Address Translation

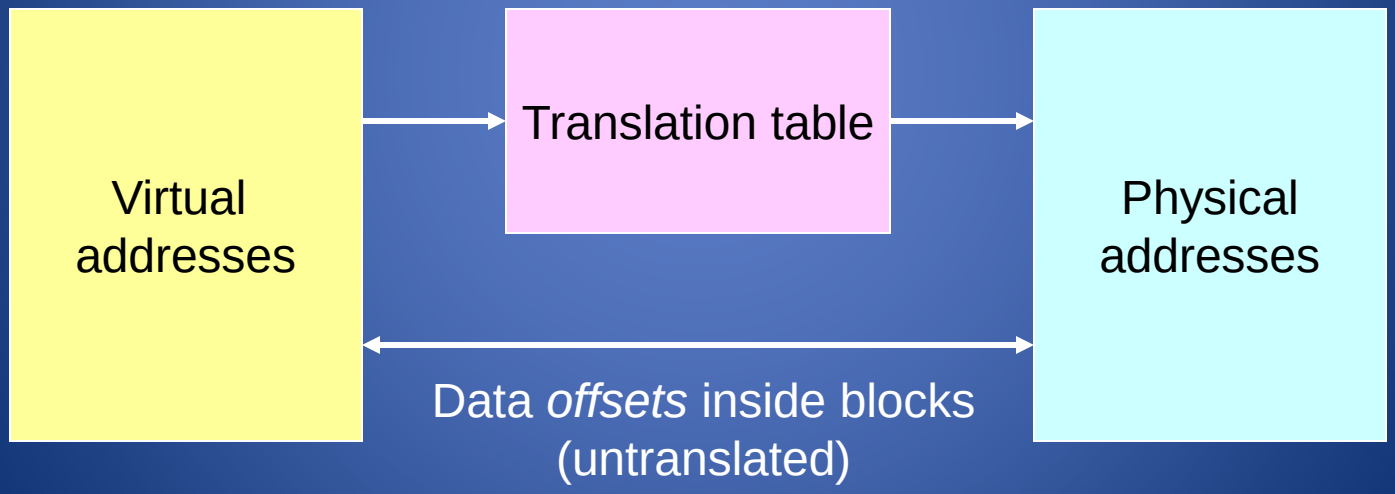
Sarah Diesburg

Operating Systems

CS 3430

Recall from Last Time…

- Translation tables are implemented in HW, controlled by SW



This Lecture…

- Different translation schemes
  - Base-and-bound translation
  - Segmentation
  - Paging
  - Multi-level translation
  - Paged page tables
  - Hashed page tables
  - Inverted page tables

Assumptions

- 32-bit machines
- 1-GB RAM max

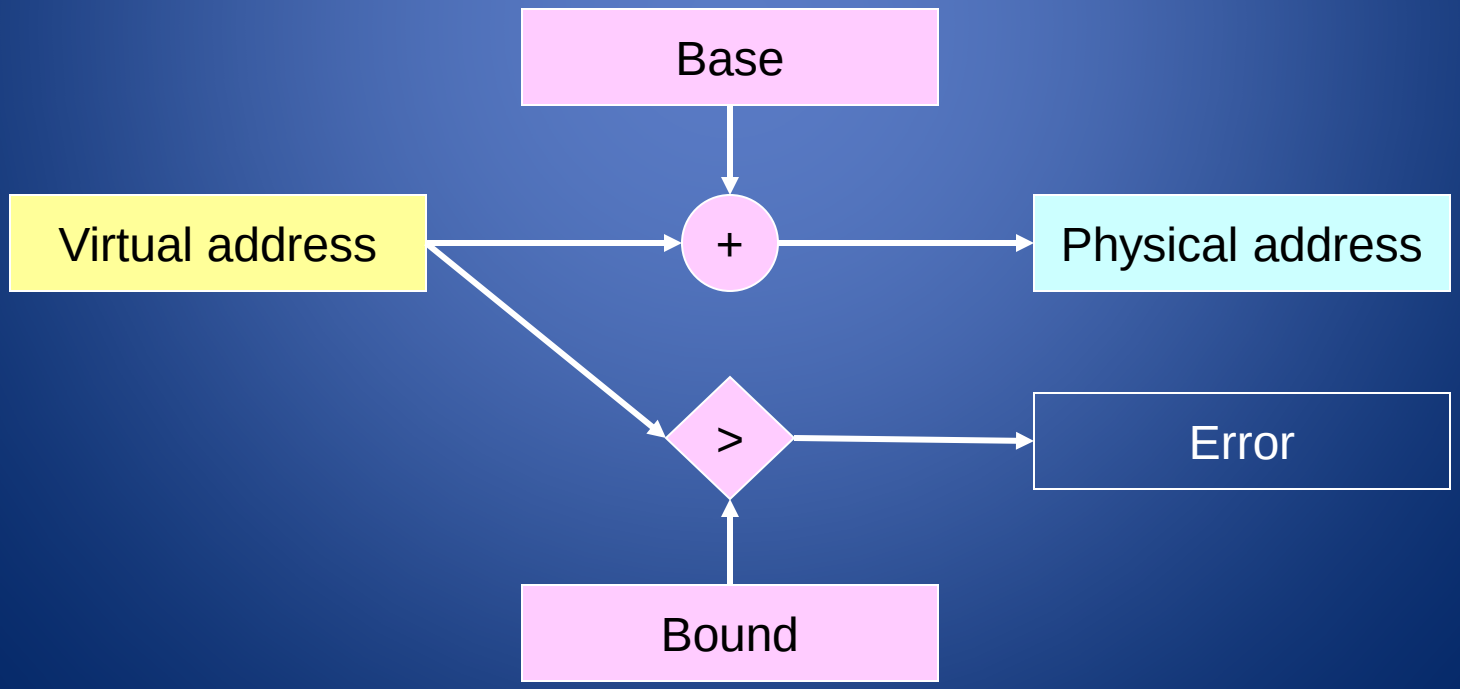
Physical address

$\lceil \log_2(1GB) \rceil$ = 30 bits for 1GB of RAM

1GB RAM = 2^{30}

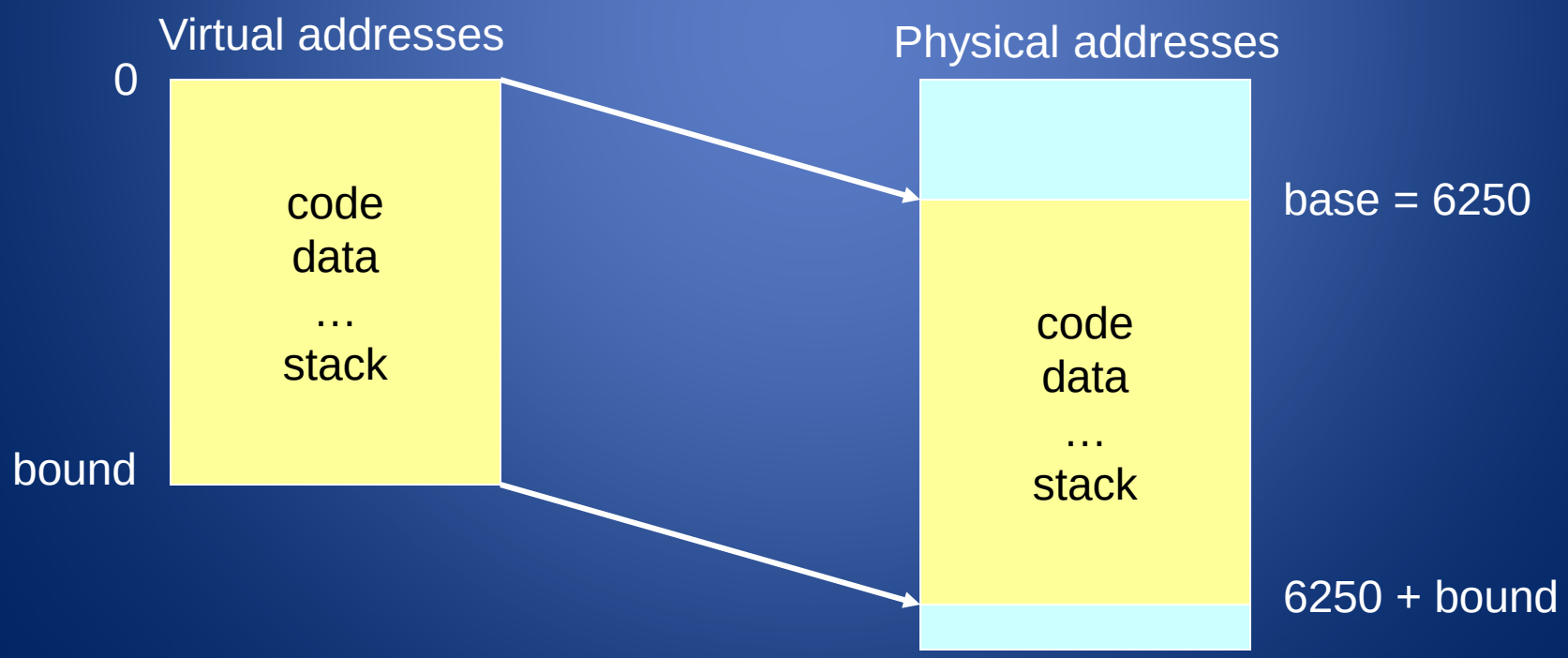
Base-and-Bound Translation

- Each process is loaded into a contiguous region of physical memory
- Processes are protected from one another



Base-and-Bound Translation

- Each process "thinks" that it is running on its own dedicated machine, with memory addresses from 0 to bound



Base-and-Bound Translation

- An OS can move a process around
  - By copying bits
  - Changing the base and bound registers

Pros and Cons of Base-and-Bound Translation

+ Simplicity

+ Speed

- ***External fragmentation***: memory is wasted because the available memory is not contiguous for allocation

- Difficult to share programs
  - Each instance of a program needs to have a copy of the code segment

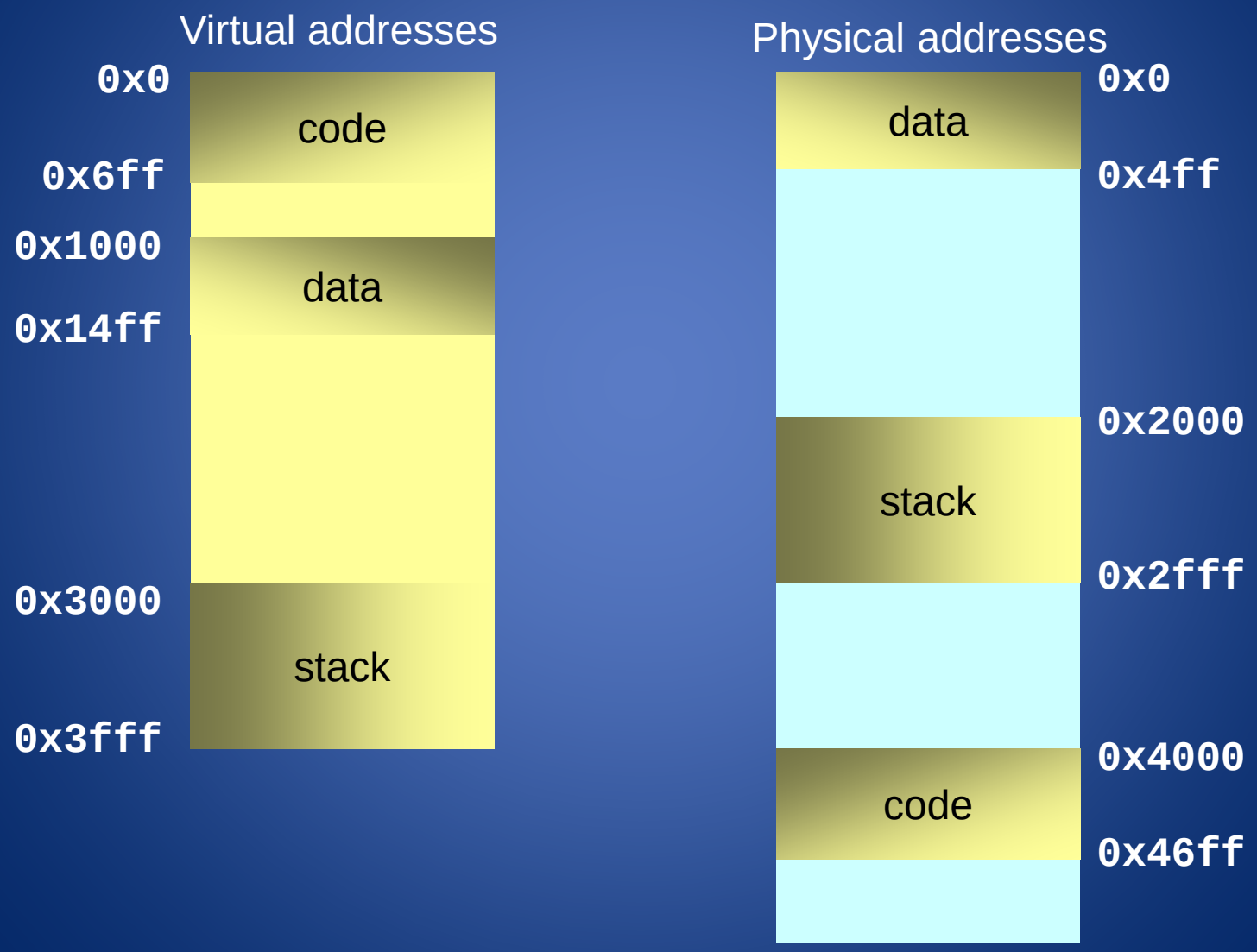
Pros and Cons of Base-and-Bound Translation

- Memory allocation is complex
  - Need to find contiguous chunks of free memory
  - Reorganization involves copying
- Does not work well when address spaces grow and shrink dynamically

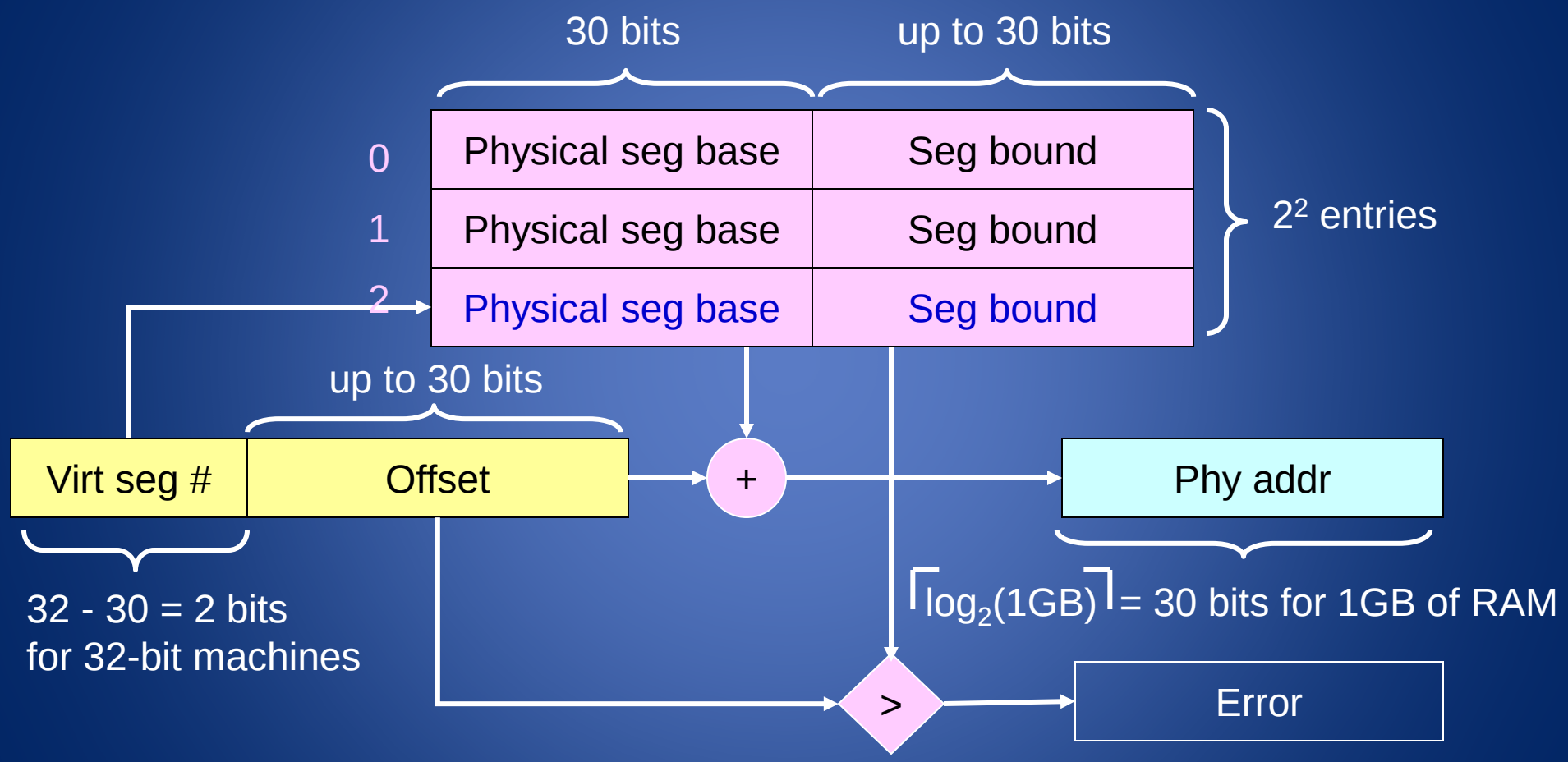
Segmentation

- ***Segment***: a region of logically contiguous memory
- ***Segmentation-based translation***: use a table of base-and-bound pairs

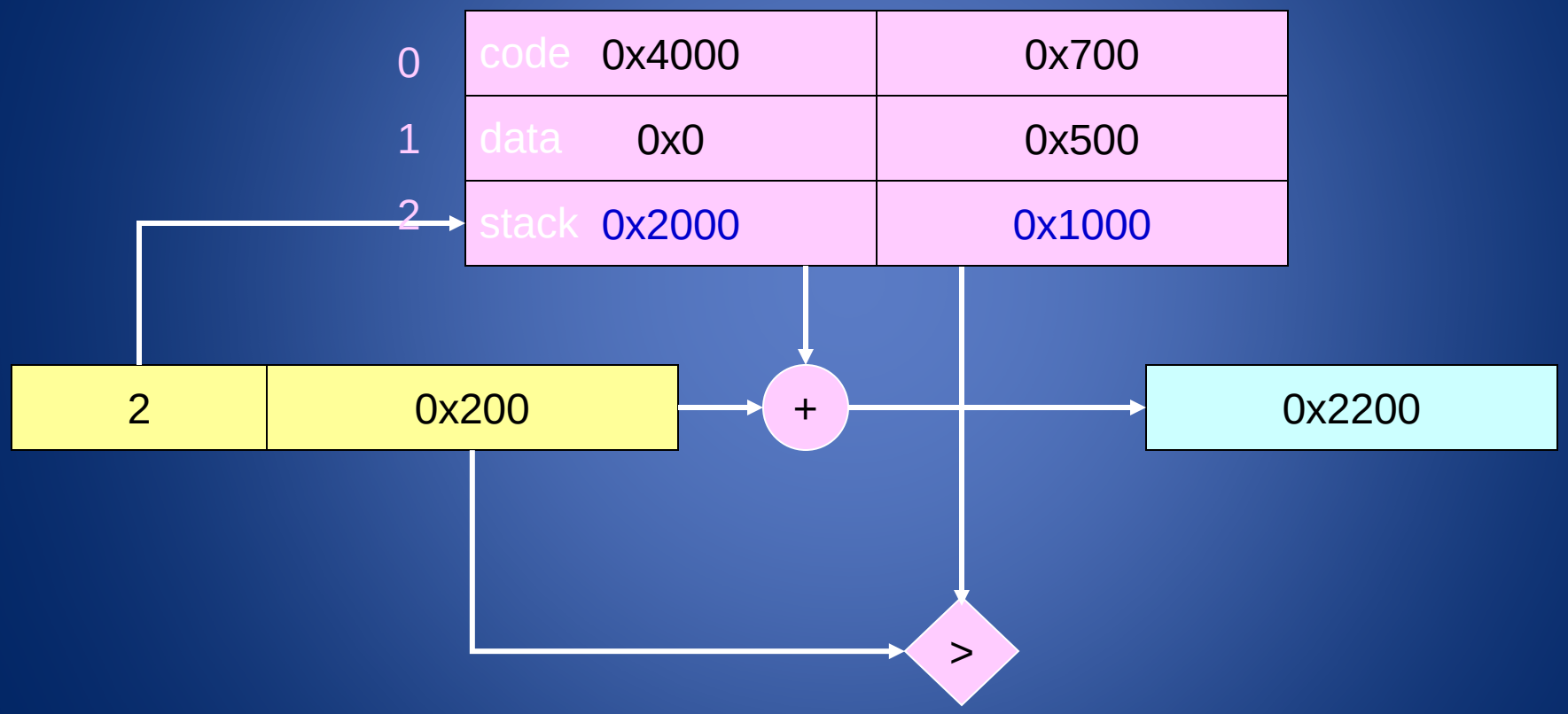
Segmentation Illustrated



Segmentation Diagram



Segmentation Diagram



Segmentation Translation

- virtual_address = virtual_segment_number:offset
- physical_base_address = segment_table[virtual_segment_number]
- physical_address = physical_base_address + offset

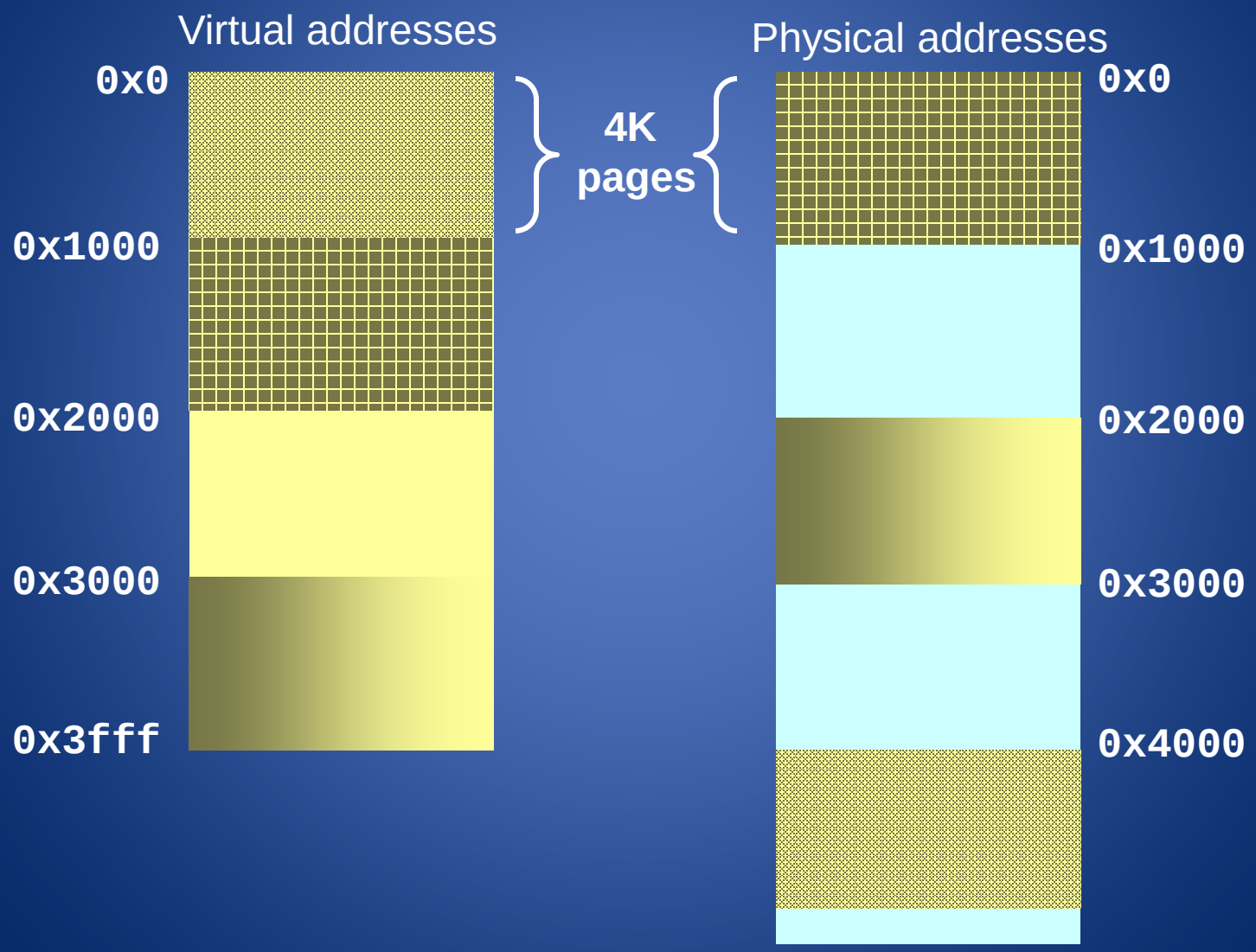
Pros and Cons of Segmentation

+ Easier to grow and shrink individual segments
+ Finer control of segment accesses
  - e.g., read-only for shared code segment
+ More efficient use of physical space
+ Multiple processes can share the same code segment
- Memory allocation is still complex
  - Requires contiguous allocation

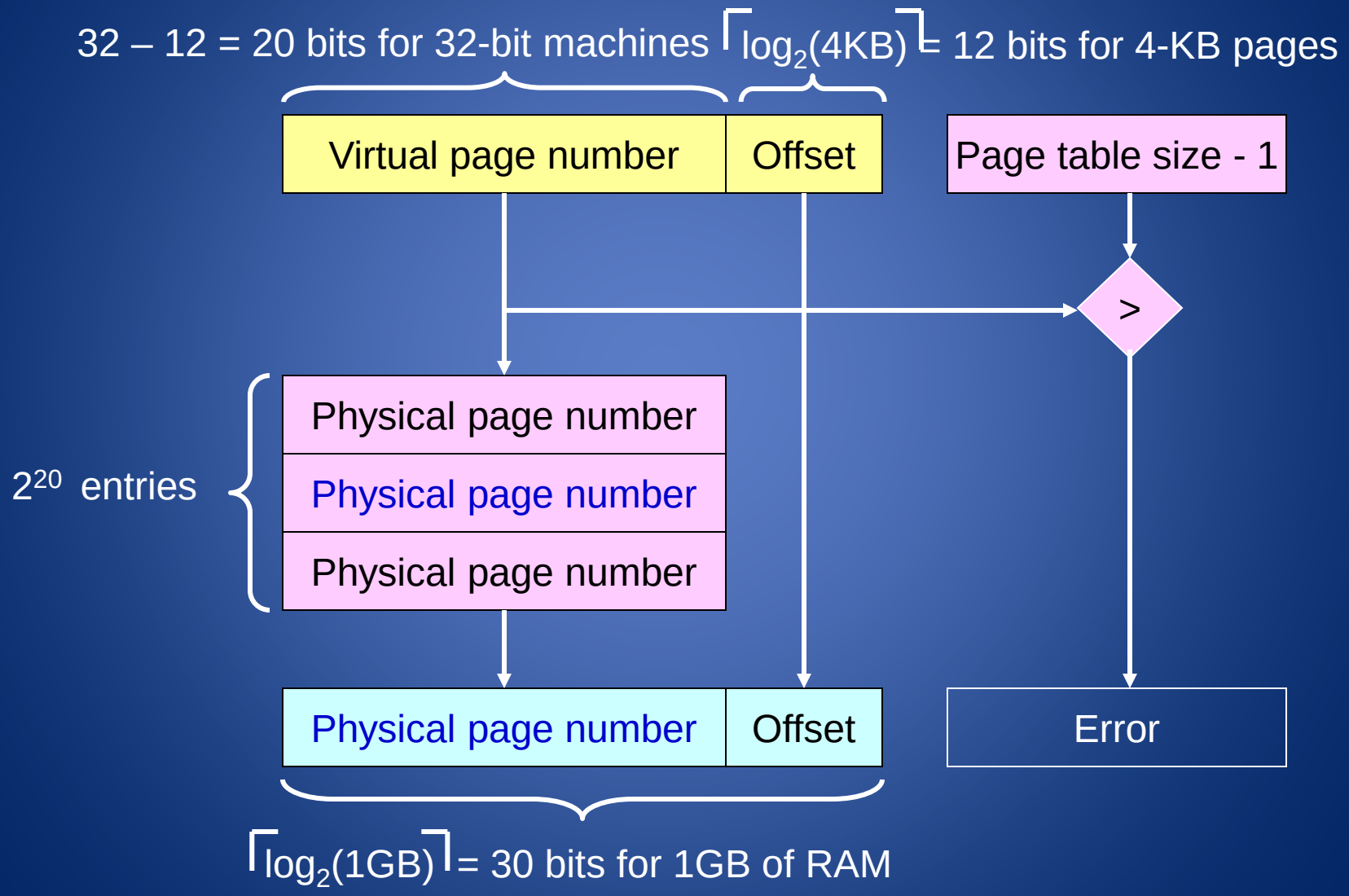
Paging

- ***Paging-based translation***: memory allocation via fixed-size chunks of memory, or ***pages***
  - Translation granularity is a page
- The memory manager uses a ***bitmap*** to track the allocation status of memory pages
  - Array of bits (0 or 1) to signify free or used pages

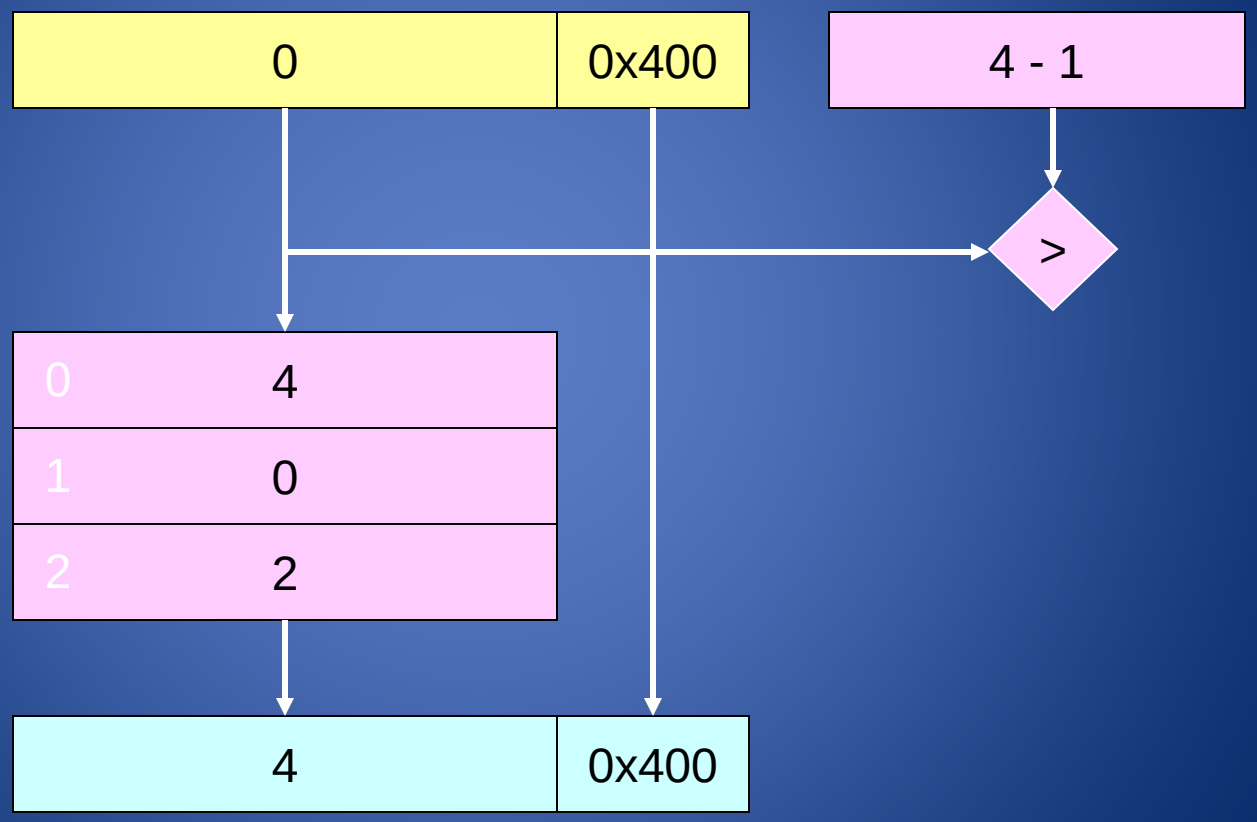
Paging Illustrated



Paging Diagram



Paging Example



Paging Translation

- virtual_address = virtual_page_number:offset
- physical_page_number = page_table[virtual_page_number]
- physical_address = physical_page_number:offset

Pros and Cons of Paging

+ Easier memory allocation

+ Allows code sharing

- ***Internal fragmentation***: allocated pages are not fully used

- The page table size can potentially be very large
  - 32-bit architecture with 1-KB pages can require 4 million table entries

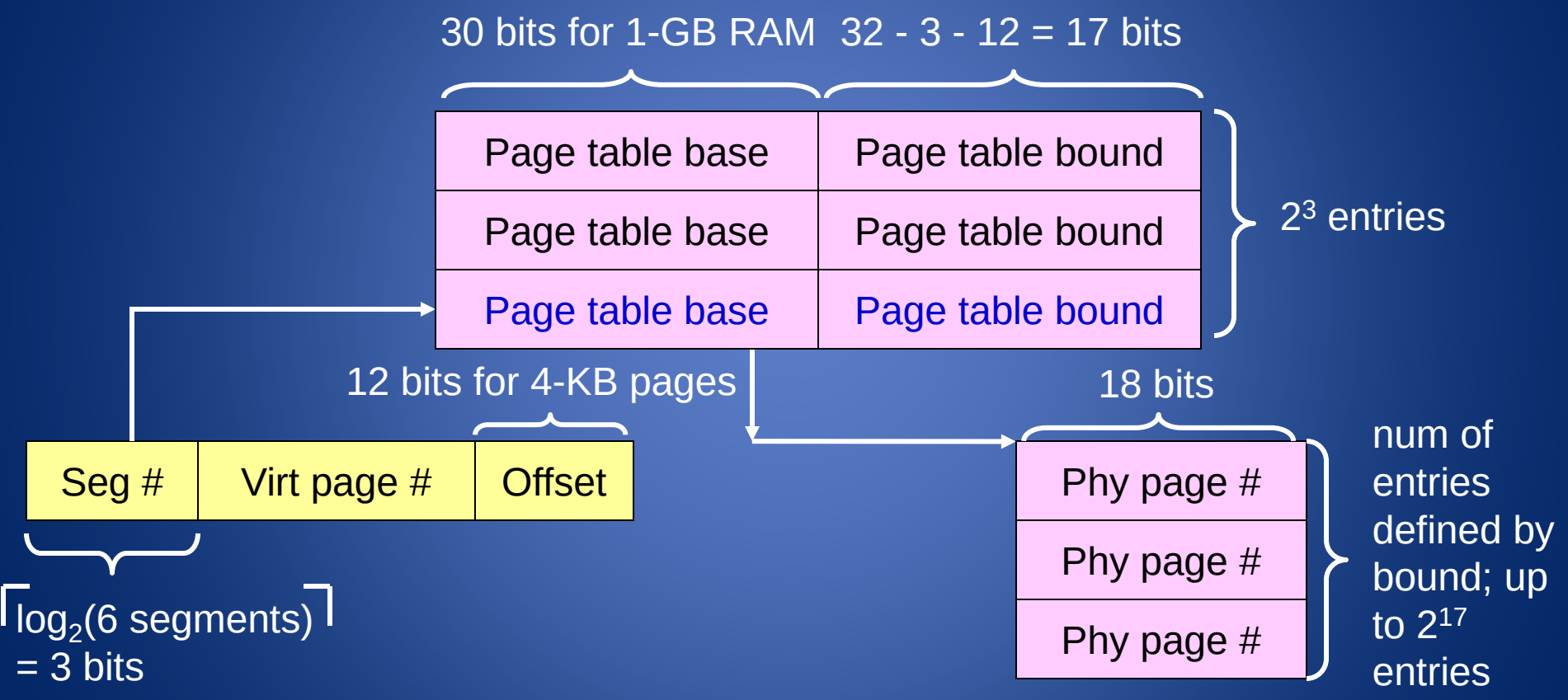
Multi-Level Translation

- ***Segmented-paging translation***: breaks the page table into segments
- ***Paged page tables***: Two-level tree of page tables

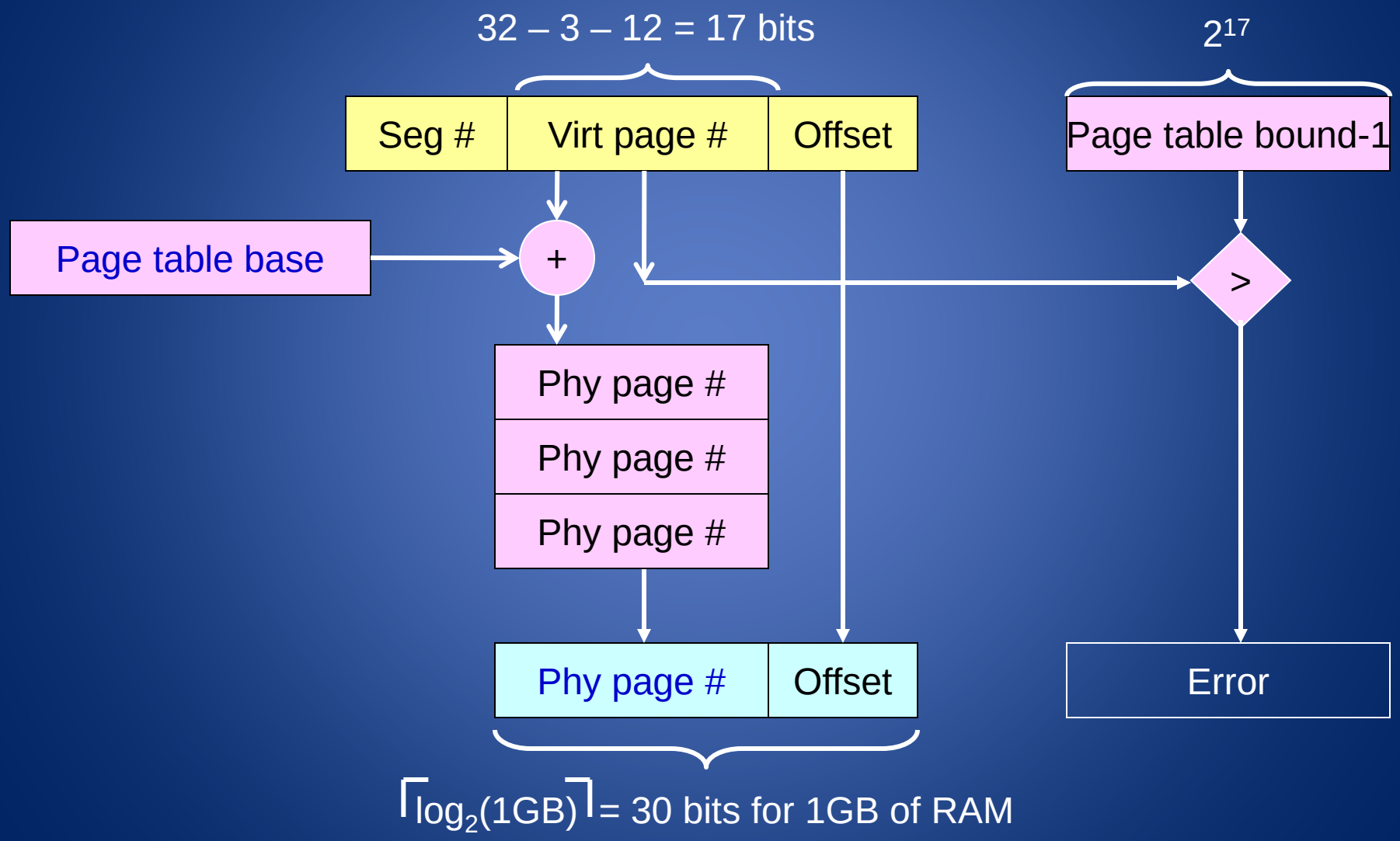
Segmented Paging

1. Start with a virtual memory address
2. Look up the page table address in the segment table
3. Index into the page table to get the physical page number
4. Concatenate the physical page number to the offset

Segmented Paging



Segmented Paging (Cont.)



Segmented Paging Translation

- virtual_address = segment_number:page_number:offset
- page_table (base address)= segment_table[segment_number]
- physical_page_number = page_table[virtual_page_number]
- physical_address = physical_page_number:offset

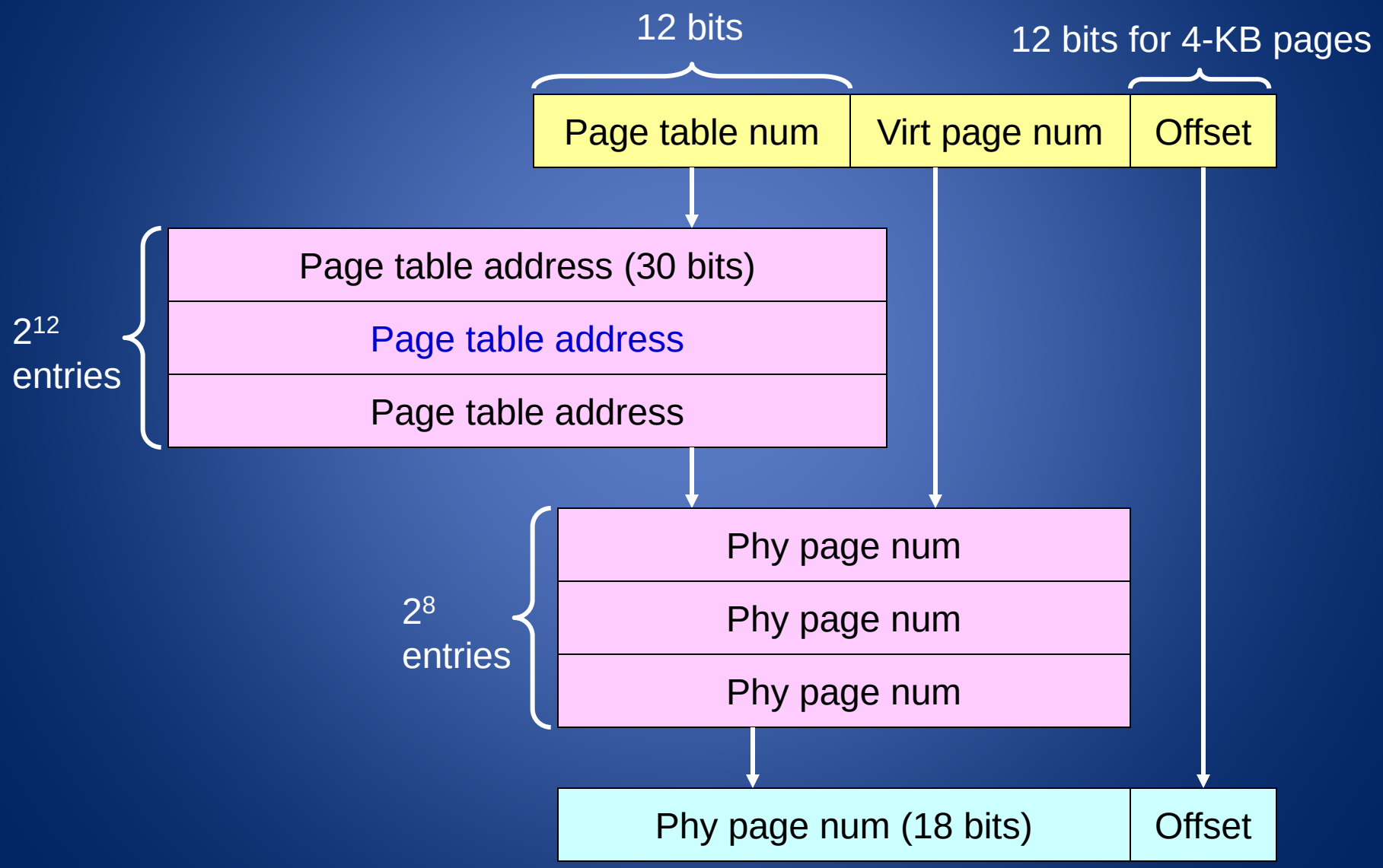
Pros and Cons of Segmented Paging

+ Code sharing
+ Reduced memory requirements for page tables
- Higher overhead and complexity
- Page tables still need to be contiguous
- Each memory reference now takes two lookups

Paged Page Tables

1. Start with a virtual memory address
2. Index into the first page table to find the address to the second page table
3. Index into the second page table to find the physical page number
4. Concatenate the physical page number to the offset

Paged Page Tables



Paged Page Table Translation

- virtual_address = outer_page_num:inner_page_num:offset
- page_table = outer_page_table[outer_page_num]
- physical_page_num = inner_page_table[inner_page_num]
- physical_address = physical_page_num:offset

Pros and Cons of Paged Page Tables

+ Can be generalized into multi-level paging

- Multiple memory lookups are required to translate a virtual address
  - Can be accelerated with ***translation lookaside buffers*** (TLBs)
    - Stores recently translated memory addresses for short-term reuses

Hashed Page Tables

- Physical_address
  = hash(virtual_page_num):offset

+ Conceptually simple

- Need to handle collisions

- Need one hash table per address space

Inverted Page Table

- One hash entry per physical page
- physical_address
  - = hash(pid, virtual_page_num):offset

+ The number of page table entries is proportional to the size of physical RAM

- Collision handling