** the contents of the page is allowed
- Page Frame Number (PFN): The physical page this virtual page is mapped to

The TLB: A Cache of Page Table Entries

A fixed number of entries, built into the hardware

V	VPN	ASID	PTE
V	VPN	ASID	PTE
V	VPN	ASID	PTE
V	VPN	ASID	PTE

- If the "V" (Valid) bit is set, that TLB entry is "valid" (not "empty")
- The ***hardware*** searches all TLB entries for a valid entry (V = 1) with the needed VPN and ASID
- If not found, the ***hardware*** loads the needed PTE from ***physical*** memory into the TLB, caching it and replacing some other TLB entry if needed

The "Contract" between Hardware and Software

Both must know the <u>meaning</u> of the <u>bits</u> in an individual PTE (PTE format)

- The hardware uses the PTE, and the kernel builds (and may modify) any

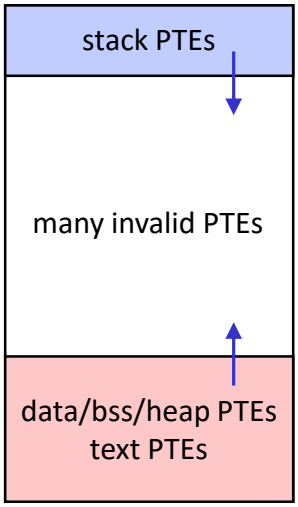
Both must know the <u>location</u> and <u>layout</u> of the page table itself

- The hardware uses the PTEs, and the kernel builds (and may modify) them
- Both must know how to find the PTE for any given virtual page
- With basic paging, each address space's page table is a single array of PTEs
  - PTBR: The ***physical*** memory address of the beginning of that array
  - PTLR: The size of that array (number of entries in that array)

Placing the Stack "Far Away" from the Heap

Need room for them to grow without overlapping

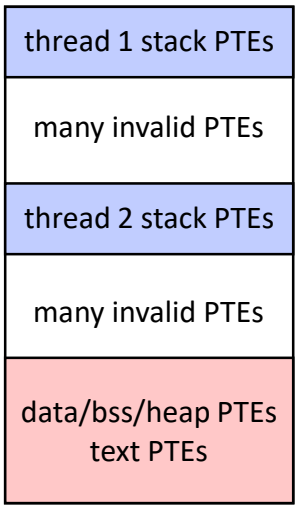
- The heap grows up, and the stack grows down
- Thus, we want to place the heap at low VPNs and the stack at high VPNs
  - Thus, heap PTEs at low index in the page table and the stack PTEs at high index in the page table
  - Room for many pages of growth in between
  - That requires leaving many invalid PTEs in between
- But that means the entire page table must be very large (e.g., 4 bytes per PTE, whether valid or not)



And Each Thread Needs Its Own Stack

Need room for them to grow without overlapping

- Each thread stack independently grows down
- We need to place all of those stacks far from each other and far from the heap
  - That requires leaving many invalid PTEs in between, ***in many places***
- But that means the entire page table must be ***very, very*** large
- ***Same problem for other large data structures that you may want to be able to grow and remain contiguous***



Where Is the Kernel in the Virtual Address Space?

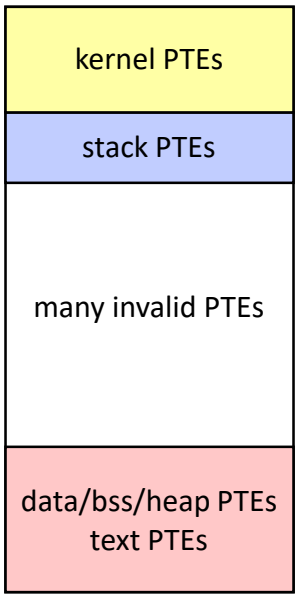
The kernel is <u>somewhere</u> in each process's virtual address space

- On interrupt/exception/trap, hardware does not change PTBR or PTLR
  - Hardware only goes into kernel mode and starts executing instructions where that interrupt vector table entry points to
  - So the correct instructions must be there, and the correct data too
- Consider pointer-based data structures in the kernel (e.g., a linked list)
  - You enter the kernel while running as one process
  - While in the kernel, the kernel context switches to another process
  - All those pointers in all those kernel data structures must still work
- ***Conclusion: The kernel needs to be at the <u>same</u> virtual address in every process's address space***

Building the Page Table for All That

Pick a starting address for the kernel

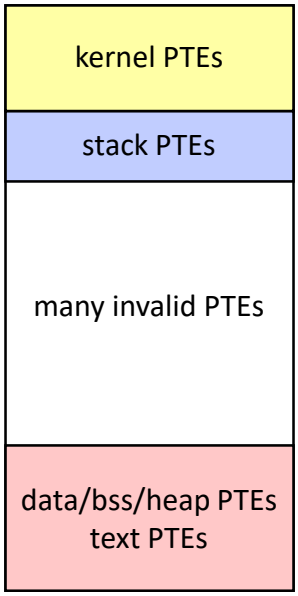
- Build the PTEs for the kernel at that VPN in the page table ***for every process***
- Build the text/data/bss/heap PTEs at the beginning of the page table, as needed for each process
  - Depending on the size of that program
- Build stack PTEs immediately below kernel PTEs
  - No reason to put them at lower VPNs
- ***Now all page tables are the same (very large) size, <u>regardless of the size of the user program</u>***



Example on a 32-Bit Computer

A common layout for 32-bit computers

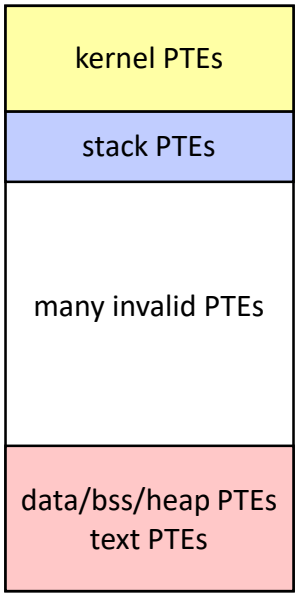
- Addresses 0 < x ≤ 0x7fffffff for the user program, addresses 0x80000000 < x ≤ 0xffffffff for the kernel
  - Thus, 2 GB of address space available for each
- Suppose the page size is 4096
- And suppose the kernel is only 16 kB = 4 pages
- User: 2 GB / 4 kB = 1/2 M pages = 1/2 M PTEs = 2 MB
- Kernel: 4 pages = 4 PTEs = 16 bytes
- Total = (2 M + 16) bytes for the page table, ***regardless of the size of the user program***
- (Only worse on a 64-bit computer)



The Kernel's PTEs Are Duplicated

A copy of the kernel's PTEs must appear in every process's page table

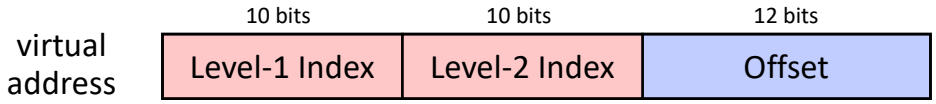
- Each process has its own page table
- The kernel appears at the same virtual address in every process's address space
- So the kernel's PTEs must appear at the same VPNs in every process's page table
  - Wastes memory having so many PTE copies
  - Difficult to keep the copies consistent between all page tables



Common Solution: Tree-Structured Page Table

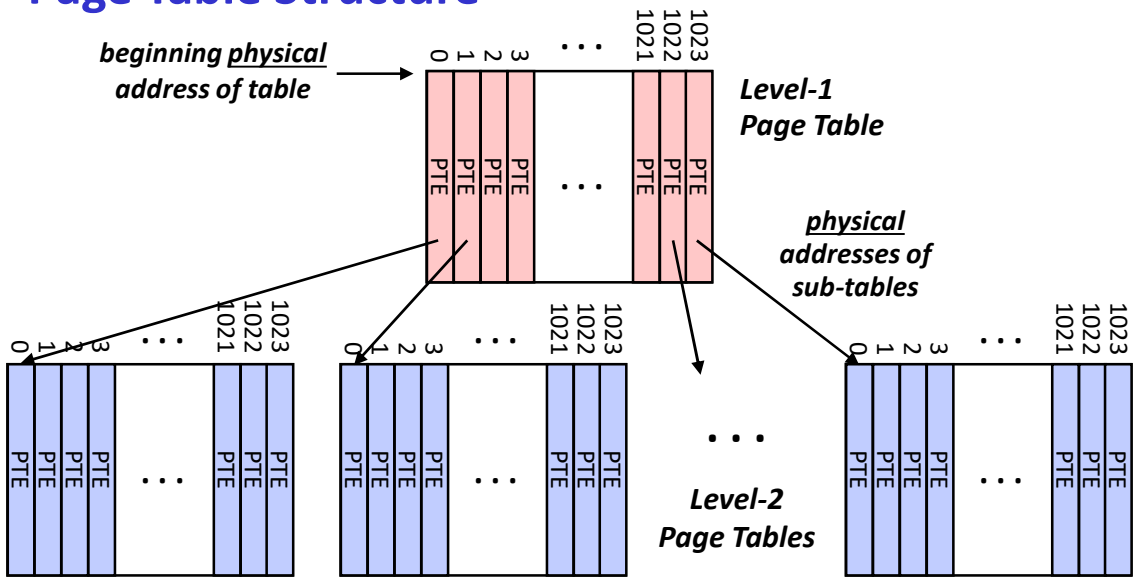
Change <u>hardware</u> to understand the page table as a hierarchy of sub-tables

- Each sub-table is indexed by different bits of the virtual address
- Example: ***32-bit Intel x86*** paging hardware – a two-level tree
  - Top 10 bits of virtual address is the index into the level-1 (top level) table
  - Next 10 bits of virtual address is the index into the level-2 table



- *One special register has **physical address** of top-level table (example Intel CR3)*
- ***This treatment of virtual addresses is built into the hardware***

Page Table Structure



So What Have We Accomplished?

Seems more complex and slower

- More tables for the kernel to build, and more levels of indirection
- 3 physical memory accesses for the hardware for each LOAD/STORE
  - 2 just for the levels of the page table, plus the actual LOAD/STORE
- But the TLB avoids almost all of those extra physical memory accesses

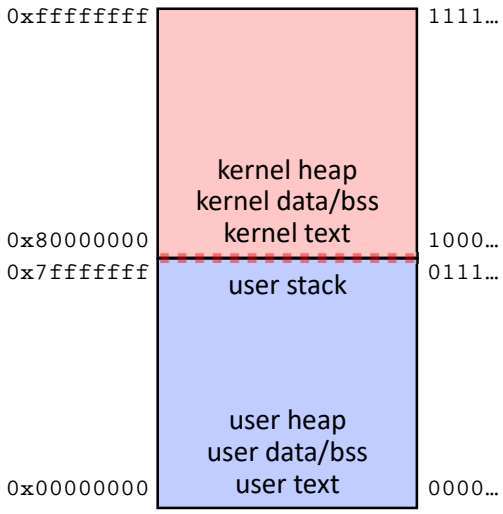
And the (small) complexity has given us many advantages, including

- Page table allocation and management is much easier
  - Only each smaller table must be physically contiguous, not the whole
  - Intel: Each table is 2^{10} entries = 1024 × 4 bytes = 4 kB = PAGESIZE
  - No more fragmentation in allocating or growing a page table
- ***And unused level-2 tables don't even need to physically exist!***

Example Address Space Layout

A very common layout for 32-bits

- Divide the address space in half
  - The bottom half for the user (addresses beginning with `0` bit)
  - The top half for the kernel (addresses beginning with `1` bit)
  - (kernel stack is omitted in this example for simplicity)
- 4 GB total, with 2 GB for each half
- PAGESIZE = 4 kB = 4096



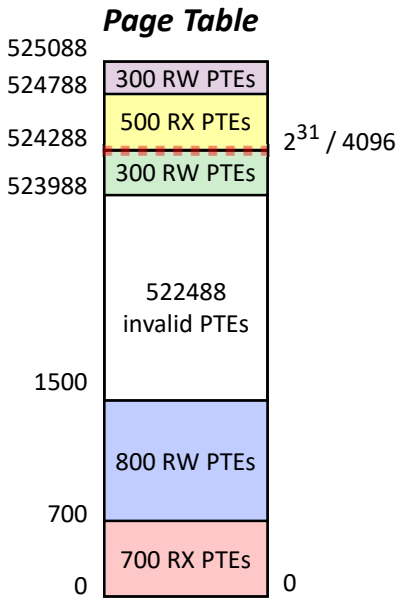
Example Page Table

Consider this using a simple ("basic") page table

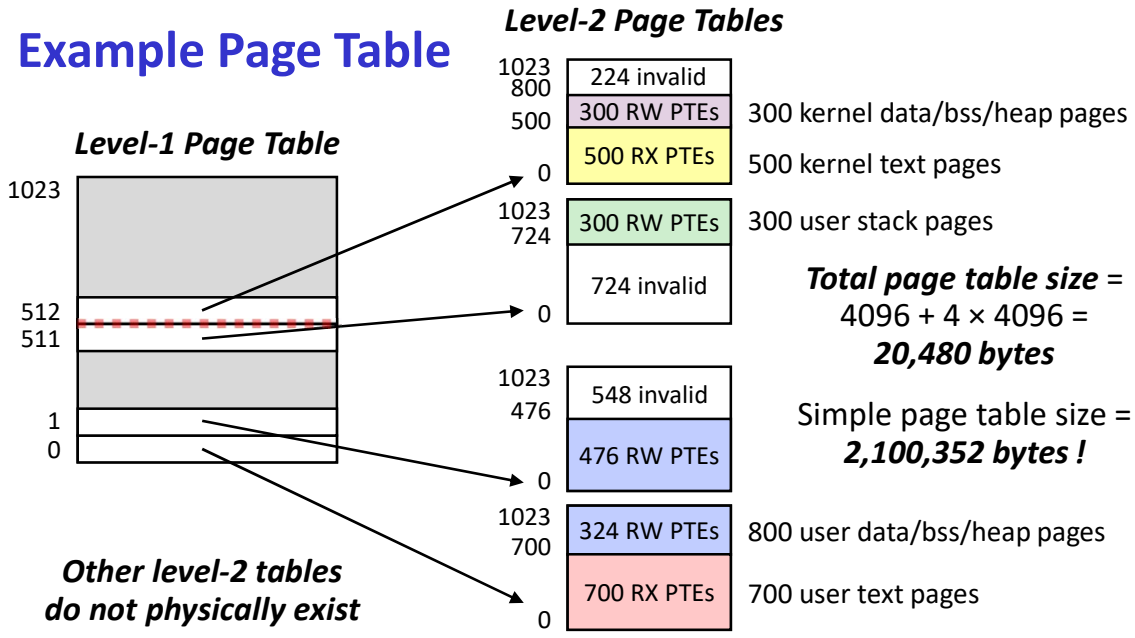
- 700 user text pages
- 800 data/bss/heap pages
- 300 user stack pages
- 500 kernel text pages
- 300 kernel data/bss/heap pages

Page table limit register = 525,088

Page table size = 525,088 × 4 = 2,100,352 bytes



Example Page Table



Total page table size =
4096 + 4 × 4096 =
20,480 bytes

Simple page table size =
2,100,352 bytes !

Logical/Geometric vs. Mathematical Layout

Example: the first byte of the user text

- The first byte in the entire virtual address space
- Means the first PTE in the first level-2 table pointed to by first level-1 entry
- First address 0x00000000: level-1 index = 0, level-2 index = 0

Example: the last byte of the user stack

- The last byte immediately below the halfway point in the virtual address space
- Last PTE in last level-2 table pointed to by the level-1 entry just below middle
- Last address 0x7fffffff: level-1 index = 0x1ff = 511, level-2 index = 0x3ff = 1023

Example: the first byte of the kernel text

- The first byte immediately above the halfway point in the virtual address space
- First PTE in first level-2 table pointed to by the level-1 entry just above middle
- First address 0x80000000: level-1 index = 0x200 = 512, level-2 index = 0

Other Big Advantages

Easy to have shared "sub-trees" if you want to

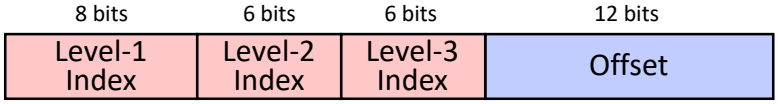
- Example: can put the kernel in every process's address space without duplicating PTEs
  - ***Just duplicate the level-1 pointer(s) to the shared level-2 subtree(s)***
- Can also be used, for example, for shared libraries, or for a block of shared memory between different address spaces, etc.

Can efficiently have almost arbitrary "sparse" address spaces

- Putting the stack far away from the heap, putting the stacks for different threads far away from each other, etc.
  - ***Any level-1 pointer that is NULL (or valid == 0) means the corresponding level-2 table doesn't exist, as if all those PTEs had valid == 0***

Different Designs for the Tree Hierarchy

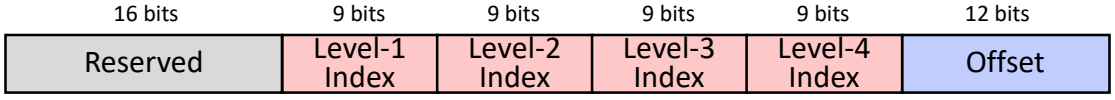
Example: 32-bit SPARC 3-level tree hierarchy



- Each level-1 table holds 2^8 = 256 entries = 1024 bytes
  - Thus can fit 4 level-1 tables in each 4096-byte physical page
- Each level-2 or level-3 table holds 2^6 = 64 entries = 256 bytes
  - Thus can fit 16 level-2 or level-3 tables in each 4096-byte physical page

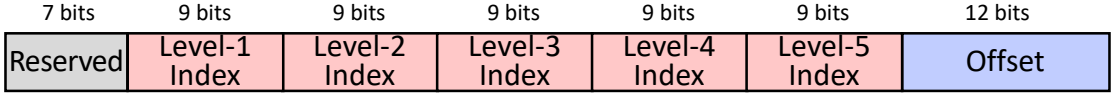
Different Designs for the Tree Hierarchy

Example: 64-bit Intel x86 4-level tree hierarchy (called "IA-32e paging")



- Really only a 48-bit virtual address
- Reserved bits must match the high-order bit of the level-1 index

Example: 64-bit Intel x86 "5-level paging" (vs. new name "4-level paging")



- Extended by another 9-bit index, to now be a 57-bit virtual address
- Newer CPUs default to ***4-level paging***, but kernel can enable ***5-level paging***