File Systems Part 7 Continued

s5fs_get_pframe (1)

```
1 static long s5fs_get_pframe(vnode_t *vnode,
      uint64_t pagenum, long forwrite, pframe_t **pfp) {
2     if (vnode->vn_len <= pagenum * PAGE_SIZE)
3         return -EINVAL;
4     mobj_find_pframe(&vnode->vn_mobj, pagenum, pfp);
5     if (*pfp) {
6         // block is cached
7         (*pfp)->pf_dirty |= forwrite;
8         return 0;
9    }
```


This is the code, from the Weenix stencil, for **s5fs_get_pframe** (it's in kernel/fs/s5fs/s5fs.c). The function is called for a particular page (block) number of a file to get the address of a **pframe** structure (referred to via the **pfp** argument) that holds the contents of the block. It deals with the following cases:

1) the desired block is in the mobj cache
2) the desired block is not in the cache, but resides on disk
3) the block is a sparse block (and thus doesn't reside on disk, but is treated as a block containing all zeroes)

At line 4, **mobj_find_pframe** is called to see if the desired block is cached (in the **mobj** associated with the file's **vnode**). If it's found, we're done and return 0 – we have the block in the cache and ***pfp** is the address of its **pframe** structure. If the intent is to modify the block, we mark it (prematurely) as modified (the **pf_dirty** flag is set to the value of **forwrite**, making it one if our intent is to modify the block). Otherwise, we continue to the next slide.

s5fs_get_pframe (2)

```
10      int new;
11      long loc = s5_file_block_to_disk_block(
            VNODE_TO_S5NODE(vnode), pagenum, forwrite, &new);
12      if (loc < 0) return loc;
13      if (loc) {
14          if (new) {
15              *pfp = s5_cache_and_clear_block(
                    &vnode->vn_mobj, pagenum, loc);
16          } else
17              s5_get_file_disk_block(vnode, pagenum, loc,
                    forwrite, pfp);
18          return 0;
19      }
```


The block did not exist in the cache. Thus we call **s5_file_block_to_disk_block** to find out where it is in the file system (i.e., we convert from a block number relative to the beginning of the file to a block number relative to the beginning of the file system). The function returns the block number (within the file system). If the block is not currently allocated within the file system, then If the **forwrite** flag is set, the block is allocated by the function. But if **forwrite** is not set (the intent is just to read the block), nothing is allocated in the file system. If there was an error (perhaps something is wrong with the disk), a negative value is returned (which is the negated error code).

If a block number is returned, **new** tells us whether it was newly allocated. If so, we add the block to the **mobj** cache and fill it with zeroes. If not (the block previously existed on disk. we call **s5_get_file_disk_block** to read the contents of the block from the file system.

s5fs_get_pframe (3)

```
20      else {
21          KASSERT(!forwrite);
22          return mobj_default_get_pframe(
23              &vnode->vn_mobj, pagenum, forwrite, pfp);
24      }
25  }
```


Finally, if the block is not allocated in the file system, it's a sparse block and we call **mobj_default_get_pframe to** allocate a page frame for the block, fill it with zeroes, and add it to the file's **mobj** cache.

Quiz 1

Suppose a thread does a *read* system call (which calls *s5fs_get_pframe*) to read a portion of a block that is sparse. It then writes data to the block, using the *write* system call. Will, as part of handling this *write*, *mobj_default_get_pframe* be called?

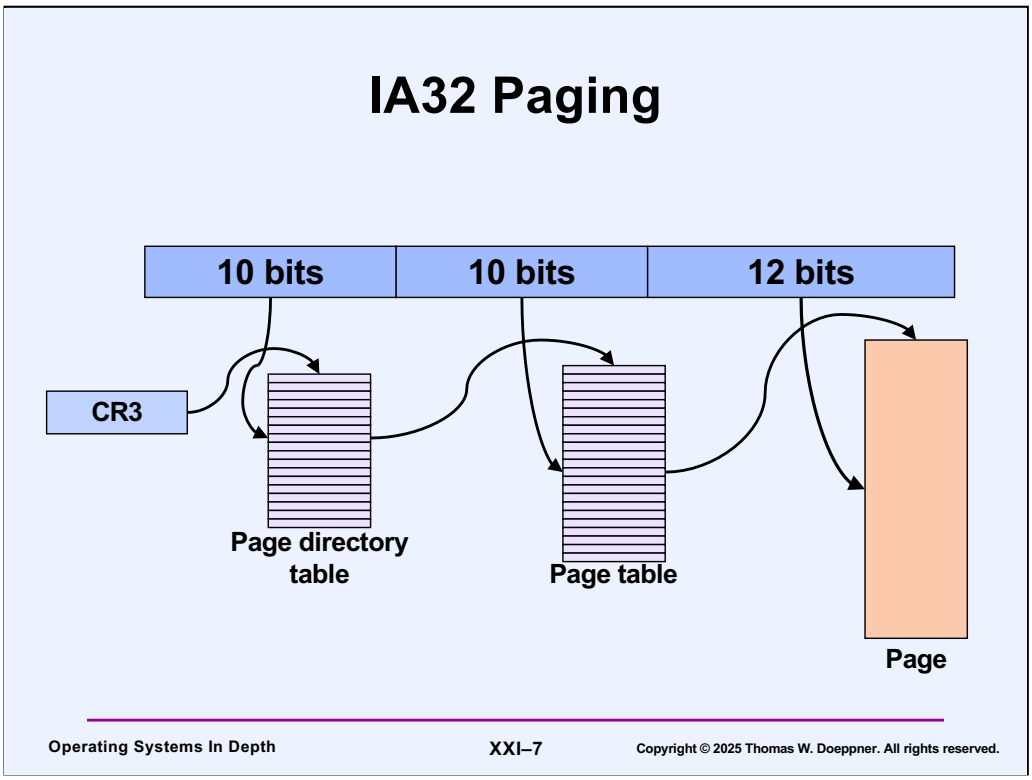
a) **no, because the block was originally a sparse block**

b) **no, the block doesn't need to be zeroed and the caller of *s5fs_get_pframe* will have put data into it**

c) **yes, since the block is sparse, *mobj_default_get_pframe* must be called to zero the block, then modify a portion of it**

d) **yes, for some other reason**

Memory Management Part 2



The IA32 architecture employs a two-level page table providing a means for reducing the memory requirements of the address map. The high-order 10 bits of the 32-bit virtual address are an index into what's called the page directory table. Each of its entries refer to a page table, whose entries are indexed by the next 10 bits of the virtual address. Its entries refer to individual pages; the offset within the page is indexed by the low-order 12 bits of the virtual address. The current page directory is pointed to by a special register known as CR3 (control register 3), whose contents may be modified only in privileged mode. The page directory must reside in real memory when the address space is in use, but it is relatively small (1024 4-byte entries: it's exactly one page in length). Though there are potentially a large number of page tables, only those needed to satisfy current references must be in memory at once.

Quiz 2

Suppose a process on an IA32 has exactly one page residing in real memory. What is the total number of combined pages of page-directory table and page tables required to map this page?

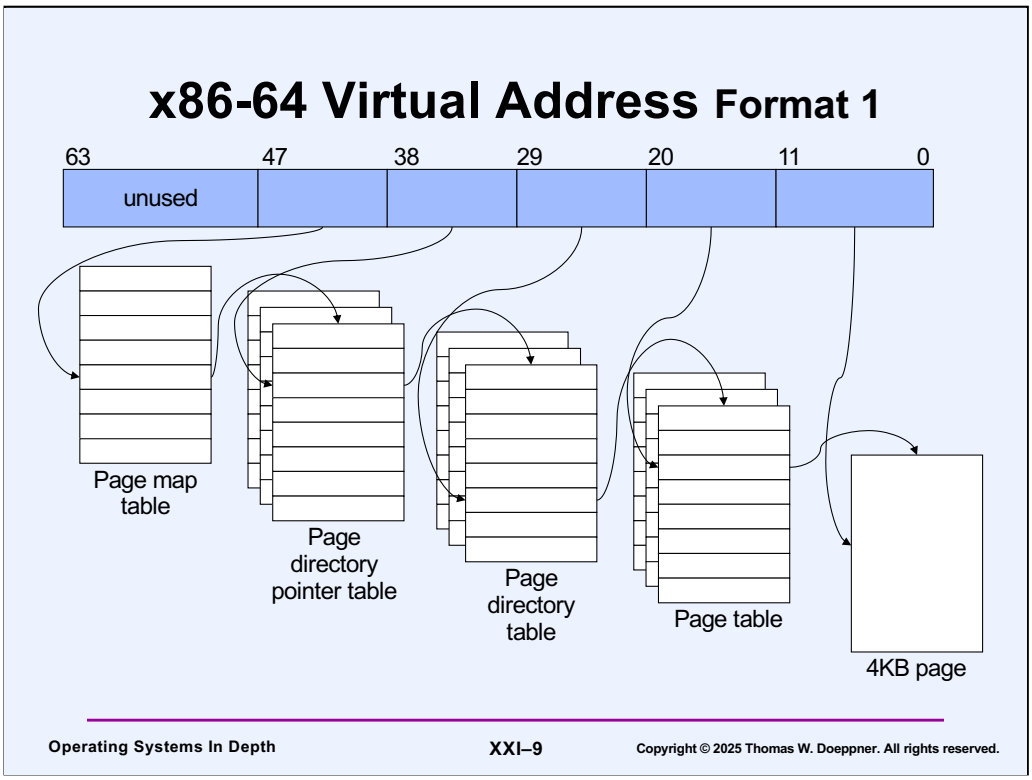
a) 1

b) 2

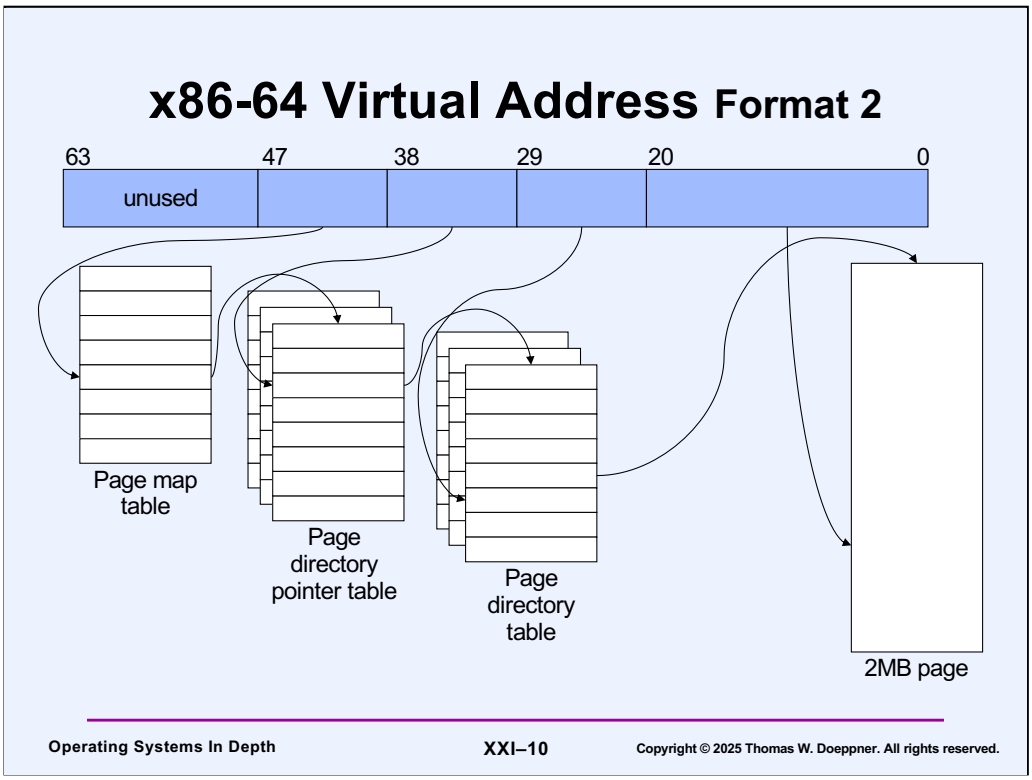
c) 4

d) 8

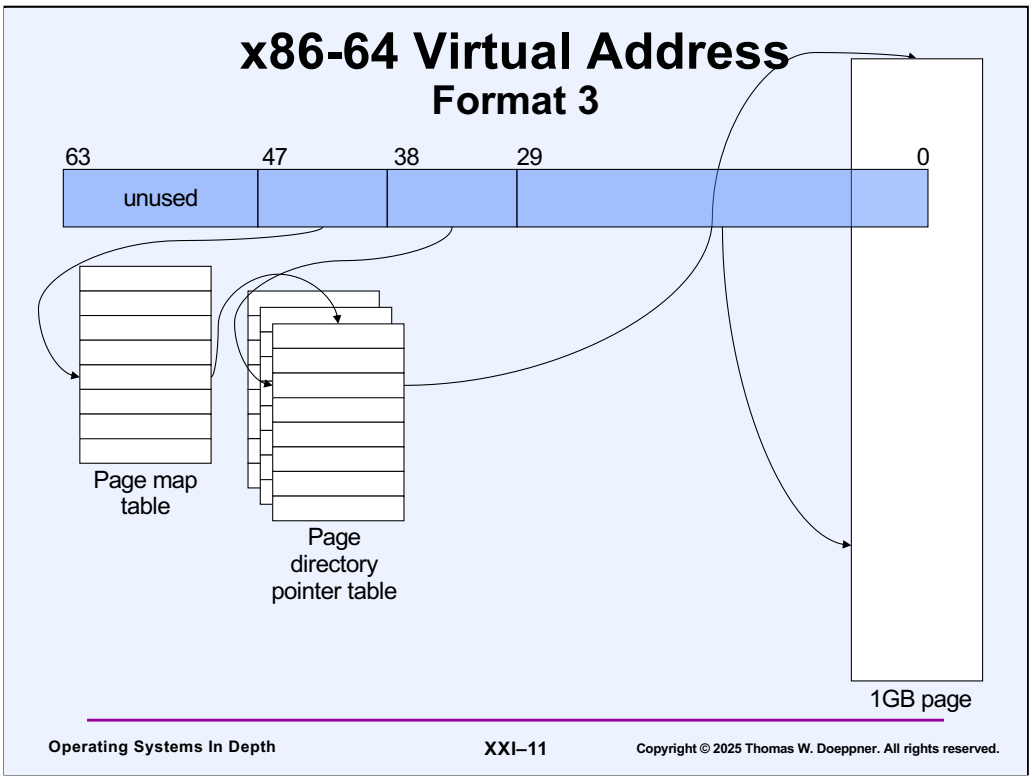
Note we're not counting the page that's mapped, only the page-directory and page table used to map it.



For the x86-64, four levels of translation are done (the high-order 16 bits of the address are not currently used: the hardware requires that these 16 bits must all be equal to bit 47), thus it really supports “only” a 48-bit address space. Note that only the “page map table” must reside in real memory at all times. The other tables must be resident only when necessary.



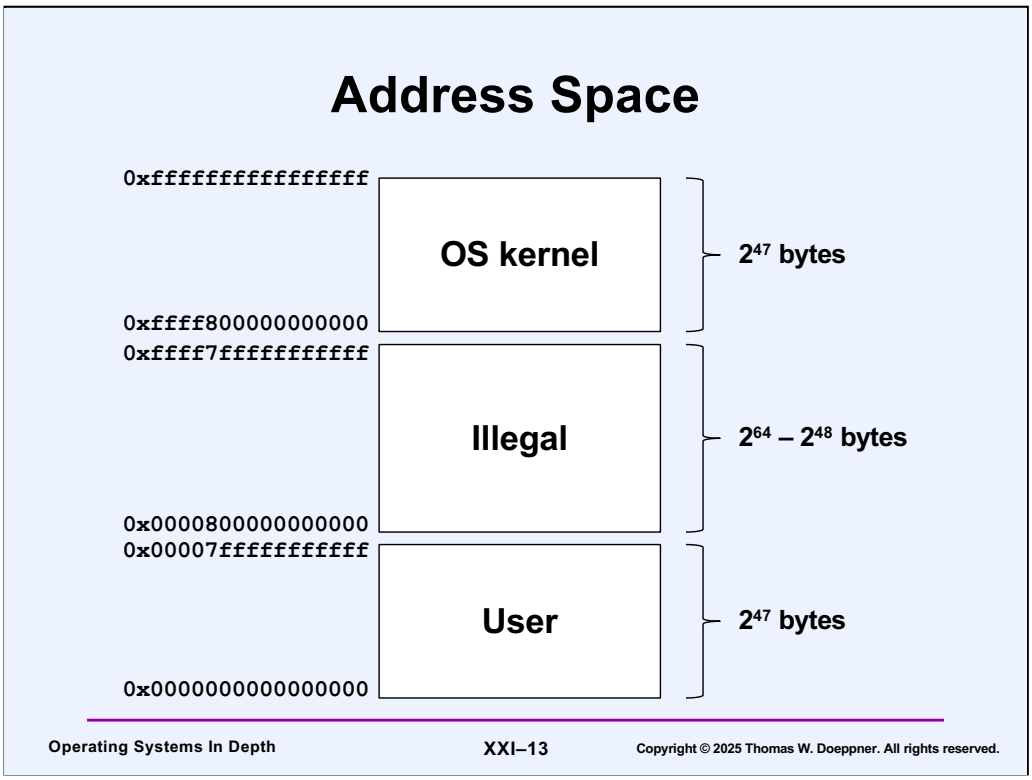
Alternatively, there may be only three levels of page tables, ending with the page-directory table and 2MB pages. Both 2MB and 4KB pages may coexist in the same address space; which is being used is indicated in the associated page-directory-table entry.



The hardware also supports 1 GB pages by eliminating the page-directory table. Not many operating systems (if any) take advantage of this yet.

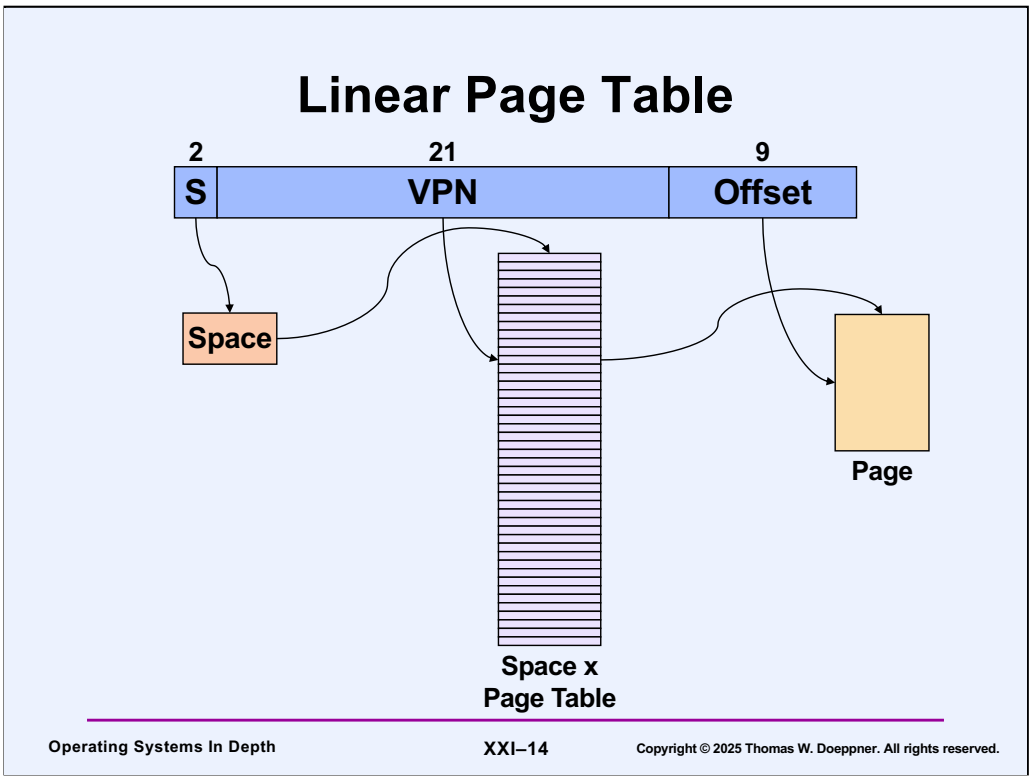
Why Multiple Page Sizes?

- Internal fragmentation
  - for region composed of 4KB pages, average internal fragmentation is 2KB
  - for region composed of 1GB pages, average internal fragmentation is 512MB
- Page-table overhead
  - larger page sizes have fewer page tables
    - less overhead in representing mappings
      - both in memory and in cache

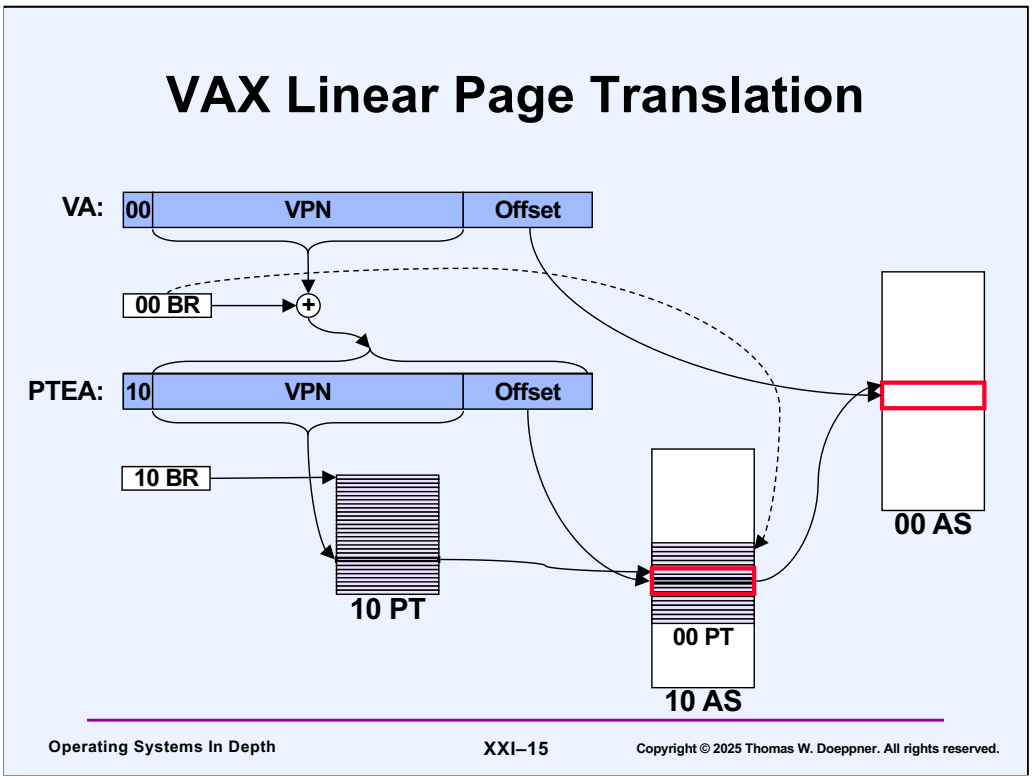


Recall that, in current implementations of the x86-64 architecture, only 48 bits of virtual address are used. Furthermore, the high-order 16 bits must be equal to bit 47. Thus the legal addresses are those at the top and at the bottom of the address space. The top addresses are used for the OS kernel, and thus mapped into all processes. The bottom address are used for each user process. The addresses in the middle (most of the address space—the slide is not drawn to scale!) are illegal and generate faults if used.

The reason for doing things this way (i.e., for the restrictions on the high-order bits) is to force the kernel to be at the top of the address space, allowing growth of the user portion as more virtual-address bits are supported.

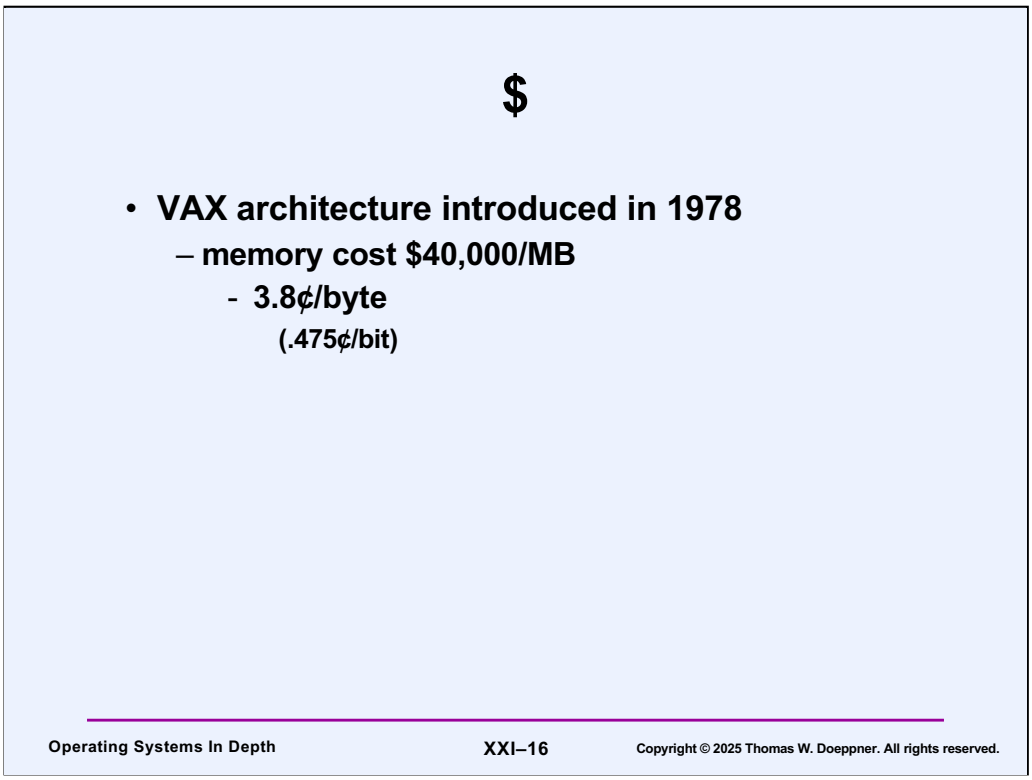


A linear page table is, ostensibly, a single-level page table mapping virtual memory to real memory. However, the page table itself may reside in virtual memory and thus may, in turn, be mapped by another linear page table (presumably one that resides directly in real memory). In the system illustrated in the slide, there are four separate "spaces". The page table mapping space 10 (whose high-order address bits are 10) resides in real memory. But the page tables mapping the other spaces reside in space 10.



In the VAX architecture there are four spaces, 00, 01, 10, and 11. The user portions of the address space are in spaces 00 and 01, the kernel is in 10 (11 wasn't used). The 00 base register indicates where in 10 space the 00 page table starts. The 10 base register indicates where in real memory the 10 page table starts. Thus mapping a 00 virtual address to real memory involves first determining where the entry of the 00 page table resides in 10 space, then determining where the entry of the 10 page table that maps it resides in real memory.

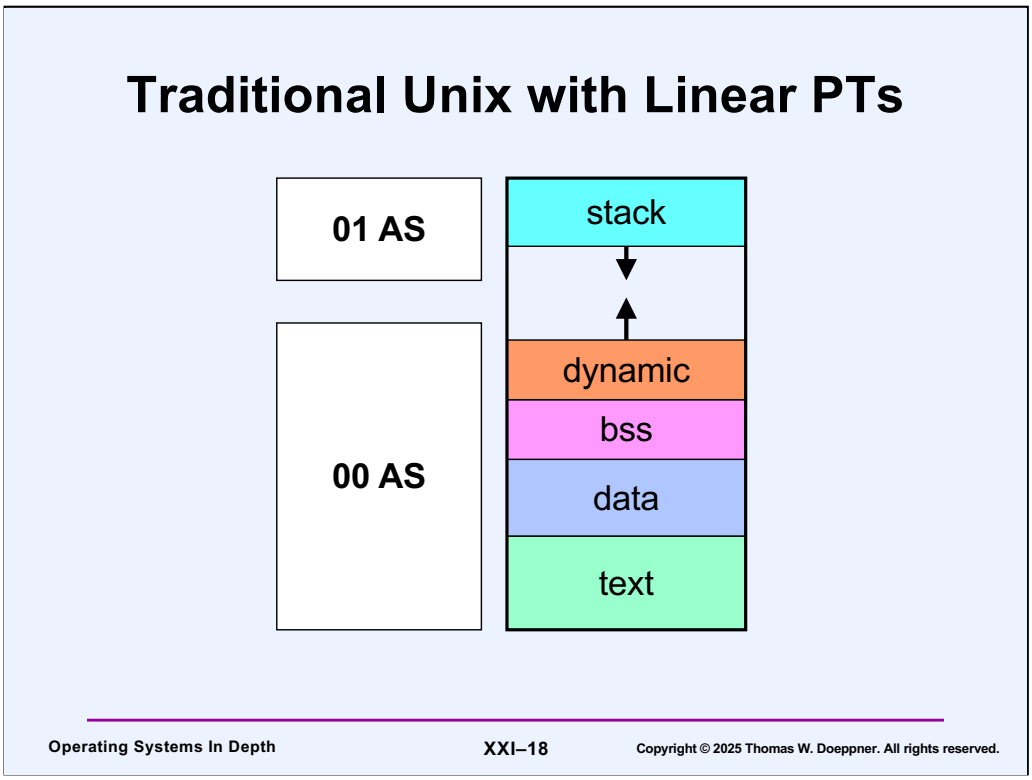
Note: PTEA stands for **page-table entry address**.



In 2025 dollars, this is equivalent to $154,209.22. (From https://www.in2013dollars.com/us/inflation/1978?amount=1)

Linear Page-Table Management

- 00 and 01 page tables each require contiguous locations in 10 space
  - with 512-byte pages, 8MB each:
    - maximum of 128 such page tables
    - (need room for other things, e.g. OS)
- Reduce size requirements with partial page tables
  - length registers constrain size of each space



The address-space requirements of traditional Unix work well with linear page tables with length constraints.

Quiz 3

Suppose the page size is 512 bytes (2^9) and each page-table entry requires 4 bytes. How many pages of page-table entries are required to map 1 megabyte (2^{20}) of address space?

a) 4

b) 8

c) 16

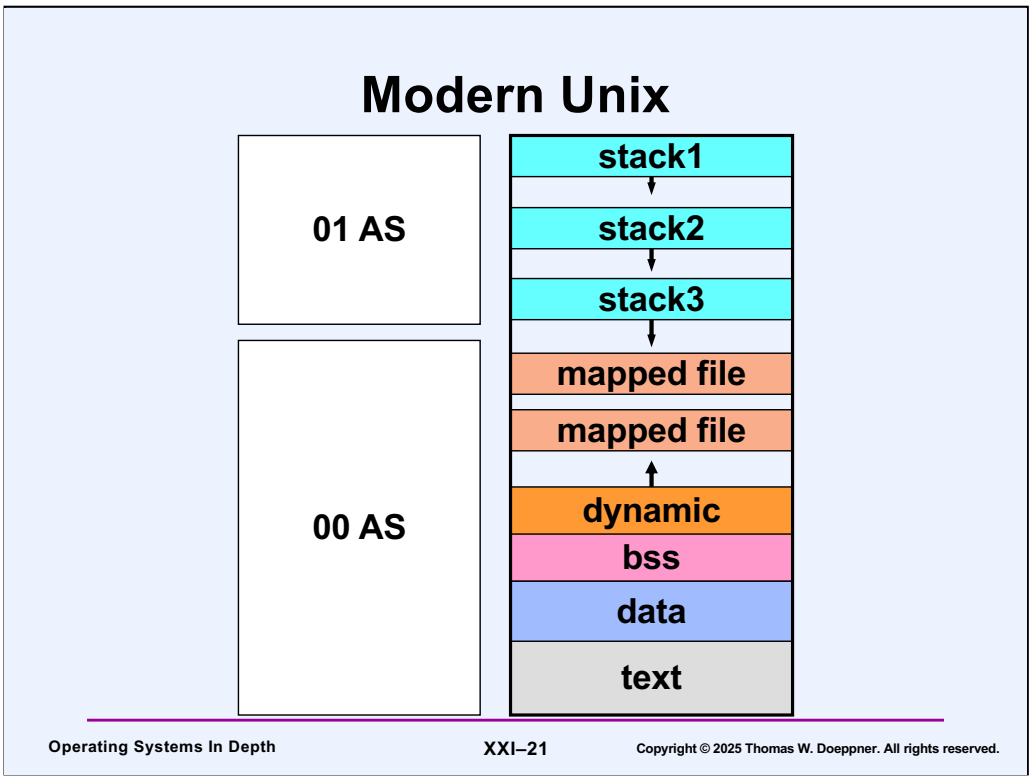
d) 32

In this problem we're not concerned about the pages of real memory required to map the 10 pages holding the 00 (or 01) page table, but just the number of pages in 10 space required to map 1 MB of 00 (or 01) memory.

$

- Limit size of 00 space to 1 MB
  - requires 16-page 00 page table in 10 virtual memory
    - requires 16 entries in 10 page table
- Same requirements if 01 space limited to 1 MB
- What are real-memory requirements?
  - 10 page table resides in real memory
  - at least one page of real memory must be allocated for each of 00 and 01 page tables
  - minimum real memory is 1152 bytes
    - $43.95 in 1978

To represent 1 MB of 00 space and 1 MB of 10 space, a total of 32 entries of 10 page table are required, occupying 128 bytes. If we have one page each for the 00 and 01 page tables, that's 1024 bytes, for a total of 1152 bytes.



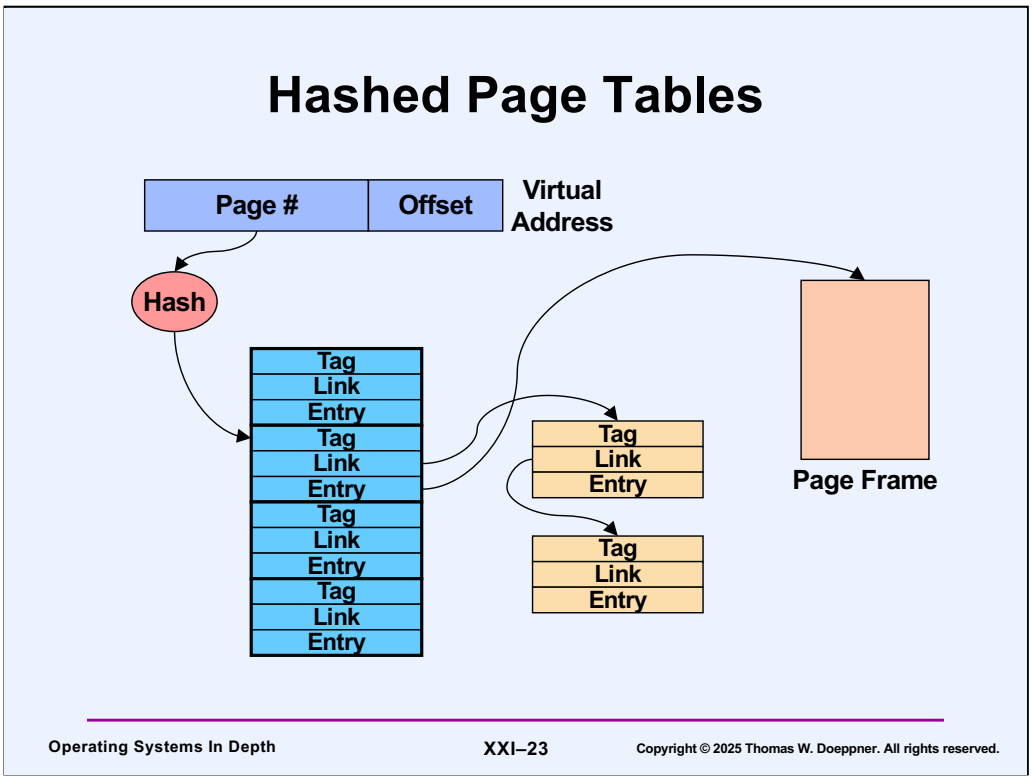
Modern Unix systems make extensive use of mapped files, requiring a number of regions in the address space. Thus two length registers wouldn't be all that effective in reducing the space required of page tables.

$

- **Requires sufficient 10 page-table entries to map almost all of 00 and 01 space**
  - **2^{14} 10 page-table entries for each space**
    - **requiring 64KB each, 128KB total**
    - **$5000 in 1978**
    - **<1¢/process today**
      - **who cares?**
      - **increase address space from 2^{32} to 2^{64}**
        - **4,294,967,296-fold increase**
          - **significant …**

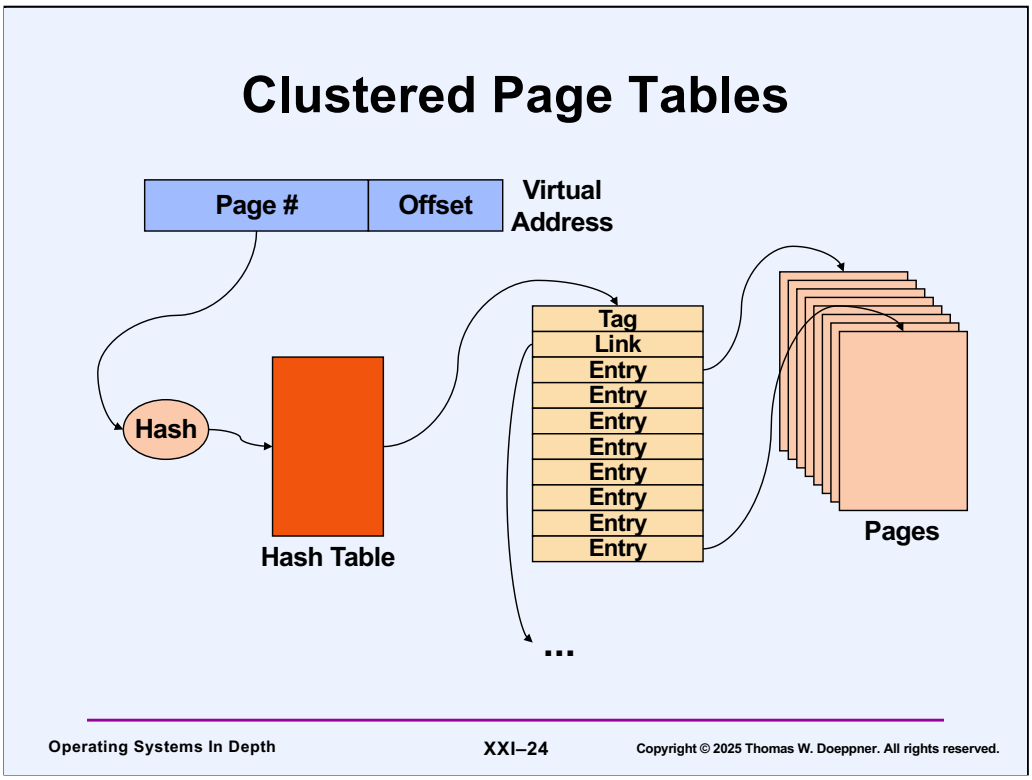
What we're concerned about here is the amount of dedicated real memory required to represent the 00 and 10 spaces: this is the real memory required simply to map the 00 and 01 page tables into real memory. It doesn't include any real memory required to hold the 00 and 01 page tables, and doesn't include the real memory into which the 00 and 01 spaces are mapped.

Each of the 00 and 01 spaces are 2^{30} bytes in length. Thus 2^{21} page-table entries are required for each in 10 space. These 2^{21}-byte page tables require 2^{23} bytes of virtual memory in 10 space, requiring 2^{14} entries of 10 page table in real memory.



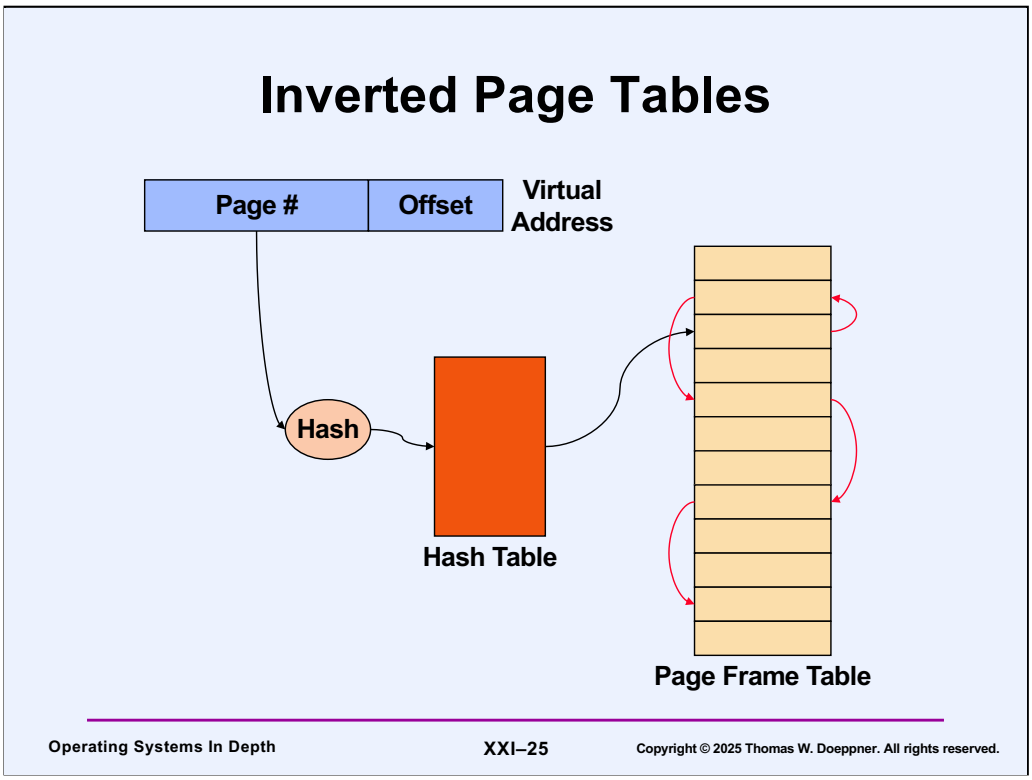
In a **hashed page table**, the page number is a key used as the entry into a hash table. Collisions are handled by chaining. In the form shown in the slide, each page requires three words to represent it.

Hashed page tables support widely but sparsely allocated address spaces well. However, they may require multiple memory accesses for some translations (which can be minimized by using hardware TLBs). Furthermore, a fair amount of extra space is required for chaining the collisions.



A variation of hashed paging that shows promise for supporting 64-bit architectures well is **clustered paging**. In this scheme, a number of pages (perhaps sixteen) are handled by each entry in the lists of hash synonyms. Thus there are three words of overhead per sixteen pages, rather than per page.

The paper "A New Page Table for 64-bit Address Spaces," by M. Talluri, M. Hill, and Y. Khalidi, **Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles**, December 1995, describes and analyzes this scheme.



Here, **inverted page tables**, a variant of hashed page tables, are used to avoid wasting a large amount of memory for the translation map. A **page frame table** is maintained that indicates for each page frame of real memory what virtual address is mapped into it. In a typical implementation (we describe here a simplification of the IBM RS/6000 scheme), the hardware takes the page number from the virtual address, hashes this into a hash table, and then follows a linked list of hash synonyms in the page-frame table until it finds the desired entry. Then the index of this entry (in the page-frame table) is the page-frame number of the page. If the entry is not found, then a page fault is generated.

This procedure would be quite slow if it were always performed exactly as described. However, it can be combined with the use of a TLB to achieve a system that, on the average, performs well.

Another difficulty with inverted page tables is that there are usually portions of several address spaces in primary storage. Thus the virtual address of a page does not identify it uniquely, since different address spaces have pages with identical virtual addresses. So, there must be some sort of **address space ID** to indicate which page is whose. This is accomplished via a hardware register that contains the address space ID of the current address space, and each entry in the page-frame table also contains an address space ID.

Quiz 4

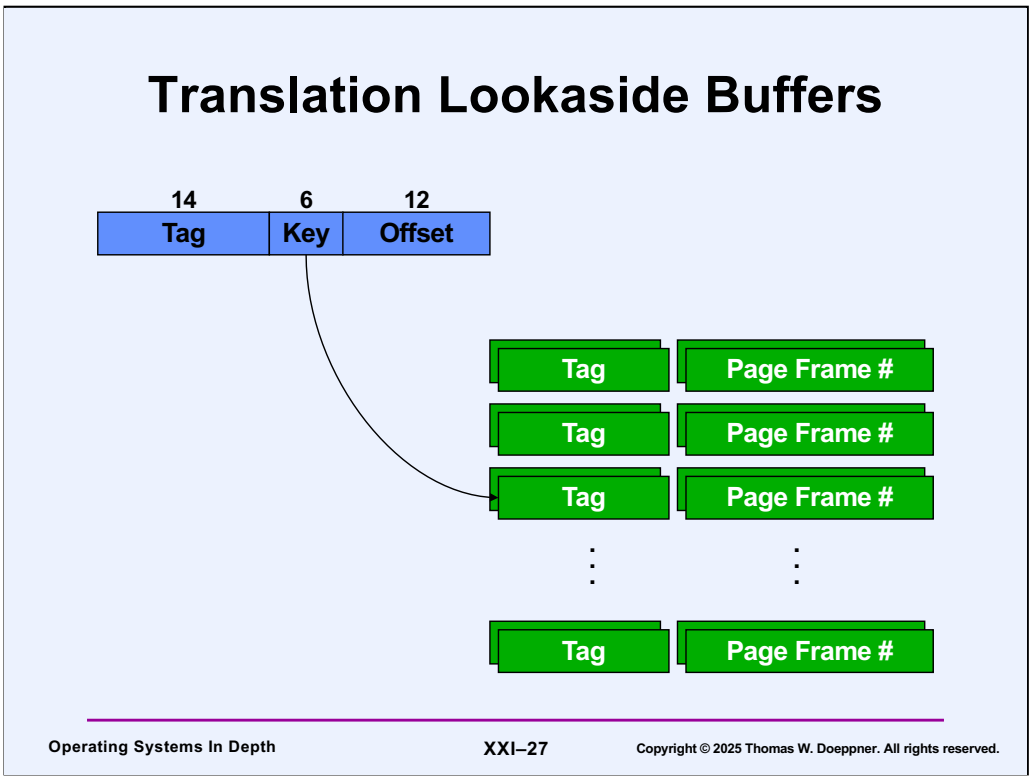
Normal page tables map virtual memory to real memory. More precisely, they map an address space and a location within that address space to real memory. Inverted page tables do the inverse mapping: given an address space ID and a location in real memory, they produce the corresponding virtual location.

a) **Inverted page tables work with all Unix systems**

b) **They don't work with any Unix system**

c) **They don't work with Unix systems that support *mmap* with shared mappings**

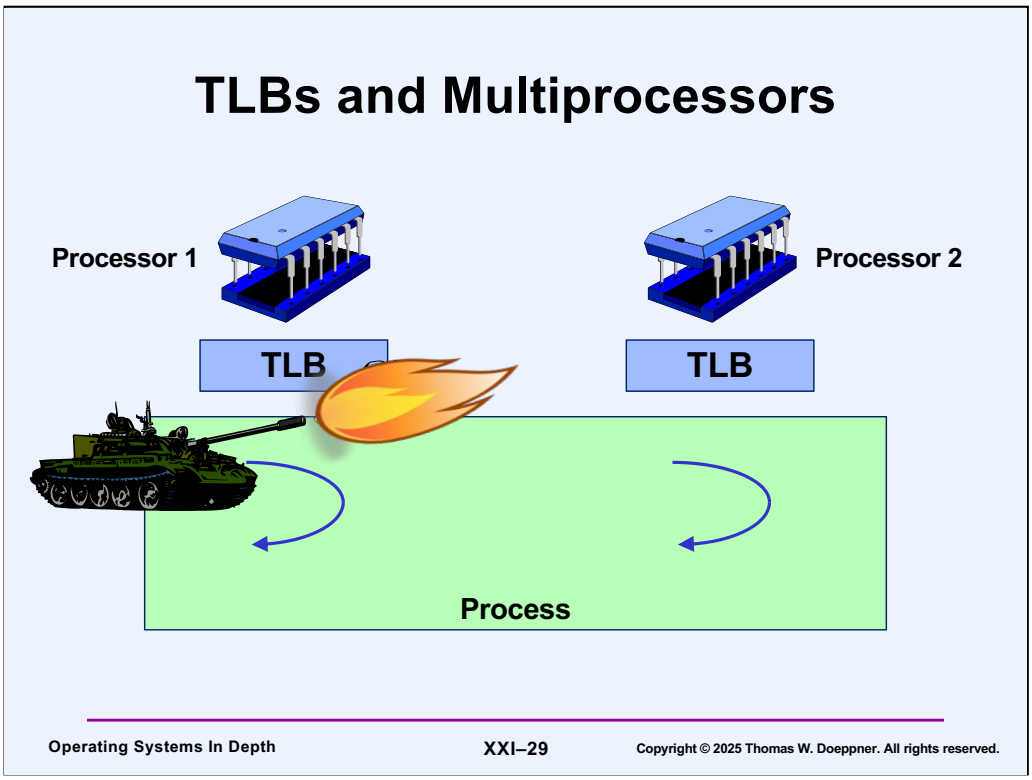
Hint: Do normal page tables (i.e. not inverted) implement a one-to-one function or a many-to-one function?



Some architectures (e.g. MIPS) employ only TLBs for address translation. Thus, if there's a cache miss, the result is a page fault and the OS is called upon to find the page and insert its mapping into the cache.

TLBs and Mappings

- TLBs provide a cache for mappings from virtual addresses to real addresses
- Mappings change when
  - pages are removed (unmapped) from memory
  - when the address space is switched from one process's to another's
- OS must explicitly flush old contents of TLB
  - either individual entries or all of it



For details, see the textbook, page 297.

TLB Shootdown Algorithm

```
// shooter code
for all processors i sharing
  address space
   interrupt(i);
for all processors i sharing
  address space
   while (noted[i] == 0)
      ;
modify_page_table();
update_or_flush_tlb();
done[me] = 1;
```

```
// shootee i interrupt handler
receive_interrupt_from_
  processor j
noted[i] = 1
while (done[j] == 0)
   ;
flush_tlb()
```