File Systems Part 7 Continued

s5fs_get_pframe (1)

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

s5fs_get_pframe (2)

```
    int new;
    long loc = s5_file_block_to_disk_block(
        VNODE_TO_S5NODE(vnode), pagenum, forwrite, &new);
    if (loc < 0) return loc;
    if (loc) {
        if (new) {
            *pfp = s5_cache_and_clear_block(
                &vnode->vn_mobj, pagenum, loc);
        } else
            s5_get_file_disk_block(vnode, pagenum, loc,
                forwrite, pfp);
        return 0;
    }
```

s5fs_get_pframe (3)

```
    else {
        KASSERT(!forwrite);
        return mobj_default_get_pframe(
            &vnode->vn_mobj, pagenum, forwrite, pfp);
    }
}
```

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

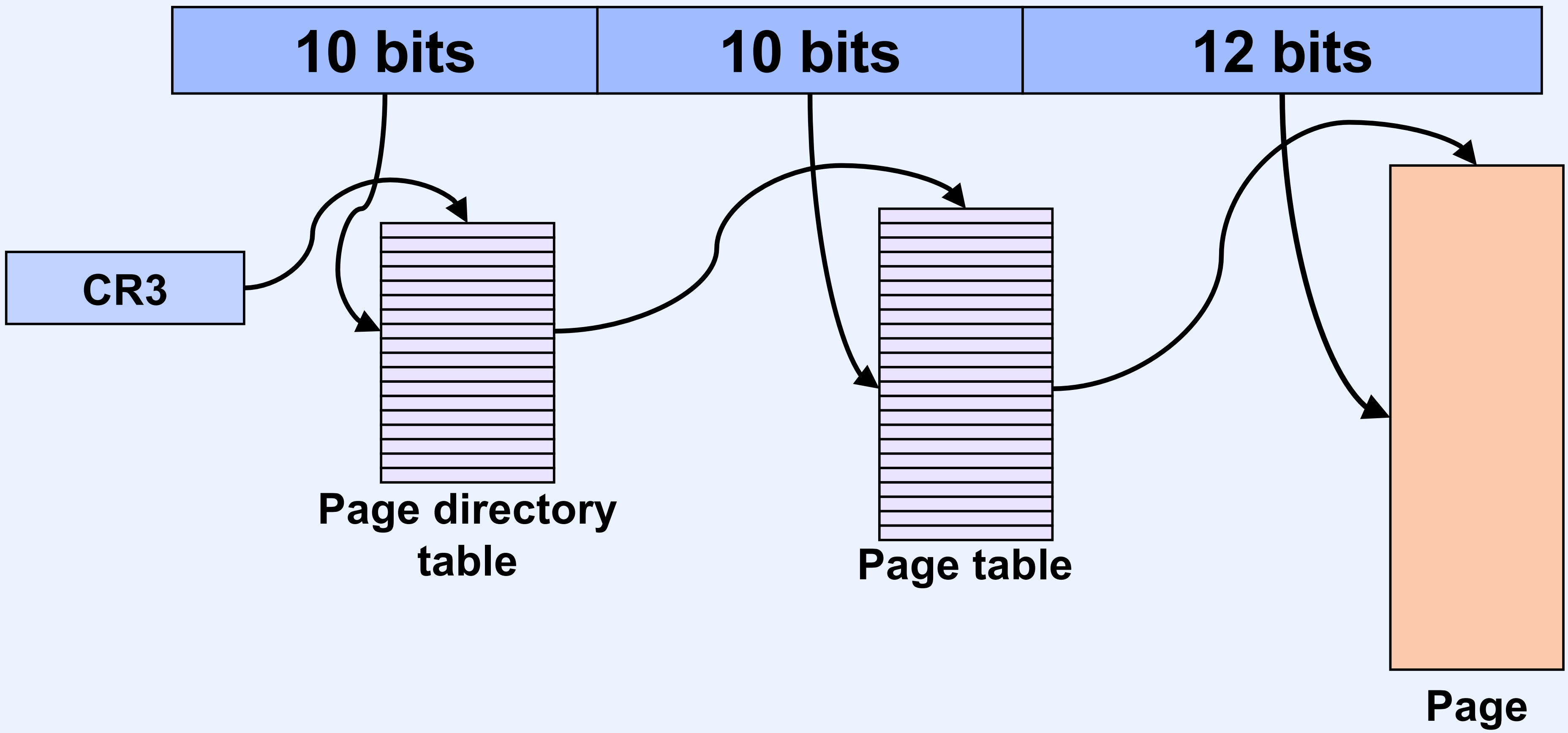
b) no, the block doesn't need to be zeroed and the caller of *s5fs_get_pframe* will have put data into it

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

IA32 Paging


10 bits
10 bits
12 bits
CR3
Page directory table
Page table
Page

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?

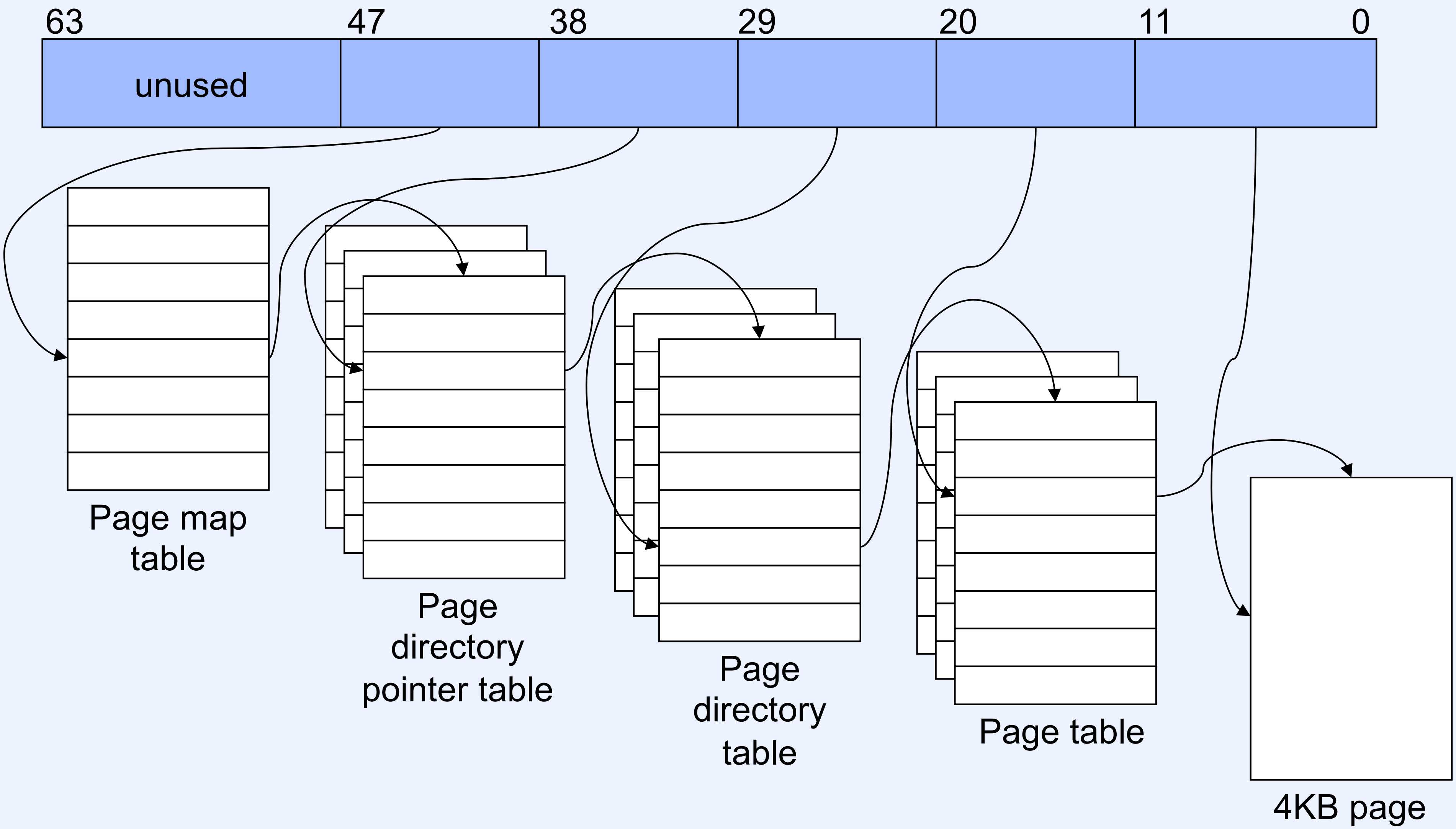
a) 1

b) 2

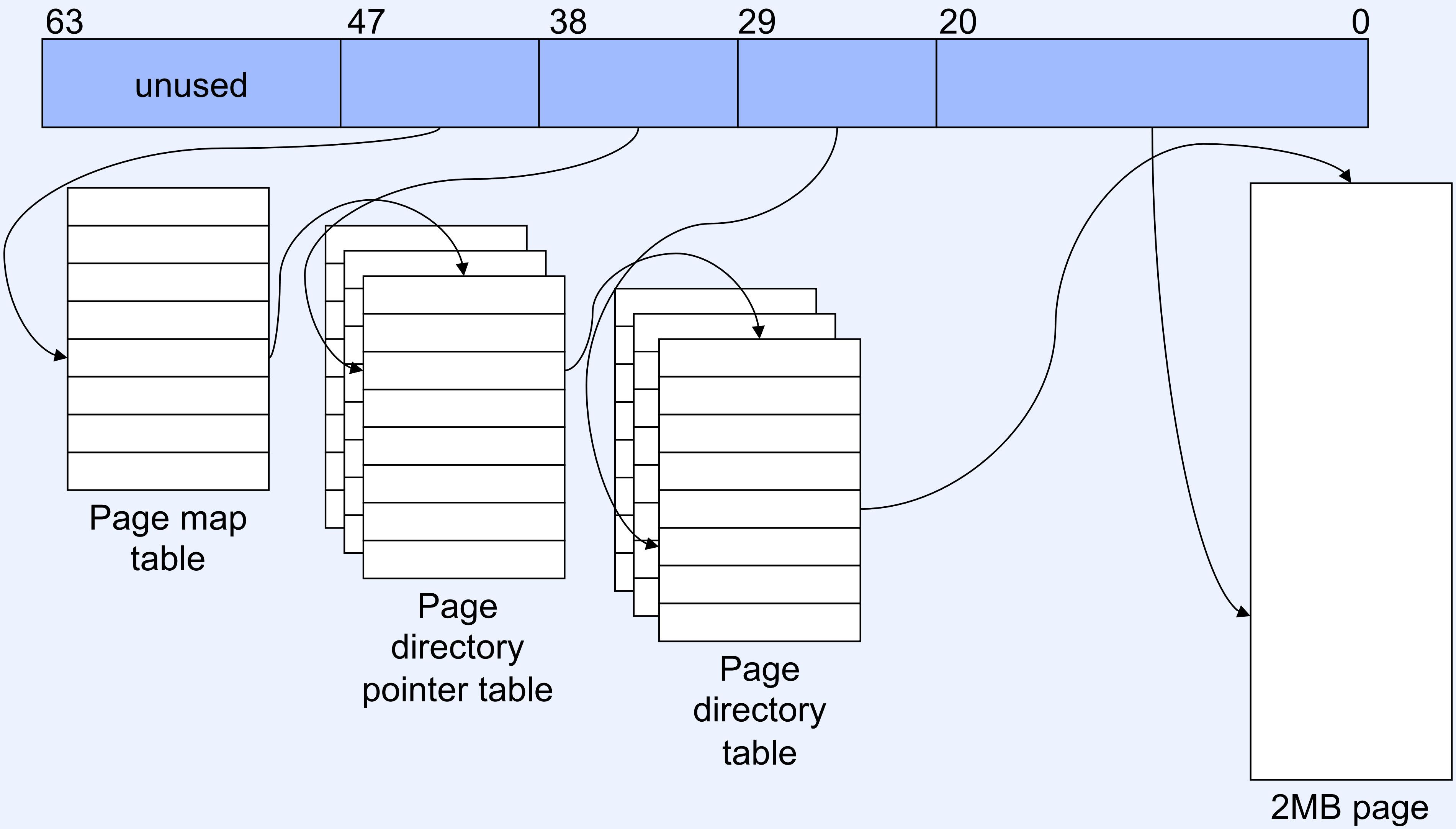
c) 4

d) 8

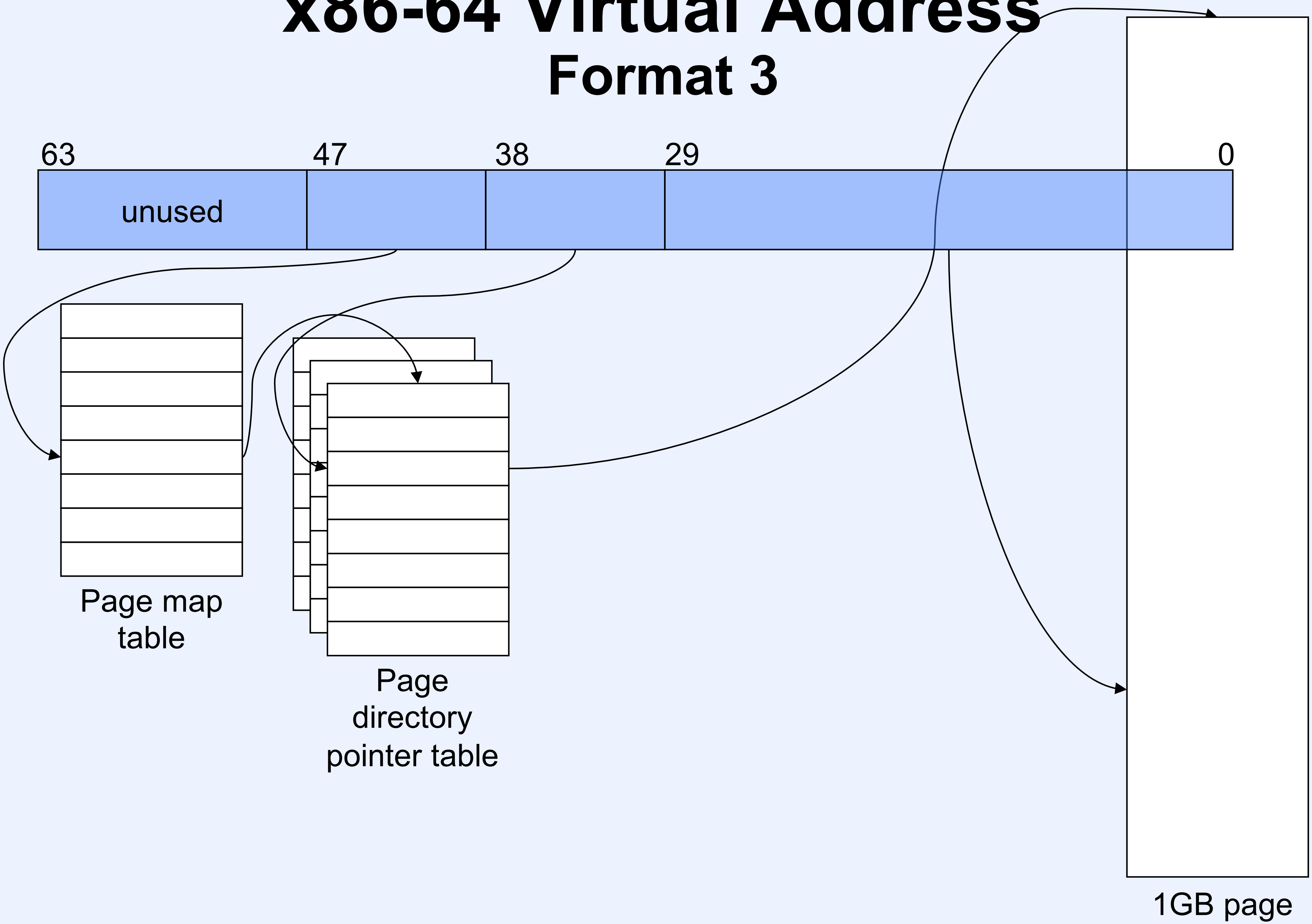
x86-64 Virtual Address Format 1



x86-64 Virtual Address Format 2



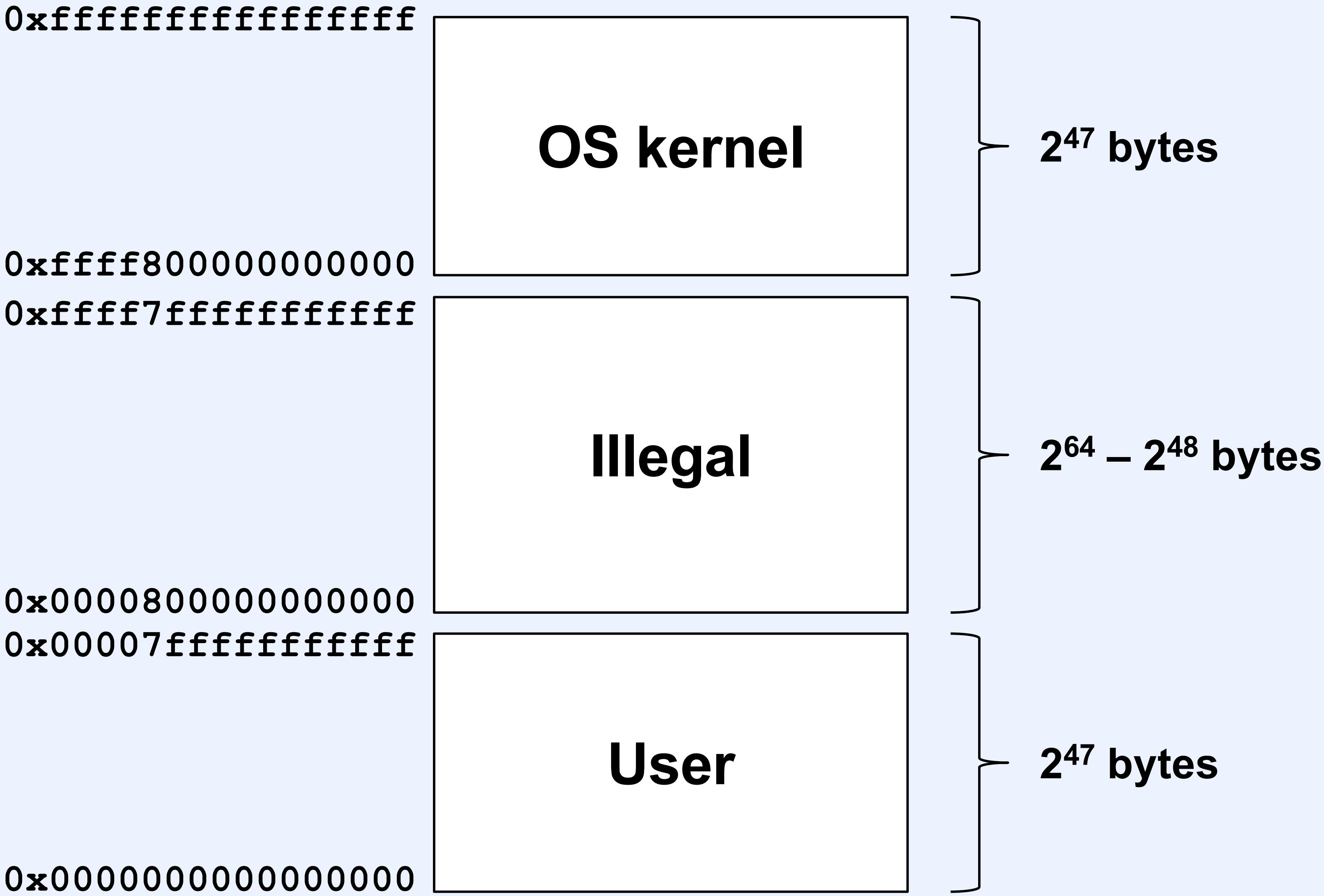
x86-64 Virtual Address
Format 3



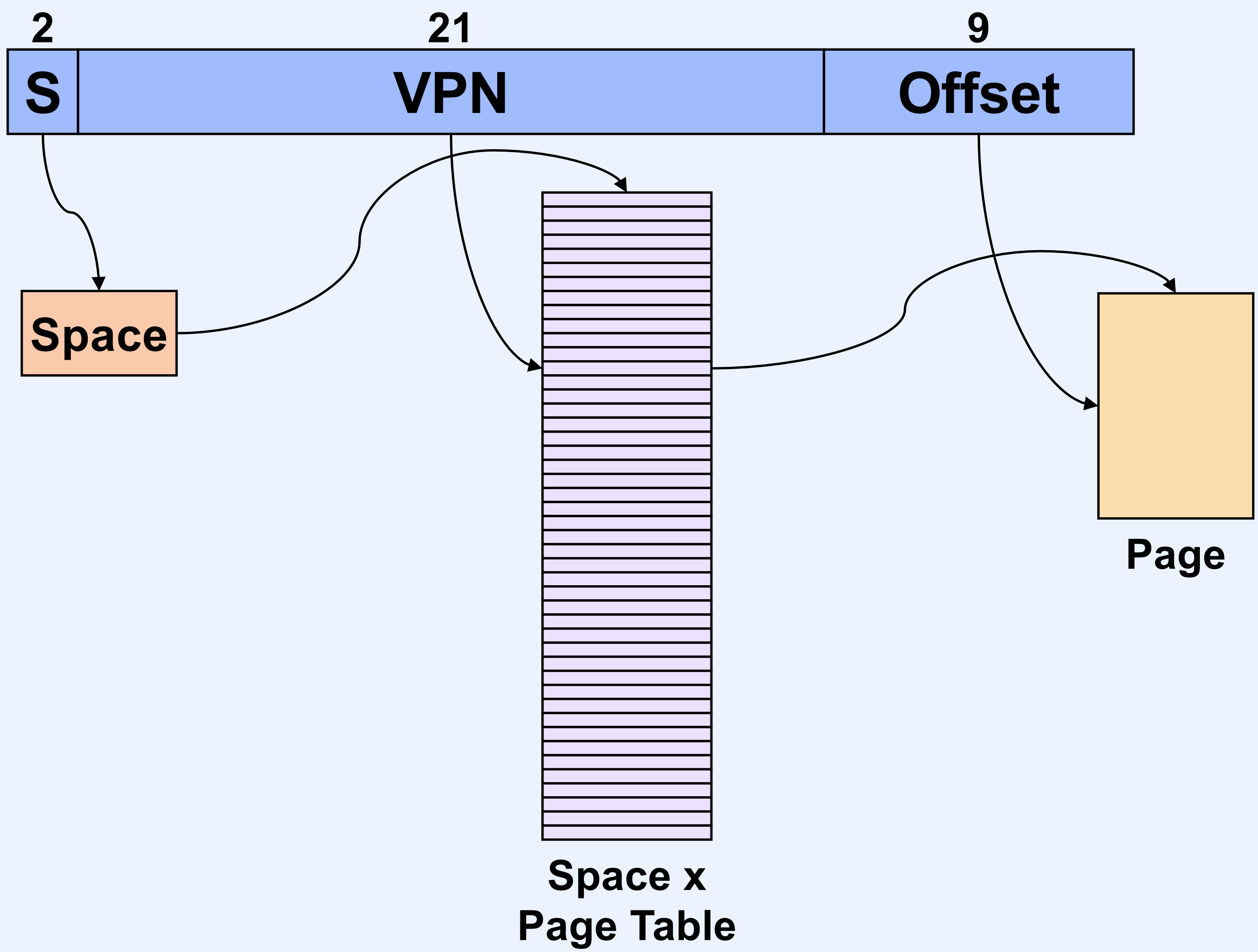
Why Multiple Page Sizes?

- 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

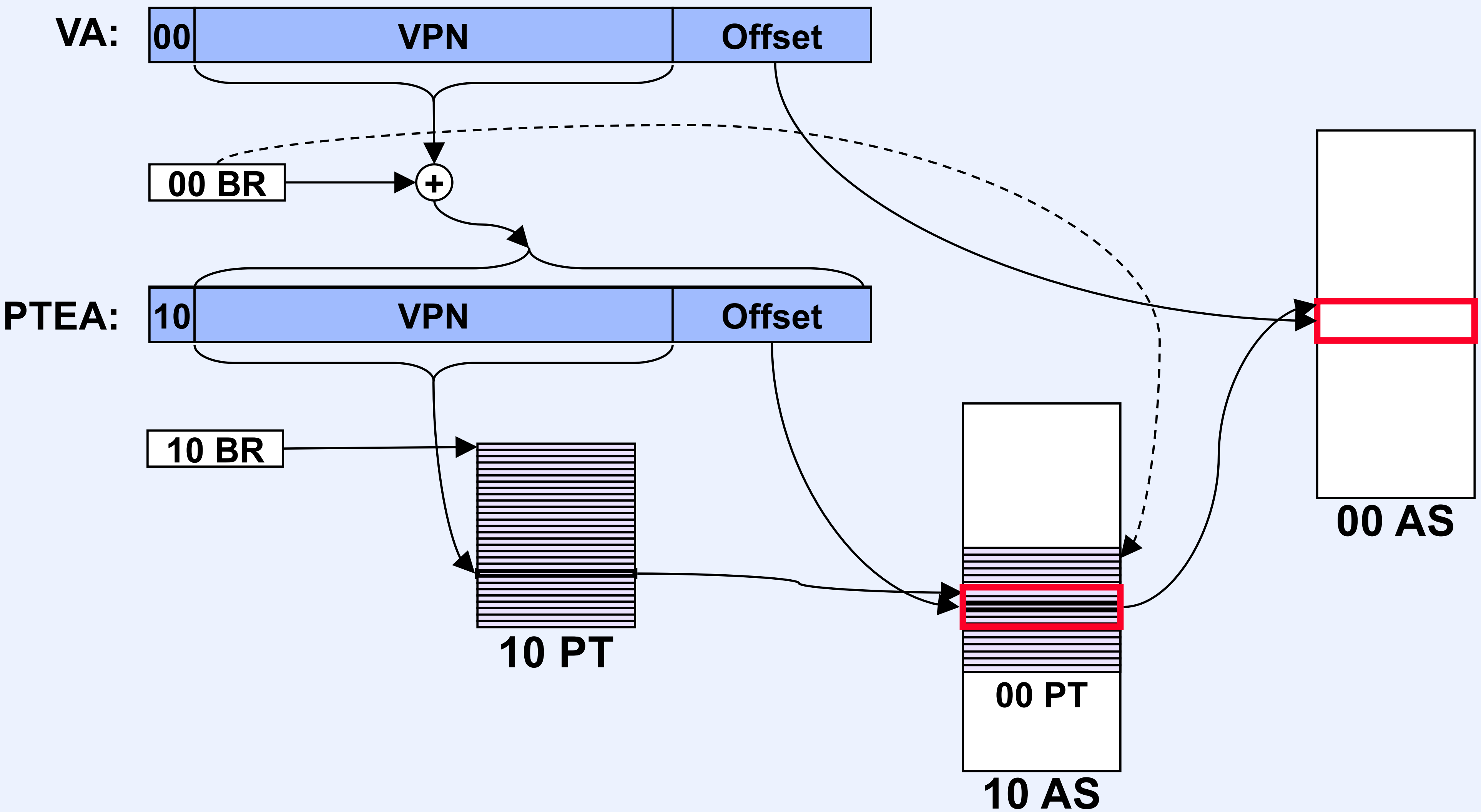
Address Space



Linear Page Table



VAX Linear Page Translation



$

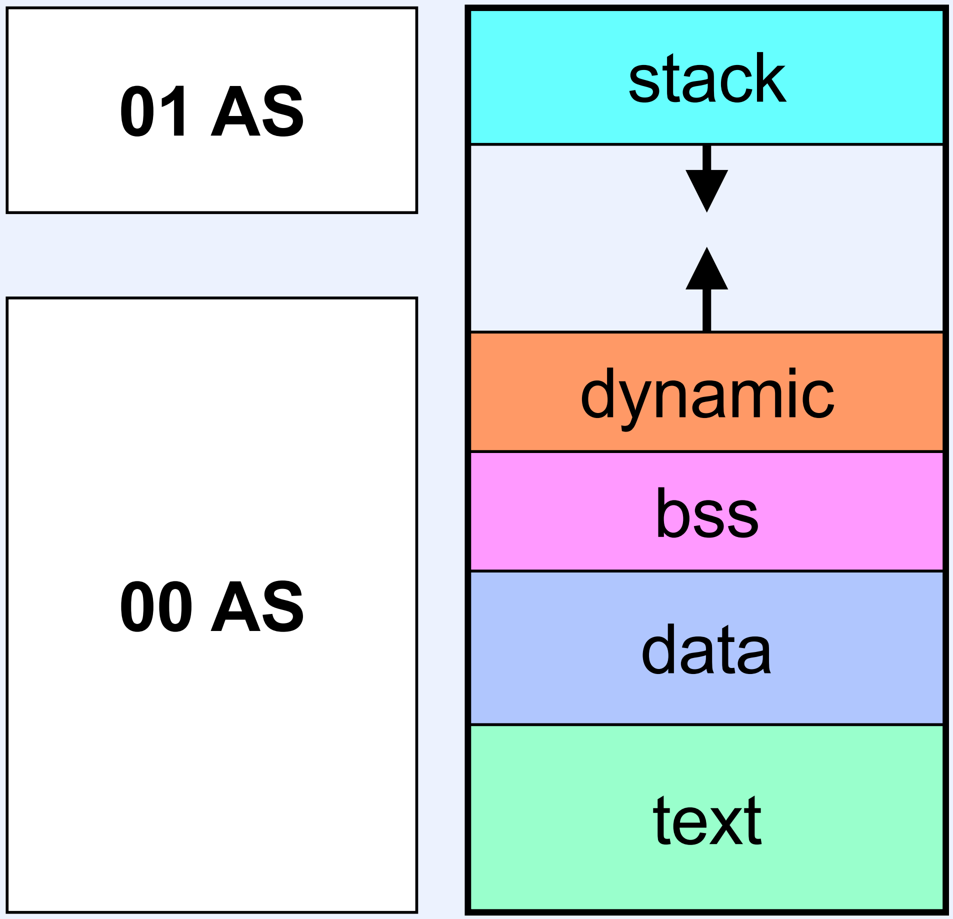
- VAX architecture introduced in 1978
  - memory cost $40,000/MB
    - 3.8¢/byte

      (.475¢/bit)

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

Traditional Unix with Linear PTs



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

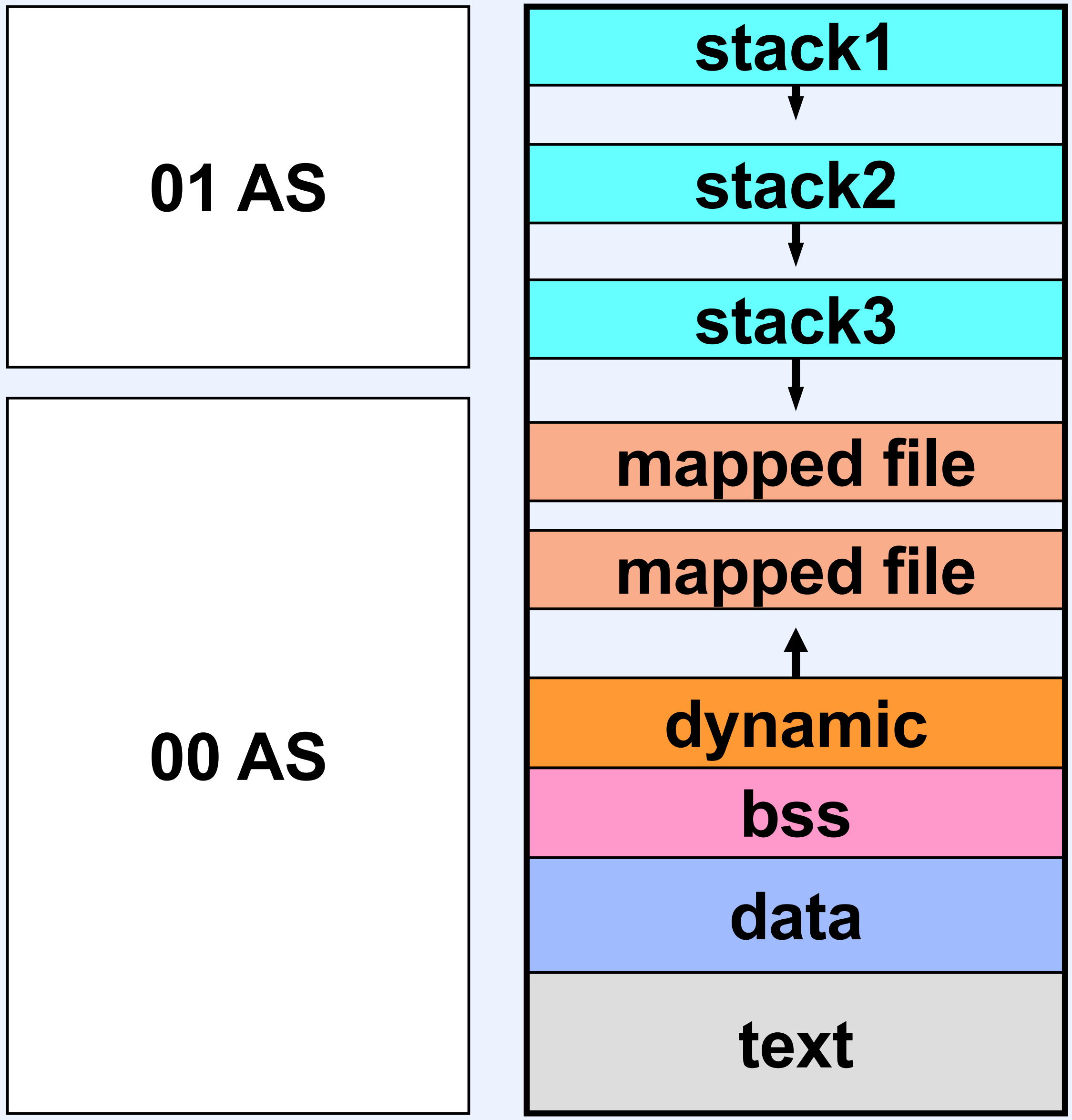
c) 16

d) 32

$

- **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

Modern Unix

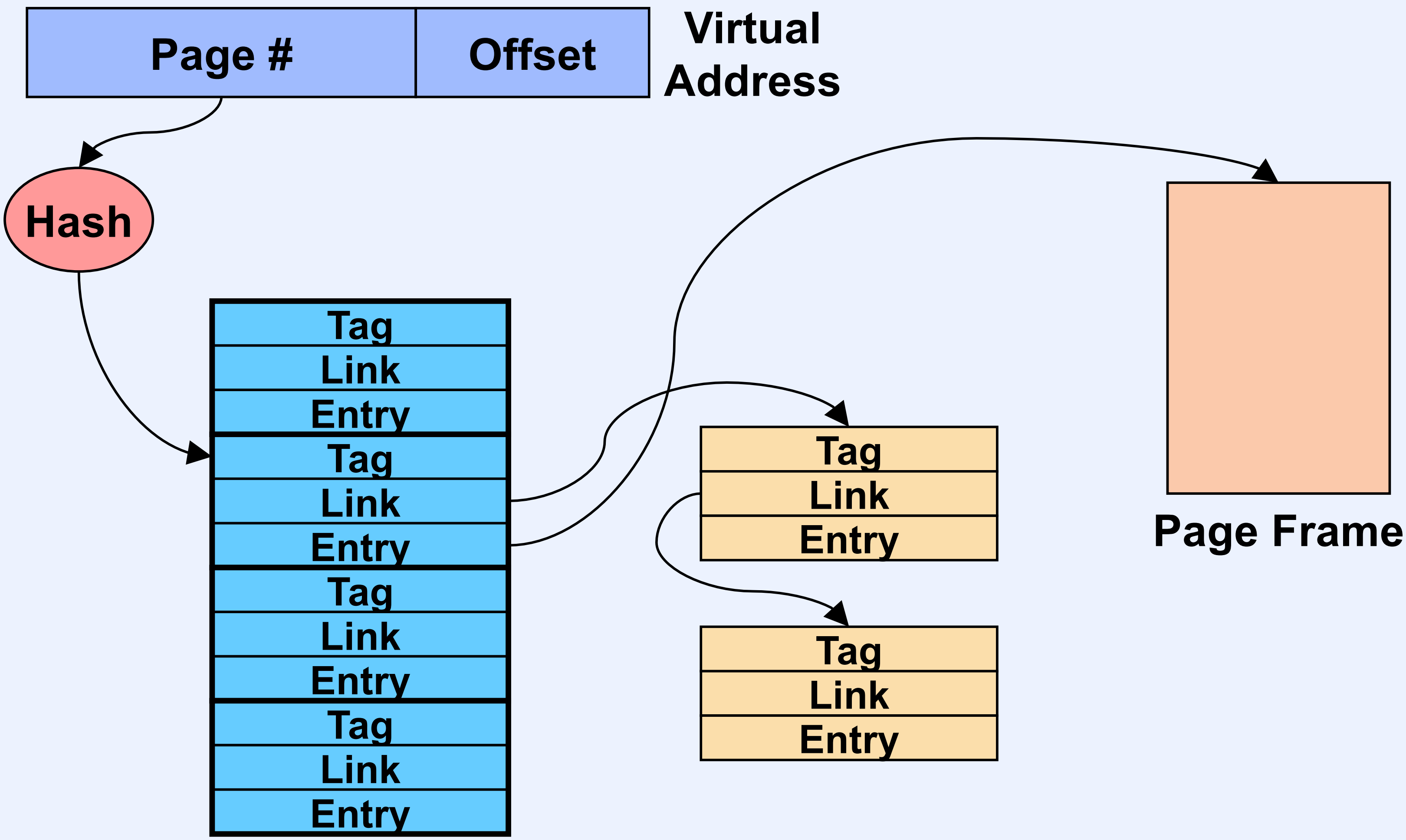

01 AS
00 AS
stack1
stack2
stack3
mapped file
mapped file
dynamic
bss
data
text

$

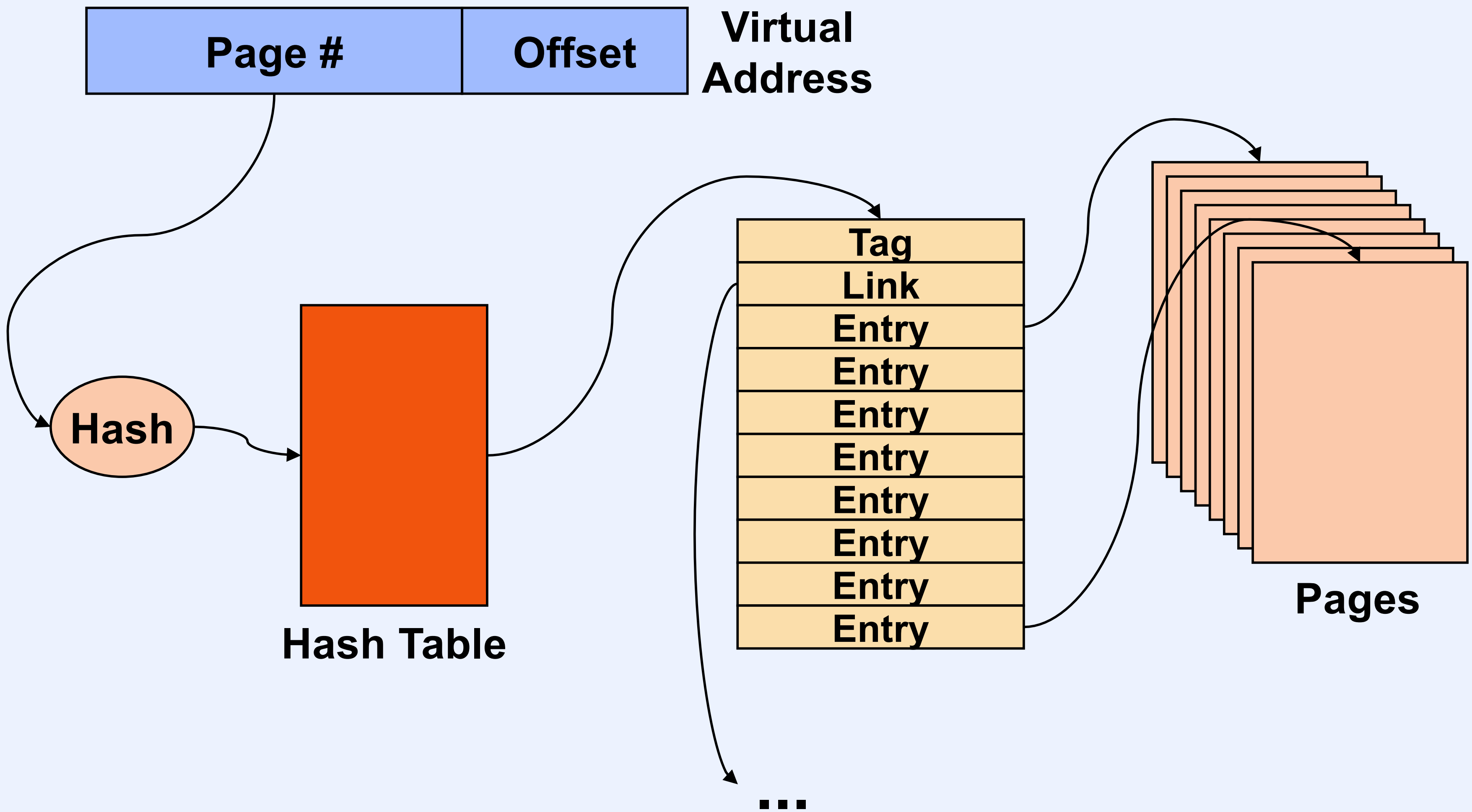
- **Requires sufficient 10 page-table entries to map almost all of 00 and 01 space**
  - **2^{14} 10 page-table entries for each space**
    - **requiring 64KB each, 128KB total**
    - **$5000 in 1978**

    - **<1¢/process today**
      - **who cares?**
      - **increase address space from 2^{32} to 2^{64}**
        - **4,294,967,296-fold increase**
          - **significant …**

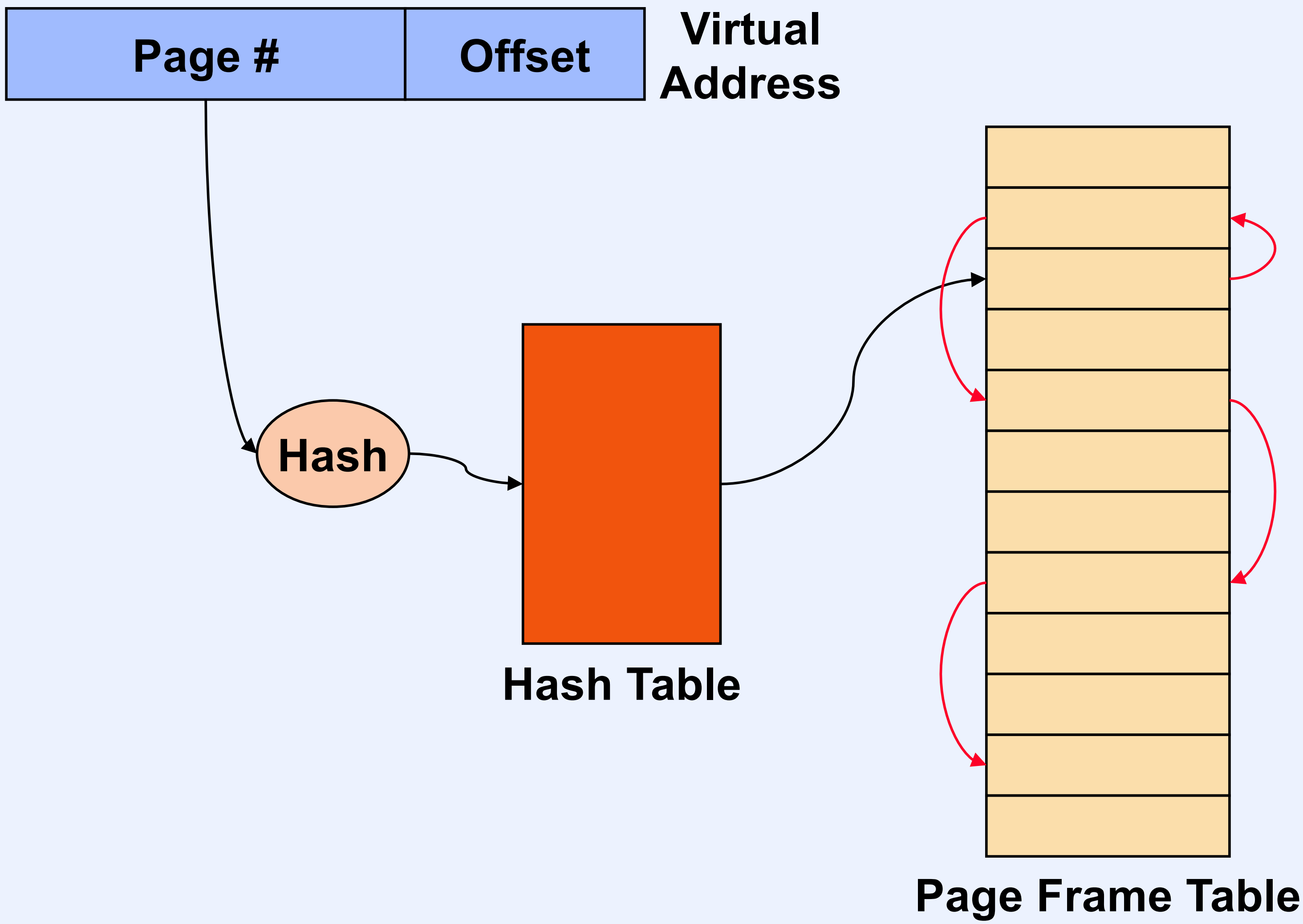
Hashed Page Tables



Clustered Page Tables



Inverted Page Tables



Quiz 4

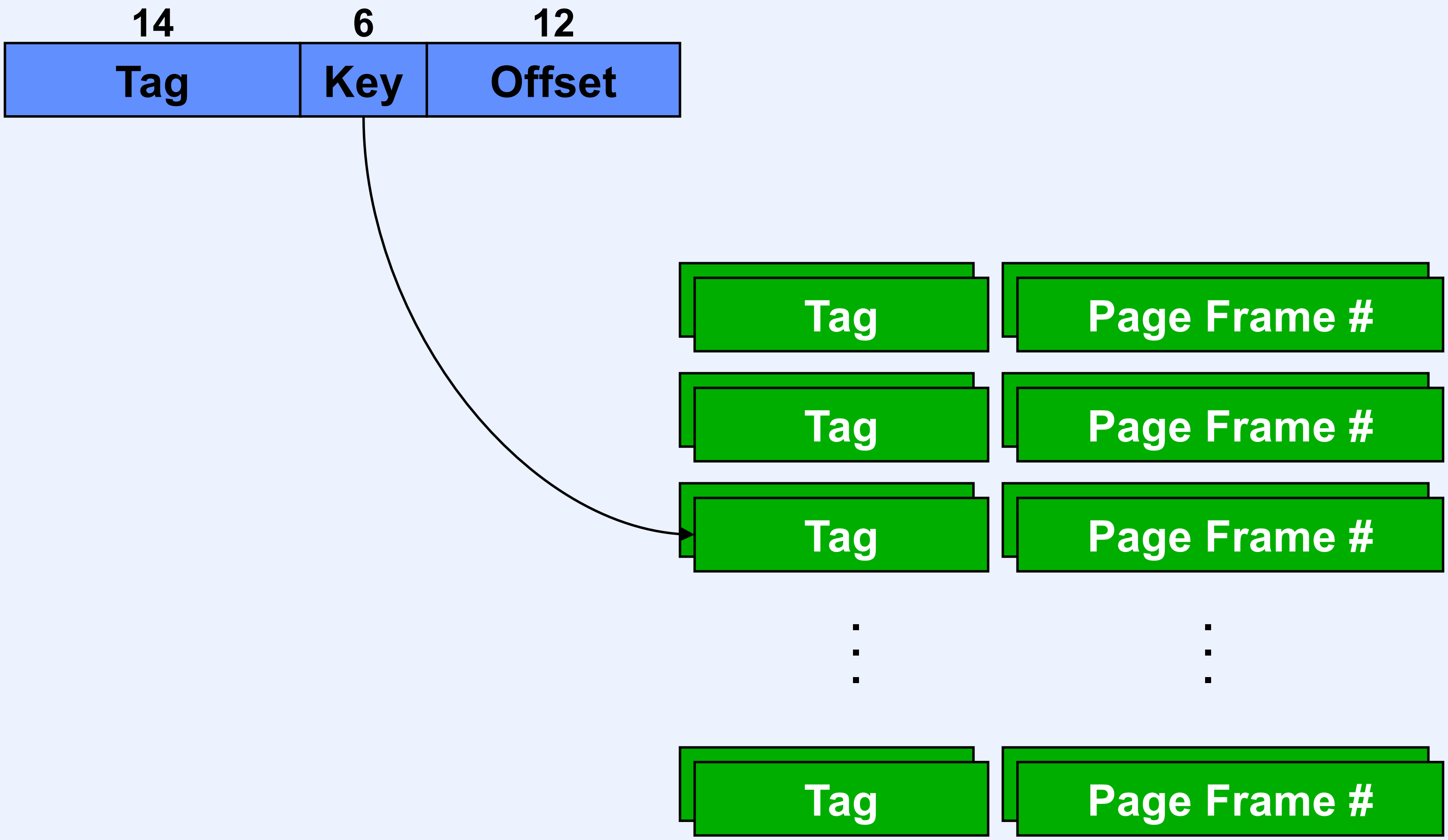
Normal page tables map virtual memory to real memory. More precisely, they map an address space and a location within that address space to real memory. Inverted page tables do the inverse mapping: given an address space ID and a location in real memory, they produce the corresponding virtual location.

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

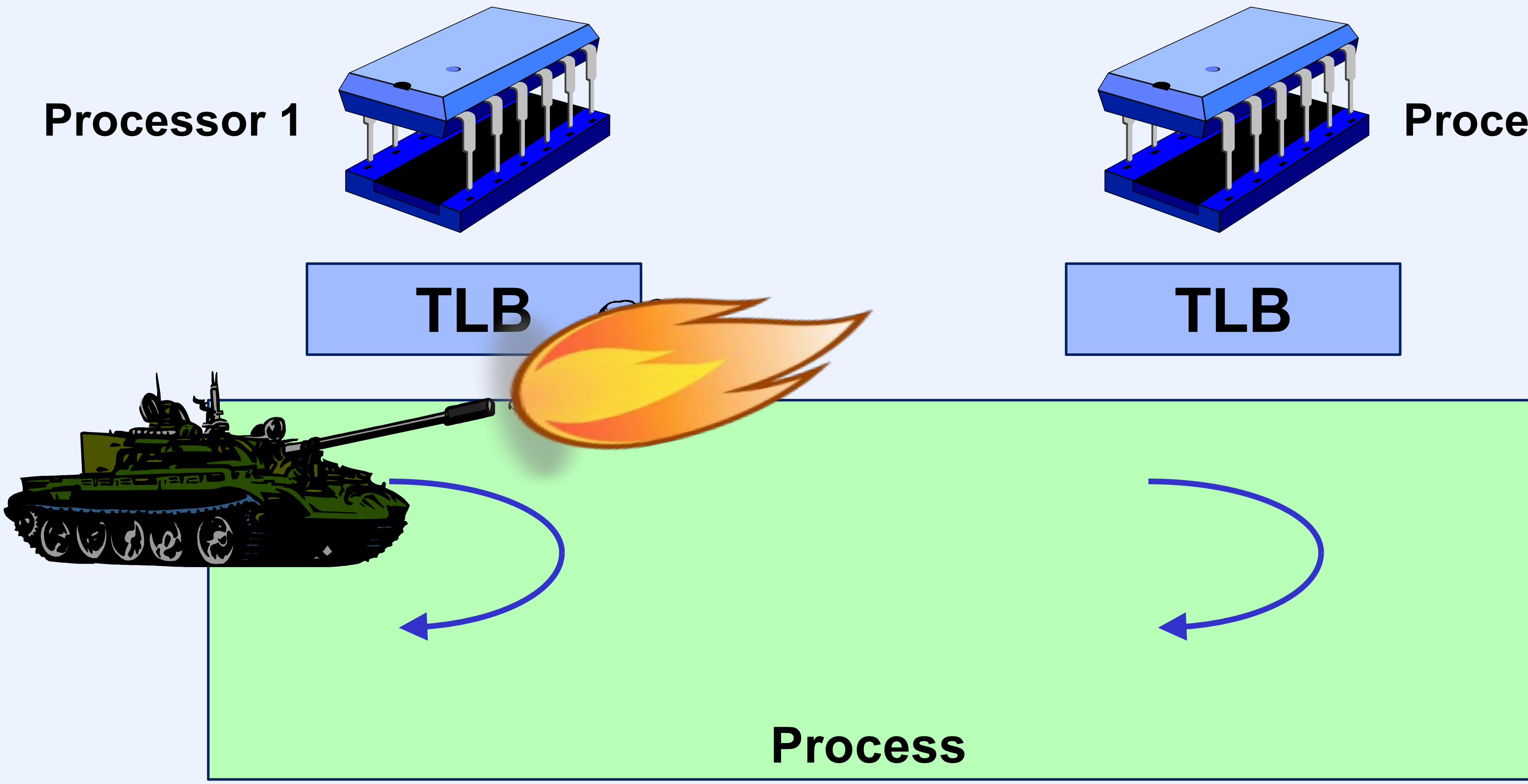
Translation Lookaside Buffers



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

TLBs and Multiprocessors



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()
```