Operating Systems 2017/2018 — Part VIII – Memory Management

Ricardo Rocha

Department of Computer Science

Faculty of Sciences

University of Porto

Slides based on the book

'Operating System Concepts, 9th Edition,

Abraham Silberschatz, Peter B. Galvin and Greg Gagne, Wiley'

Chapter 8

Background

- **Memory (or main memory)** consists of a **large array of bytes** each with its **own address**
  - The memory unit sees only a stream of memory addresses and it does not know how they are generated or what they are for
- Memory and registers are the only storage CPU can access directly
  - Programs must be brought from disk into memory to be run
  - CPU then fetches instructions from memory according to the program counter
  - These instructions may cause additional loading/storing from/to specific memory addresses

Basic Hardware

- Most CPUs can decode instructions and perform simple operations on **registers** at the rate of **one or more operations per CPU clock tick**
- **Memory** is accessed via a **memory bus** which may take **several cycles of the CPU clock tick**
  - Causes the CPU to stall when it does not have the data required to complete the instruction that it is executing
  - This situation is intolerable because of the frequency of memory accesses
- **Cache** sits between memory and CPU registers
  - Typically built into the CPU chip and automatically managed by the hardware in order to speed up memory access without any operating system control

Memory Space

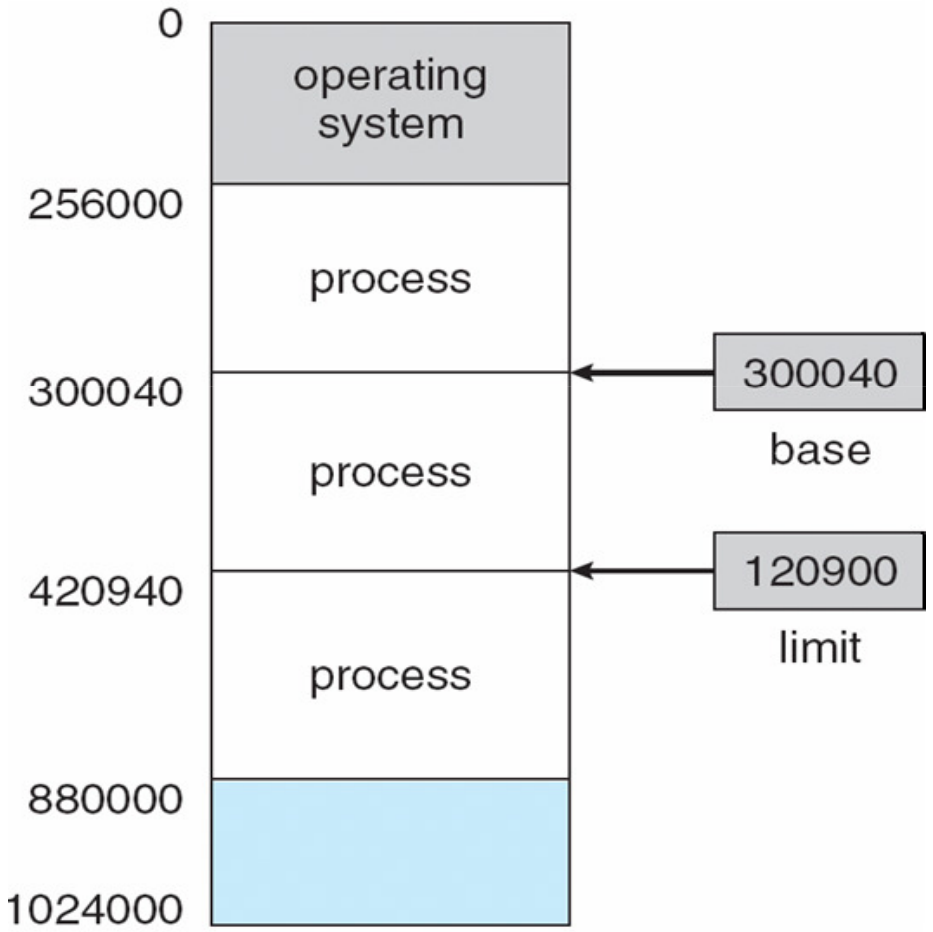
- **Uniprogramming**
  - Applications always runs at same place in physical memory
  - Applications can access any physical address
  - No memory protection required since only one application at a time
- **Multiprogramming**
  - Applications can run in different physical memory locations (use linker/loader to adjust addresses while program loaded into memory)
  - Applications may not access all physical addresses (memory usually divided into two partitions: (i) operating system, usually in low memory together with the interrupt vector; (ii) user processes, usually in high memory)
  - Memory protection required to prevent address overlap between processes

Memory Protection

- Memory protection is required to **ensure correct operation**
  - Protect operating system processes from access by user processes
  - Protect user processes from one another
- Memory protection must be **provided by the hardware**
  - Usually, the operating system does not intervene between the CPU and its memory accesses because of the resulting performance penalty
- We need to ensure that each process has a **separate memory space**
  - Separate per-process memory space protects the processes from each other and is fundamental to having multiple processes loaded in memory for concurrent execution
  - To separate memory spaces, we need the ability to determine the range of legal addresses that the process may access

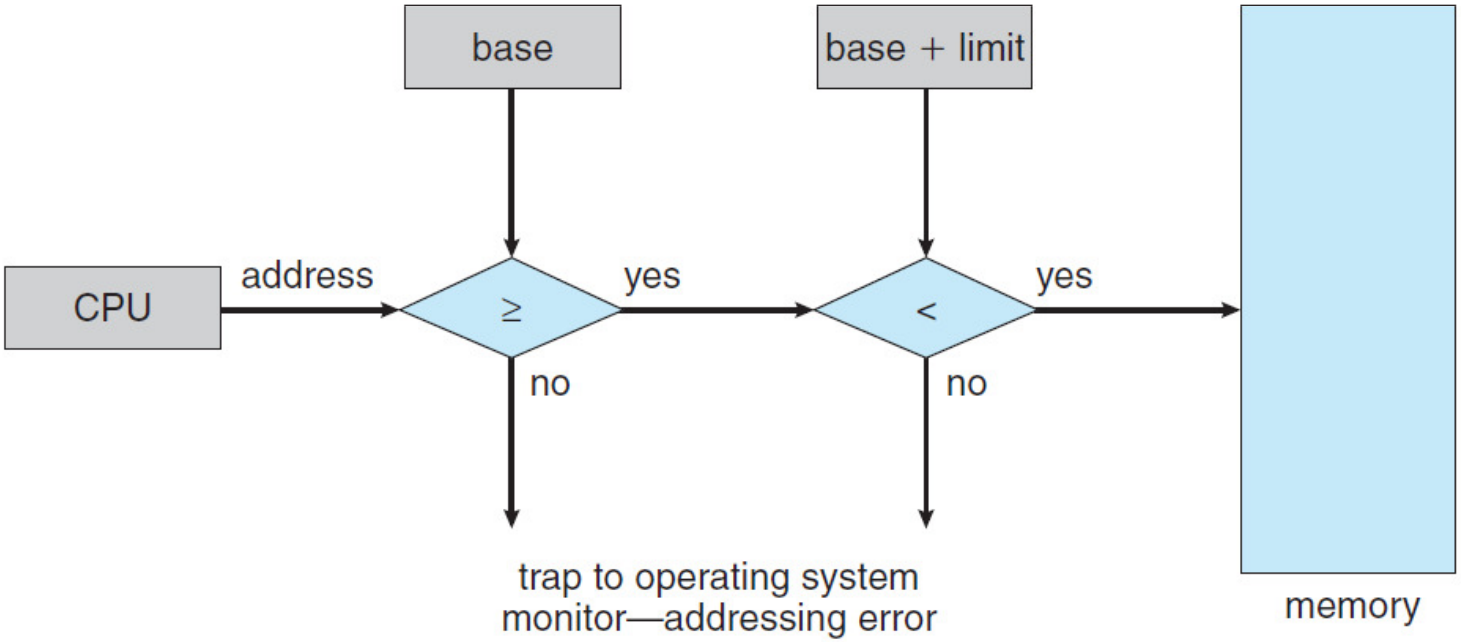
Base and Limit Registers

- A pair of registers define the **address space** of a process:
  - The **base register** holds the smallest legal physical memory address
  - The **limit register** specifies the size of the range
- During context switch, OS uploads base/limit registers from PCB
  - User not allowed to change base/limit registers
- Without this protection, bugs in any program can cause other programs or even the operating system to crash



Hardware Address Protection

- Memory protection is accomplished by the CPU hardware by comparing every address generated in user mode with the base/limit registers
  - Any attempt by a process executing in user mode to access operating system memory or other processes' memory results in a trap to the operating system
  - This prevents a user process from (accidentally or deliberately) modifying the code or data structures of either the operating system or other processes



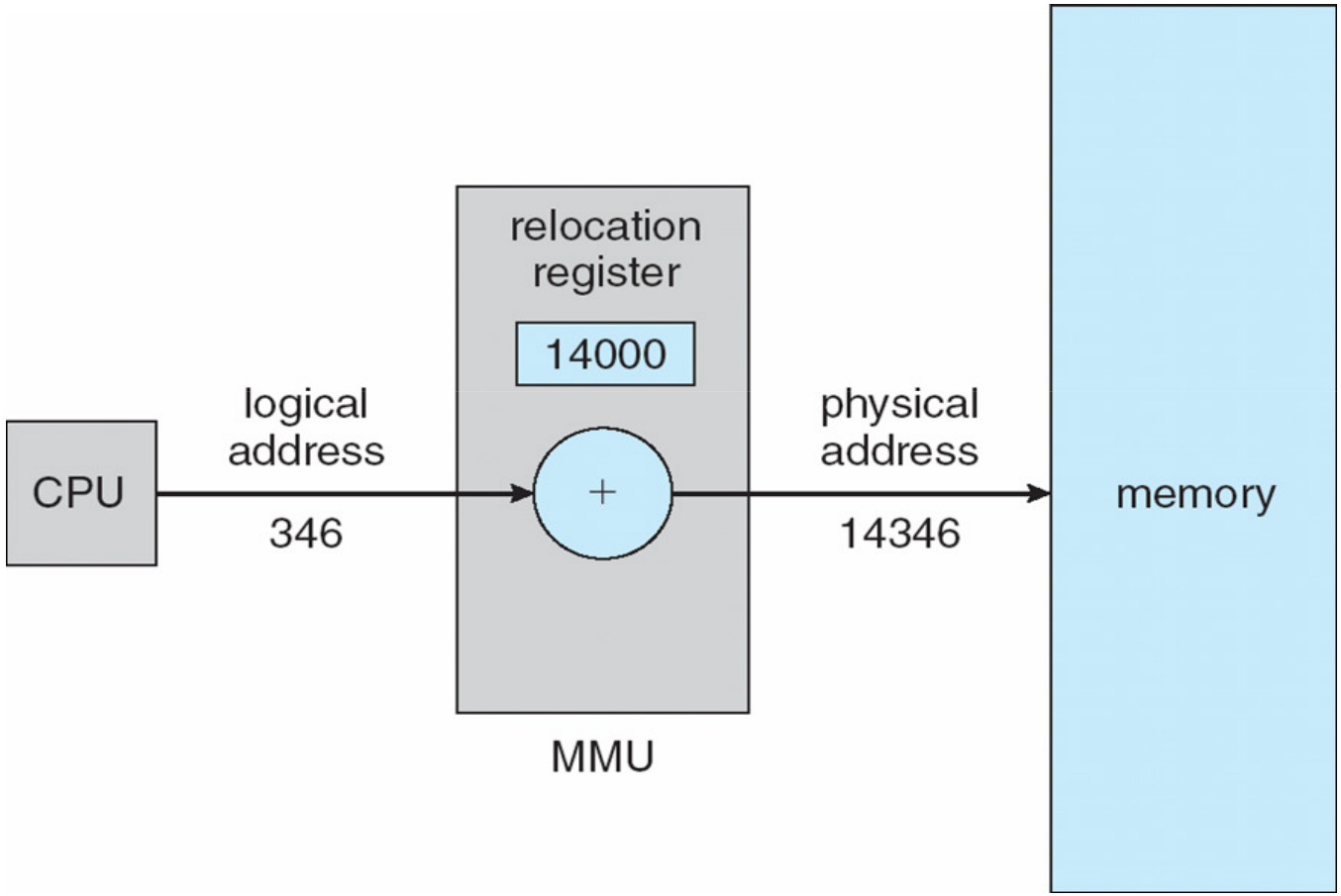
Logical x Physical Address Space

- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management
  - **Logical address** – address generated by the CPU
  - **Physical address** – address seen by the memory unit
- The set of all logical addresses generated by a program is the **logical address space** and the set of all physical addresses corresponding to these logical addresses is the **physical address space**
- At compile time and load time, the address-binding scheme generates the same logical and physical addresses
- At execution time, the address-binding scheme results in different logical and physical addresses

Memory Management Unit (MMU)

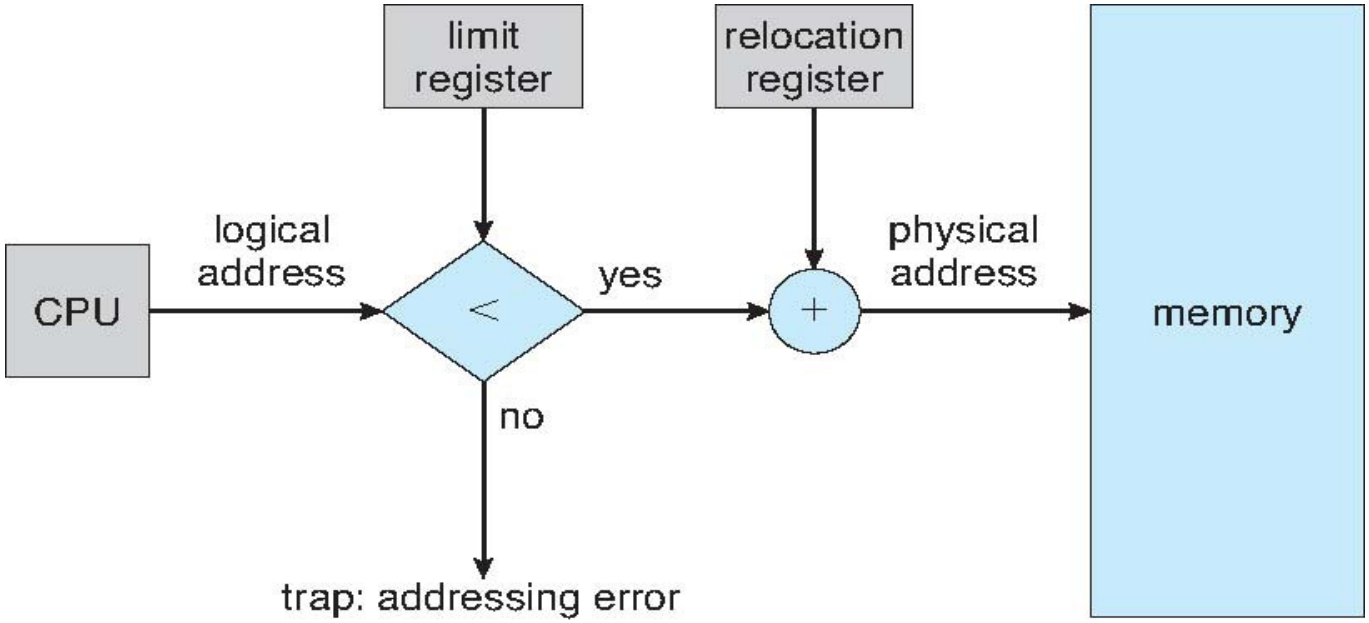
- The **hardware device** that at run time **maps logical to physical addresses** is called the **memory-management unit (or MMU)**
  - Many different methods to accomplish such mapping are possible
- To start, consider the following mapping scheme:
  - The base register is now called a **relocation register**
  - The value in the relocation register is **added** to every address generated by a user process at the time the address is sent to memory
- The user program generates only logical addresses (in the range 0 to Max) and never sees the real physical addresses (in the range B to B+Max for a base value B)
  - Execution time address-binding occurs when reference is made to location in memory

MMU Using a Relocation Register



Memory Protection Revisited

- Relocation and limit registers can be used to prevent a process from accessing memory it does not own
  - Limit register contains the range of logical addresses (each logical address must be less than the limit register)
  - Relocation register contains the value of the smallest physical address

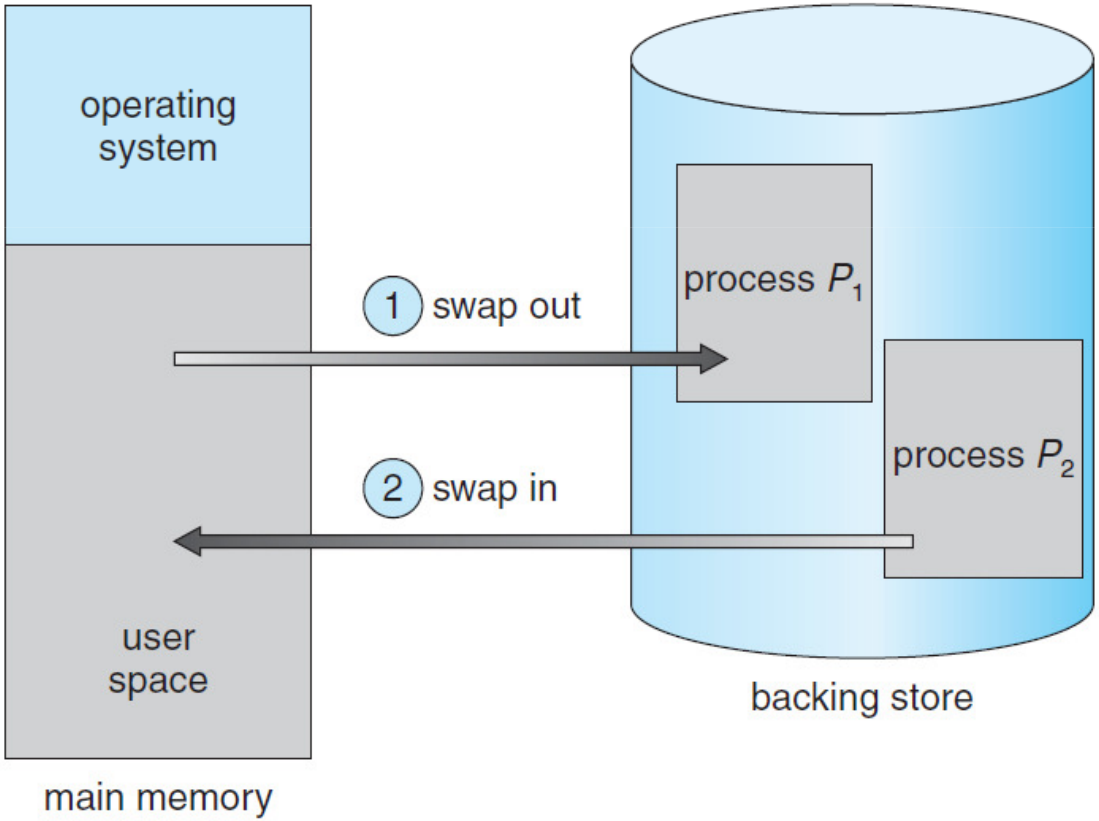


Swapping

- What if not all processes fit in memory?
  - Use an extreme form of context switch where some of the processes in memory are **temporarily swapped out of memory to a backing store**
  - Makes it possible for the total physical address space of all processes to exceed the real physical memory of the system
- **Backing store** – commonly a fast disk large enough to accommodate copies of all memory images for all users
  - A **ready queue** maintains the ready-to-run processes which have memory images on the backing store
- **Dispatcher** – checks whether a process scheduled to run is in memory
  - If it is not, and if there is no free memory region, the dispatcher **swaps out** a process currently in memory and **swaps in** the desired process
  - Context-switch time in such a swapping system is fairly high

Swapping

- Major part of swap time is transfer time (we must swap both out and in)
  - Total transfer time is directly proportional to the amount of memory swapped



Swapping

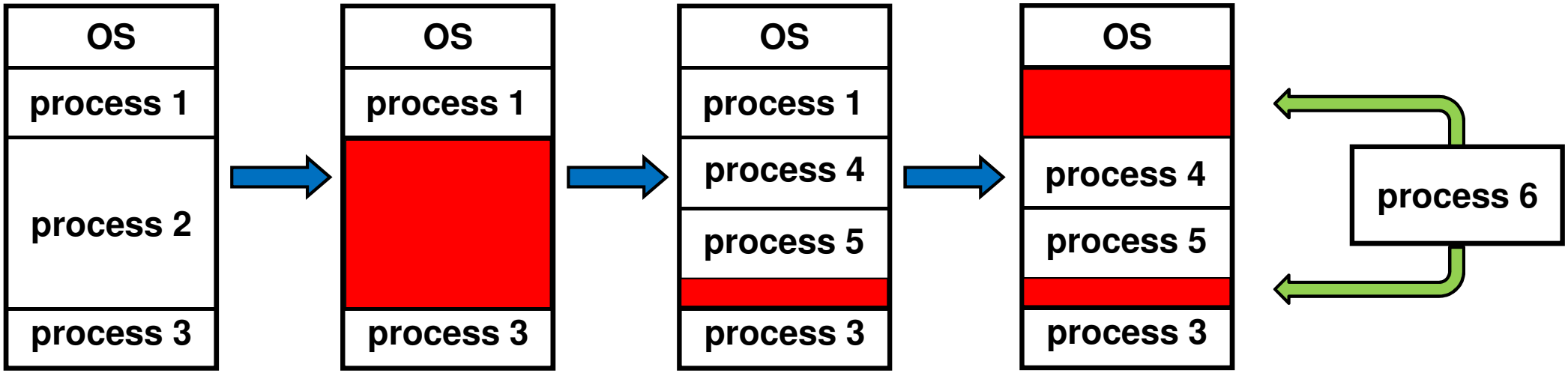
- Does a swapped out process need to swap back in to the same physical addresses?
  - No, if using **dynamic allocation** to change the relocation register
- Can a process waiting for a pending I/O operation be swapped out?
  - Yes, if we use **double buffering** and execute I/O operations only into kernel buffers (transfers between kernels buffers and process memory buffers then occur only when the process is swapped back in)
- Modified versions of swapping are found on many systems (e.g., UNIX, Linux, and Windows)
  - In one common variation, swapping is normally disabled but will start if the amount of free memory is extremely low
  - Another interesting variation involves swapping only portions of processes – rather than entire processes – to decrease swap time

Contiguous Memory Allocation

- Contiguous memory allocation is one early method where **each process is contained in a single contiguous section of memory**
- A simple scheme is to divide memory into several **fixed-sized partitions**
  - Each partition contains exactly one process
  - The degree of multiprogramming is bound by the number of partitions
- A generalization is the **variable-partition scheme**
  - Each partition still contains exactly one process but it is sized to the process' needs
  - The kernel keeps a table indicating which parts of memory are occupied and which are available (**memory holes**)

Variable-Partition Scheme

- A possible algorithm for the variable-partition scheme is:
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it (holes of various size can exist throughout memory)
  - When a process exits, it frees its partition and adjacent free partitions are combined in to larger holes
  - Memory is allocated until no available hole is large enough to hold a process – the kernel can then wait until a large enough block is available, or it can skip down the input queue to see whether some other process can be satisfied



Variable-Partition Scheme

- The problem of accommodating a process in memory given a list of free holes is a particular instance of the general **dynamic storage-allocation problem**. There are many solutions to this problem:
  - **First-fit**: allocate the first hole that is big enough (searching can start either at the beginning of the set of holes or at the location where the previous search ended)
  - **Best-fit**: allocate the smallest hole that is big enough (must search entire list unless the list is ordered by size; produces the smallest leftover hole)
  - **Worst-fit**: allocate the largest hole (must also search entire list; produces the largest leftover hole)
- Simulations have shown that:
  - Both first-fit and best-fit are better than worst-fit in terms of speed and storage
  - First fit is generally faster than best-fit

Internal Fragmentation

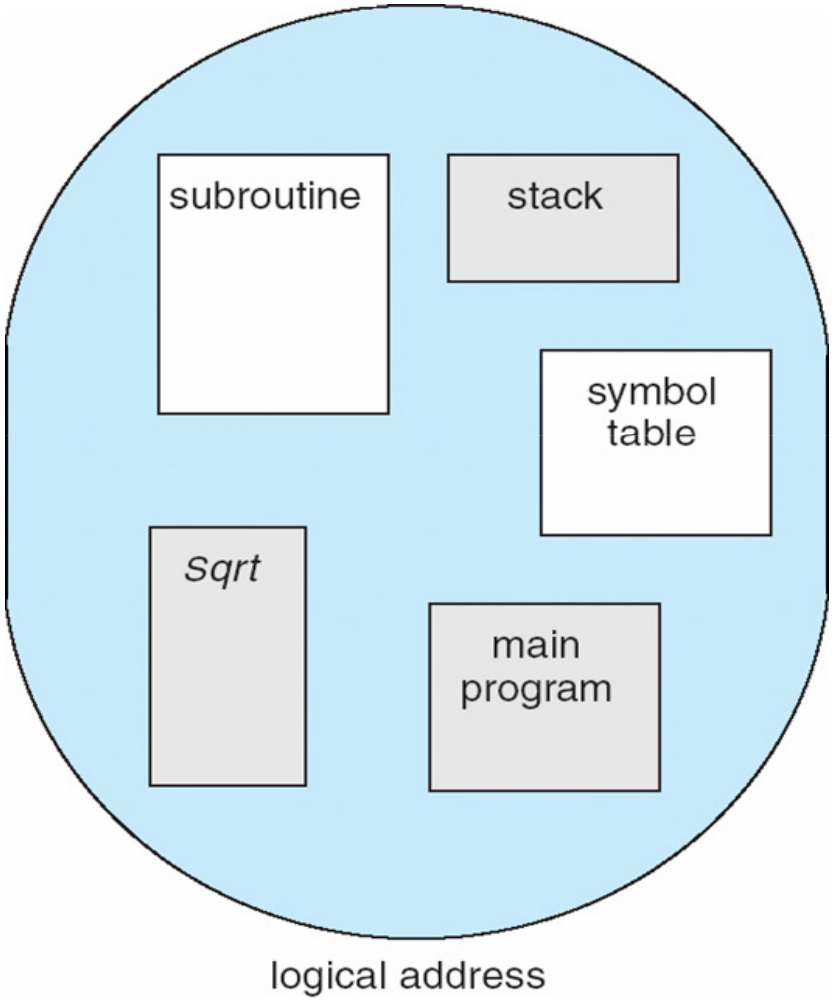
- **Internal fragmentation** exists when the memory allocated to a process is slightly larger than the requested memory
  - This size difference is **unused memory that is internal to a partition**
- Happens as a result of trying to avoid the overhead of keeping track of small holes
  - Consider a hole of 1,000 bytes and a process requesting 998 bytes, we are left with a small hole of only 2 bytes – better ignore hole than handle it
  - A general approach to avoid small holes is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size

External Fragmentation

- **External fragmentation** exists when there is enough total memory space to satisfy a request but the available space is not contiguous
  - Storage is fragmented into a large number of small holes
  - In the worst case, we could have a hole between every two processes
- A possible solution is to shuffle memory to **compact all free memory in one single large hole**
  - Only possible if dynamic allocation, done at execution time, and we are using double buffering for pending I/O operations
  - Can be very expensive
- Another possible solution is to use **noncontiguous memory allocation schemes**
  - Segmentation and paging are two of those schemes

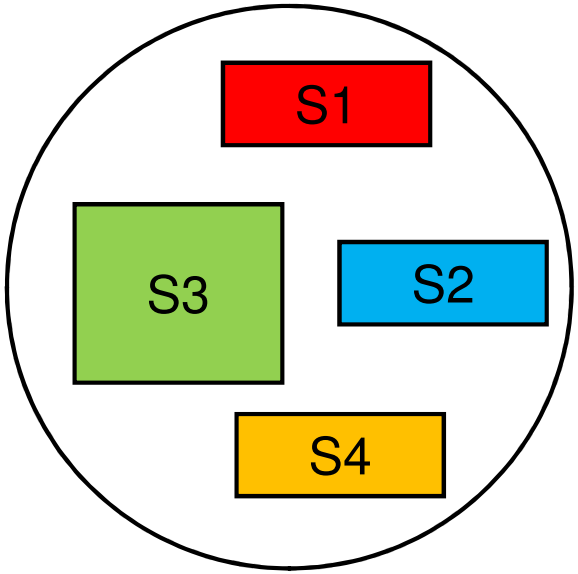
Programmer View of Memory

- Programmers view memory as a **collection of separate variable-sized logical units** (or segments) **with no necessary ordering** among them:
  - main program
  - procedures, functions
  - objects, methods
  - local variables, global variables
  - stack
  - symbol table
  - arrays

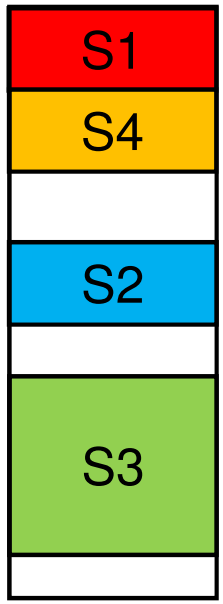


Segmentation

- Segmentation is a memory management scheme that fits the programmer view of memory
  - A logical address space is a collection of segments
  - Each segment is given a region of contiguous memory (has a base and a limit) and can reside anywhere in physical memory



logical address space



physical memory space

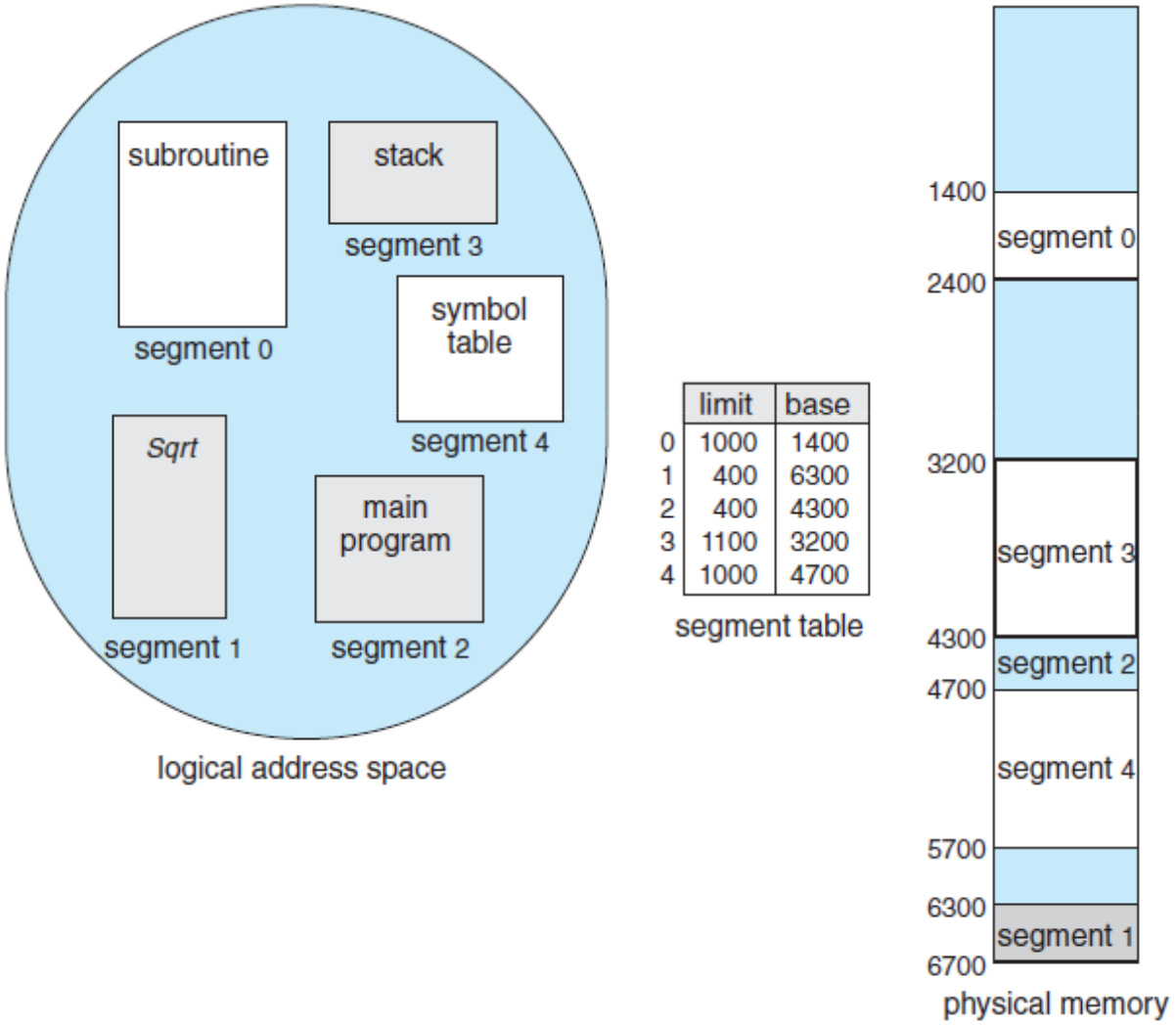
Segmentation

- Each segment has an **identifier (number)** and a **length**
- Logical addresses (within a segment) consist of tuples of the form:

```
<segment-number, offset-within-segment>
```

- A **segment table** maps two-dimensional logical addresses into one-dimensional physical addresses. Each entry includes:
  - **Segment base** – contains the starting physical address where the segment resides in memory
  - **Segment limit** – specifies the length of the segment

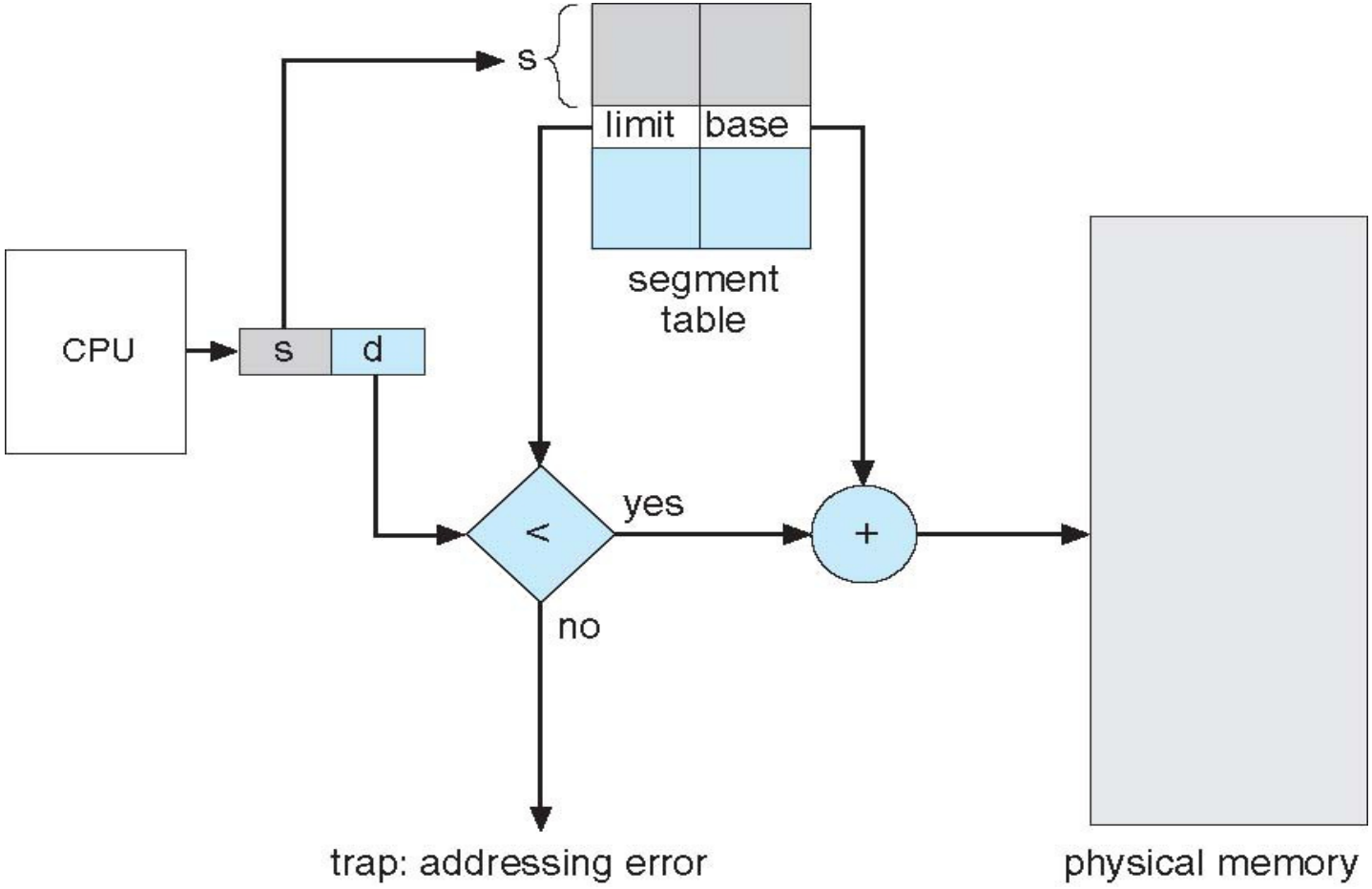
Segmentation



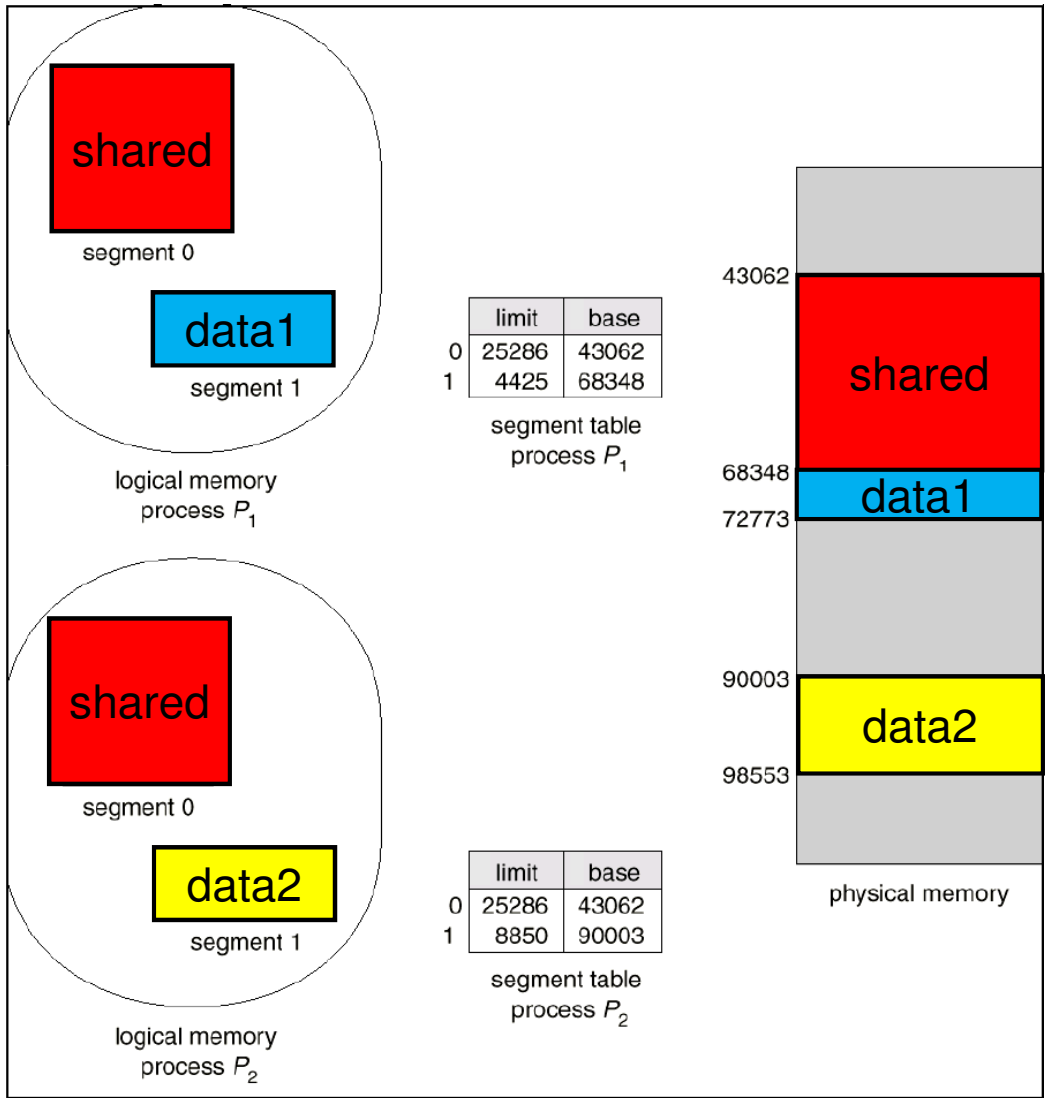
Segmentation Architecture

- Segment table resides in PCB
  - Segment table base register (STBR) points to the segment table of the current process
  - Segment table length register (STLR) indicates number of segments in use for the current process (segment number S is legal if S < STLR)
  - When a context switch occurs, the STBR and STLR registers are updated to the info in the new process's PCB
- Segment table entries also includes protection data, such as:
  - Validation bit (legal/illegal segment)
  - Read/write/execute/sharing privileges

Segmentation Architecture



Sharing Segments



Managing Segments

- When a new process is loaded into memory:
  - Create a new segment table and store it in the process's PCB
  - Allocate space in physical memory for all of the process's segments
- If there's no space in physical memory:
  - Compact memory (move segments, update bases) to make contiguous space
  - Swap one or more segments out to disk
- To enlarge a segment S:
  - If space above segment S is free, just update the segment's limit and use some of that space
  - Move segment S to a larger free space
  - Swap the segment above S to disk

Managing Segments

- Advantages:
  - Segments do not have to be contiguous
  - Segments can be swapped independently
  - Segments can be shared between different processes
- Disadvantages:
  - Suffers from the general dynamic storage-allocation problem (requires a memory allocation scheme like first-fit, best-fit, worst-fit, …)
  - Suffers from external fragmentation

Paging

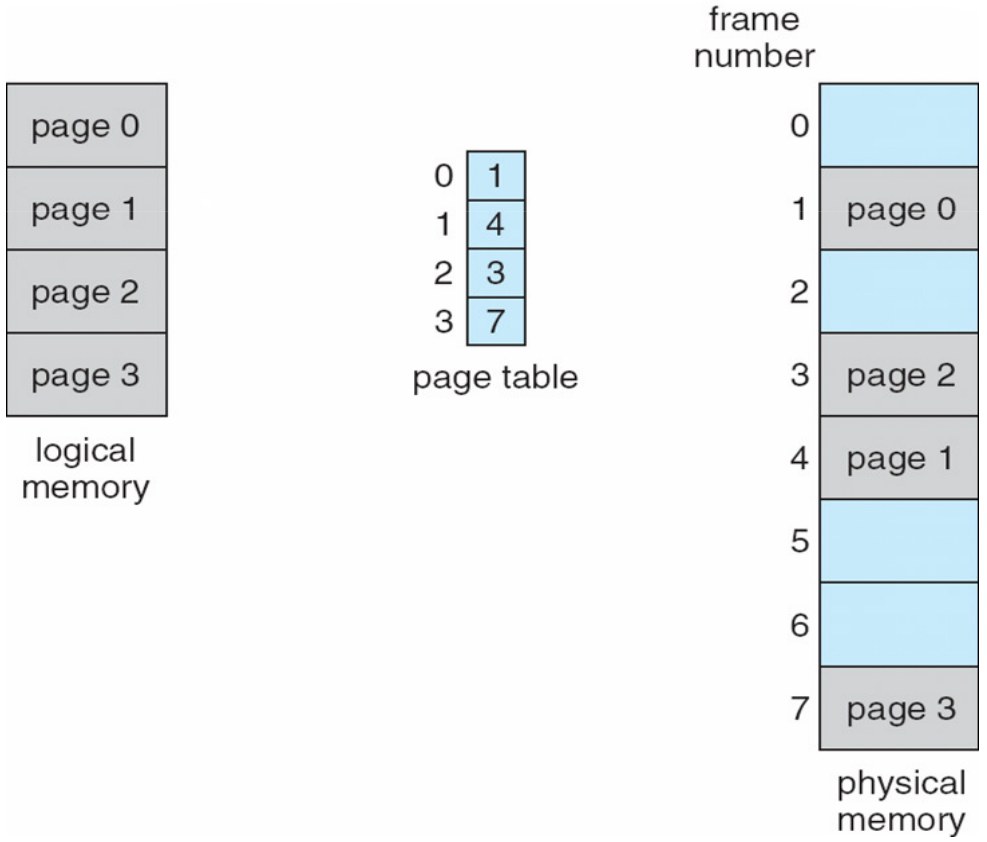
- Like segmentation, paging is another memory management scheme that permits the physical address space of a process to be noncontiguous
- Compared to segmentation, paging:
  - Makes allocation and swapping easier (no variable-size memory segments)
  - Avoids external fragmentation
  - No compaction required
- Because of its advantages, paging in its various forms is used in most operating systems, from mainframes to smartphones

Paging

- The basic idea is to organize memory in **fixed size (power of 2) blocks**
  - Divide **logical memory** into fixed size blocks called **pages**
  - Divide **physical memory** (and backing store) into blocks of same size called **frames**
- To load a process of size N pages, need to find N free frames
  - Pages do not have to be loaded into a contiguous set of frames
  - Need to keep track of free frames

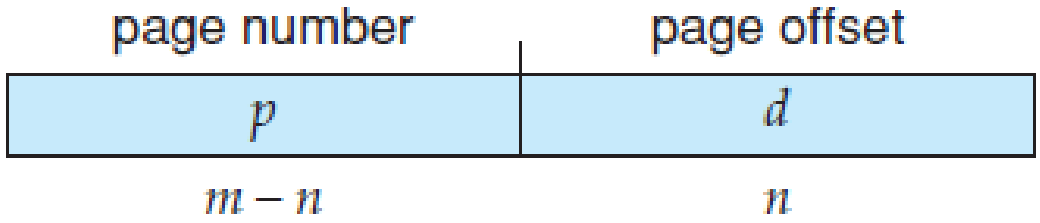
Paging Model of Memory

- A **page table** keeps track of every page in a particular process
  - Each entry contains the corresponding frame in physical memory
  - Translates logical to physical addresses



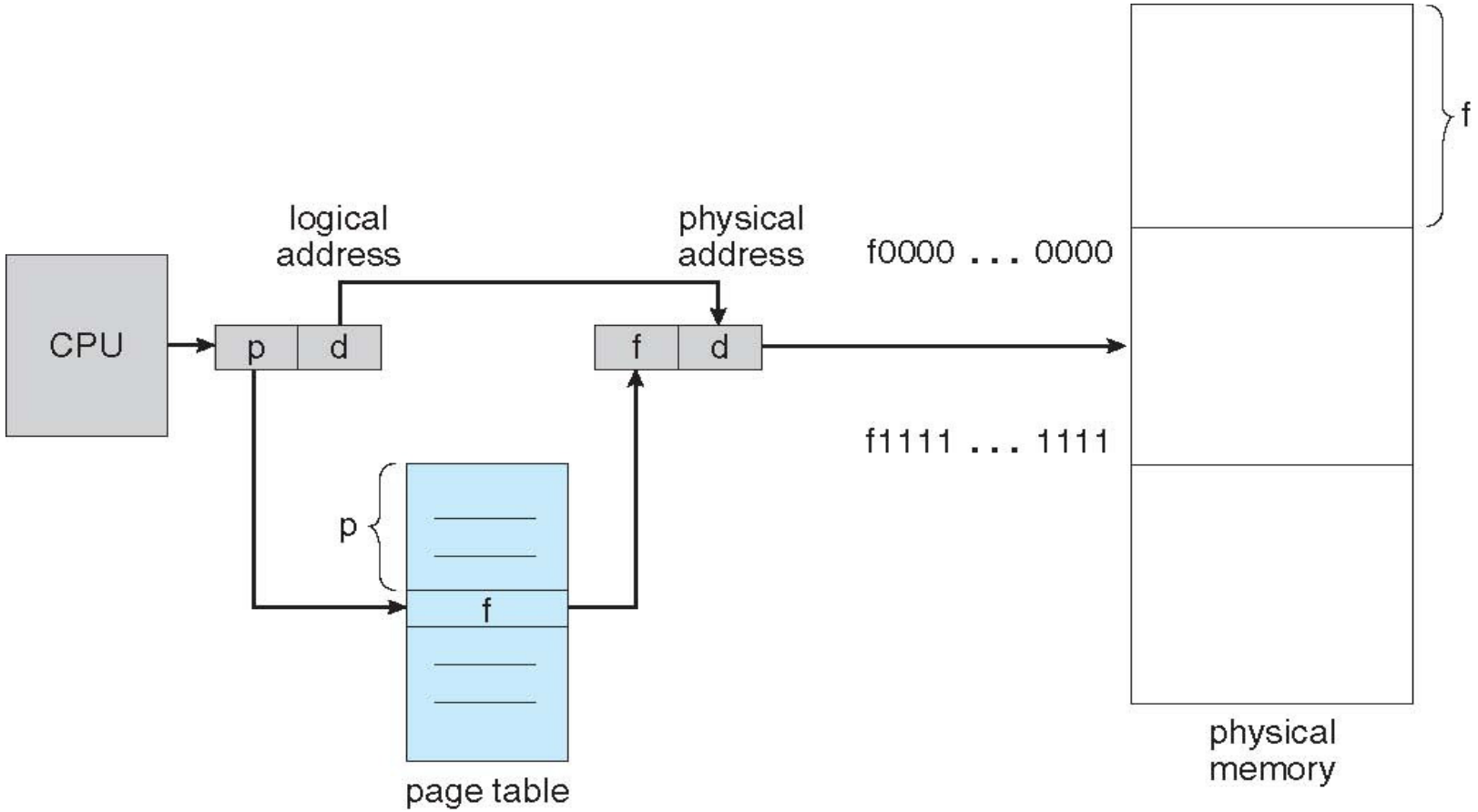
Address Translation Scheme

- Every logical address is divided into two parts:
  - **Page number (p)** – index into the page table
  - **Page offset (d)** – displacement within the page
- The base address in the page table combined with the page offset defines the physical memory address that is sent to the memory unit
- If the size of the logical address space is 2^m and a page size is 2^n bytes (m>n), then the high-order m–n bits of a logical address designate the page number (p), and the n low-order bits designate the page offset (d)



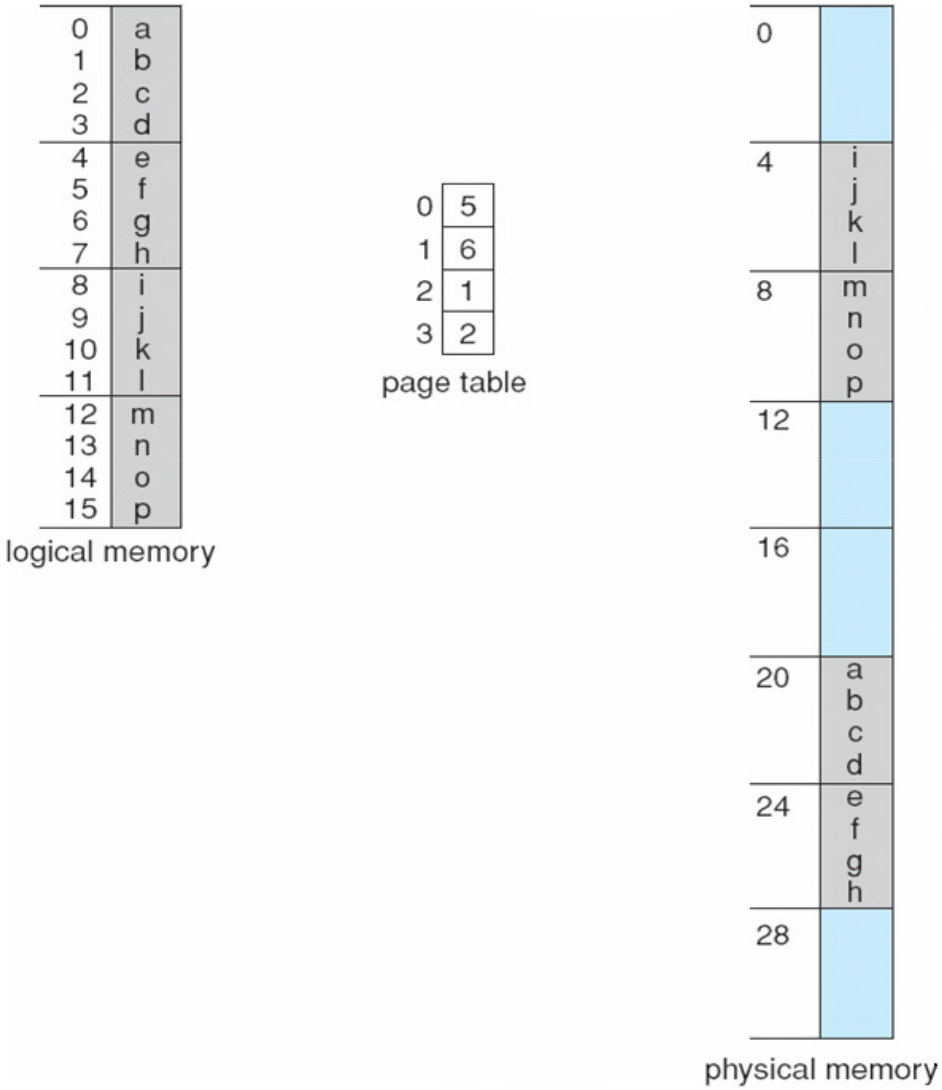
Paging Hardware

- Required hardware support is slightly less than for segmentation
  - No need to keep track of, and compare to, limit

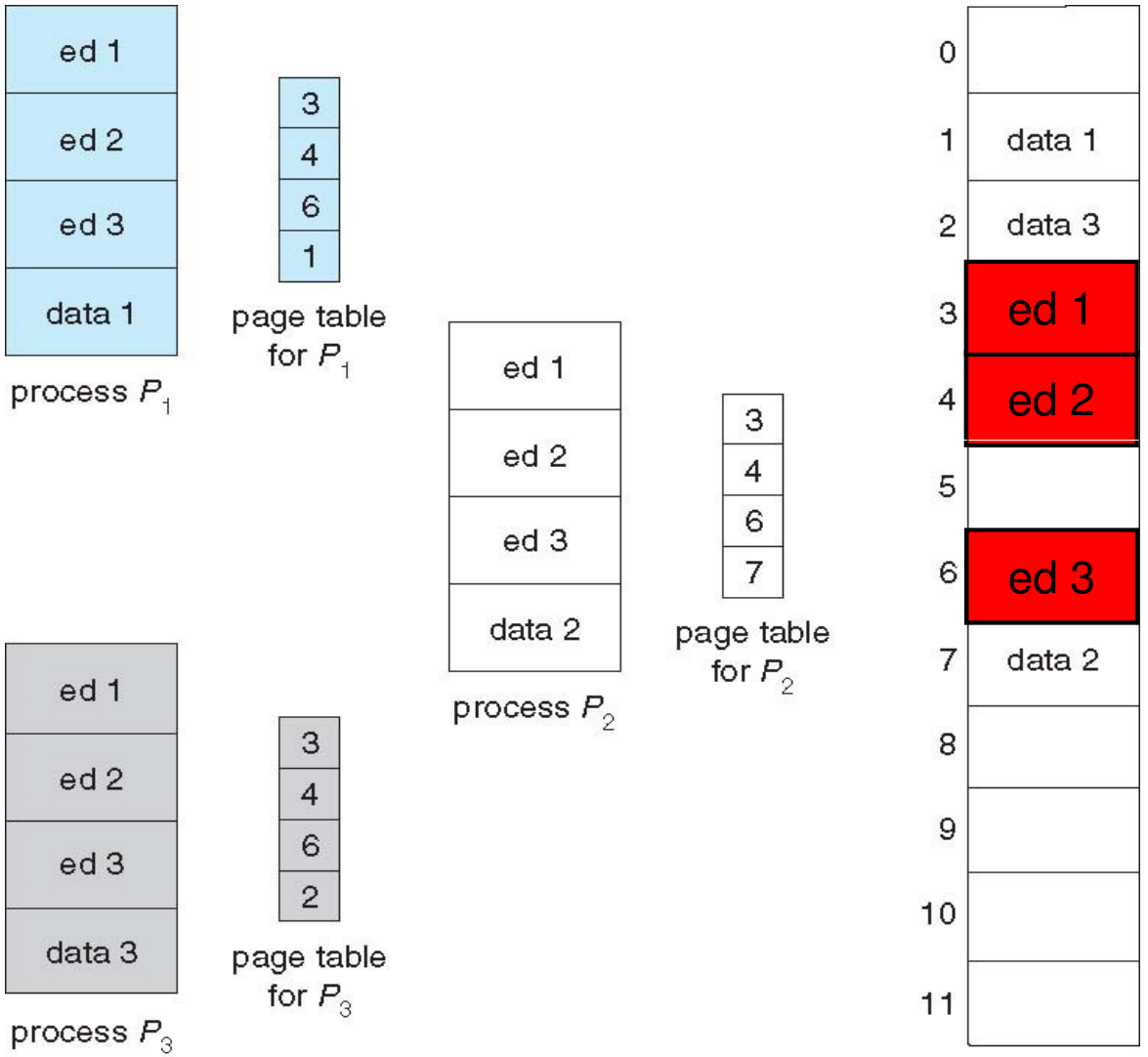


Example: 4-byte Pages and 32-byte Memory

- Logical address 2 (page 0 – offset 2)
  - Page table: page 0 → frame 5
  - Maps to physical address 22 = 5*4+2
- Logical address 5 (page 1 – offset 1)
  - Page table: page 1 → frame 6
  - Maps to physical address 25 = 6*4+1
- Logical address 15 (page 3 – offset 3)
  - Page table: page 3 → frame 2
  - Maps to physical address 11 = 2*4+3



Sharing Pages



Paging: Internal Fragmentation

- Should pages be as big as our previous segments?
  - No, can lead to lots of internal fragmentation
- Calculating internal fragmentation (example)
  - Page size: 2,048 bytes
  - Process size: 72,766 bytes = 35 pages + 1,086 bytes = 36 frames
  - Internal fragmentation: 2,048 – 1,086 = 962 bytes
- Worst case fragmentation
  - N pages + 1 byte = N+1 frames
  - Internal fragmentation: almost an entire frame
- On average fragmentation
  - One-half page per process

Paging: Page Size

- So, are small page sizes desirable?
  - No, since each page table entry takes memory to track (this overhead is reduced as the size of the pages increases)
  - Also, disk I/O is more efficient when the amount of data being transferred is larger
- Typically, today, pages are between 4 KB and 8 KB in size
  - Some systems support even larger page sizes
  - Some other support multiple page sizes (e.g., Solaris uses page sizes of 8 KB and 4 MB)
  - Researchers are now developing support for variable on-the-fly page size

Page Table Implementation

- Modern computers allow the page table to be very large
  - Consider a 32 bit logical address space, if page size is 4KB (2^{12}) the page table could have up to 2^{20} (approximately 1 million) page entries, each maybe 4 bytes (2^2) long for a total memory for the page table of 4MB (2^{22})
- The use of fast registers to implement the page table is not feasible, thus the page table is also kept in the process' logical memory
  - Page table base register (PTBR) points to the page table
  - Page table length register (PTLR) indicates size of the page table

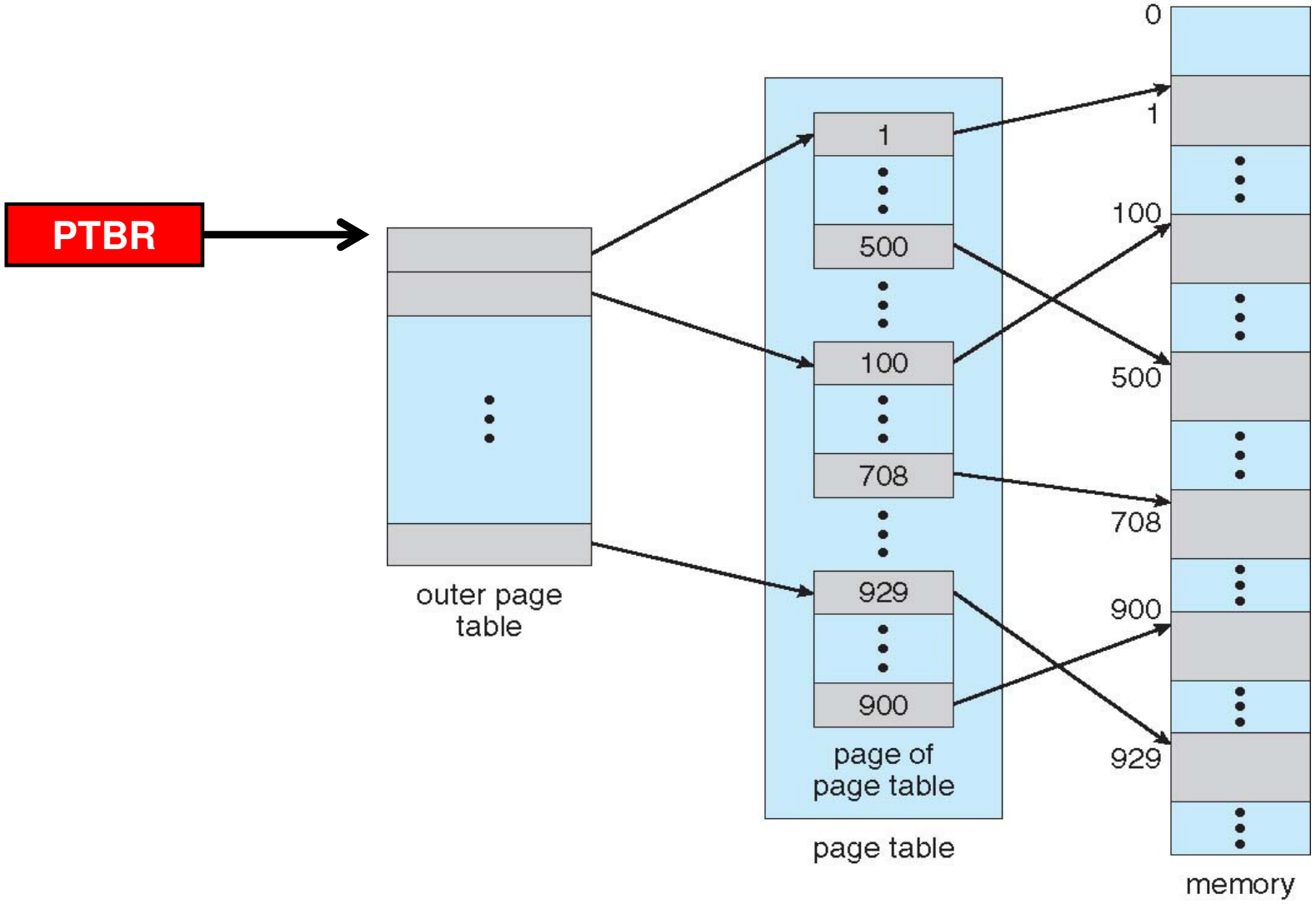
Page Table Implementation

- **Problem I:** memory required to store the page table costs a lot
  - Don't want to allocate the page table contiguously in main memory
  - **Solution:** hierarchical paging
- **Problem II:** every memory reference requires two (or more) memory accesses
  - One access for the page table entry and another for the actual physical address
  - **Solution:** translation look-aside buffer (TLB)

Hierarchical Paging

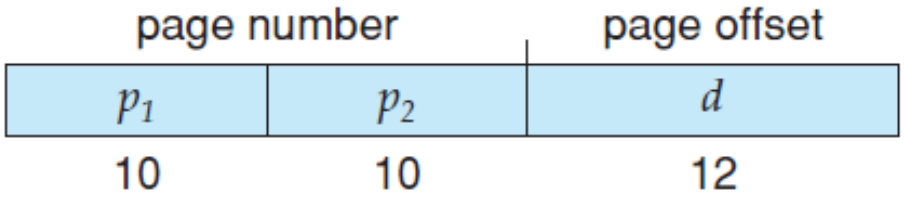
- Break up the page table into a **tree of page tables** scheme in which **each page table is itself also paged**
- **Pros:** only need to allocate as many page table entries as we need
  - Size is proportional to usage, don't need every 2nd-level tables for sparse address spaces
  - Even when exist, if not in use, 2nd-level tables can reside on disk
- **Cons:** every memory reference requires an extra memory access per page table level
  - Seems very expensive!

Two-Level Page Table Scheme

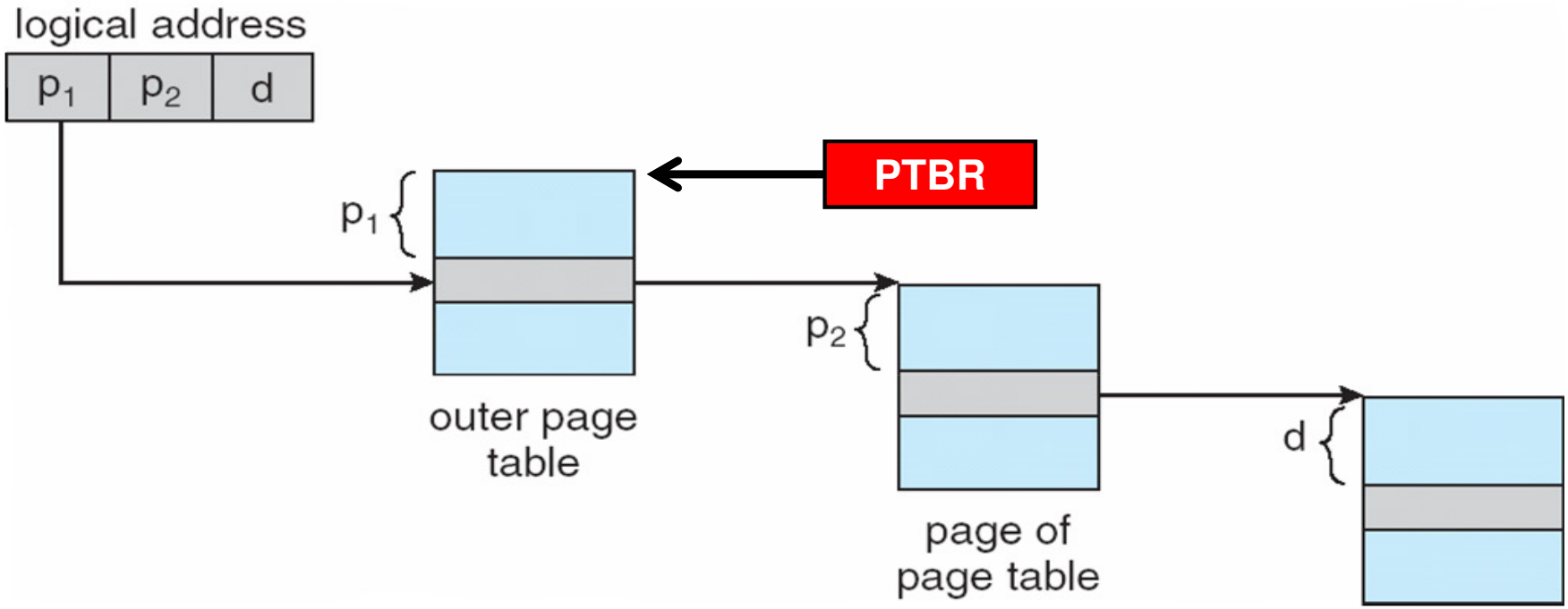


Two-Level Page Table Scheme

- Consider again a 32-bit logical address space and a page size of 4 KB
  - A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into a
  - 10-bit page offset (2^{10} entries * 2^2 entry size (32 bits) = page size of 4 KB)
  - 10-bit page number
- Thus, a logical address is as follows:
  - p1 is an index into the outer page table
  - p2 is the displacement within the page of the inner page table

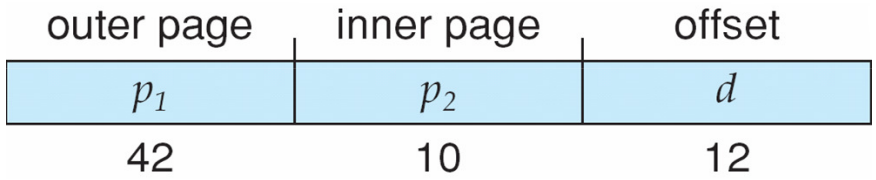


Address Translation Scheme

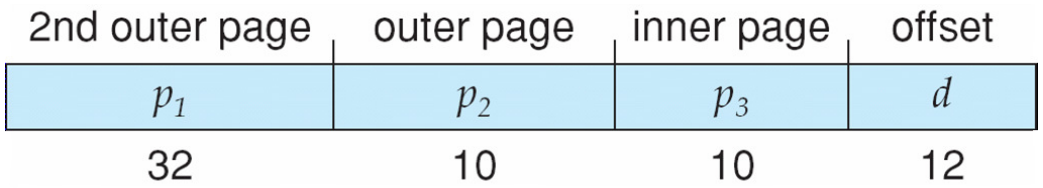


64-bit Logical Address Space

- For a 64-bit logical address space, a two-level paging scheme might be no longer appropriate
  - If the page size is 4KB (2^{12}), the page table consists of up to 2^{52} entries
  - A two-level paging scheme with 2^{10} inner pages of 4-byte (2^2) entries, requires an outer page table with 2^{42} entries (2^{44} bytes in size)



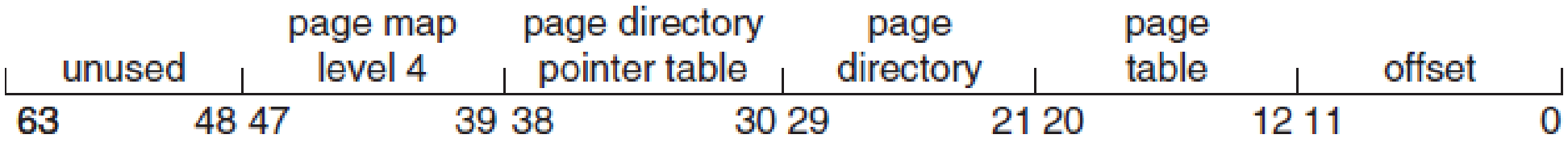
- One solution is to add a 2nd outer page table



  - But the 2nd outer page table is still 2^{34} bytes in size
  - And possibly 4 memory access to get to one physical memory location!!!

64-bit Physical Address Space

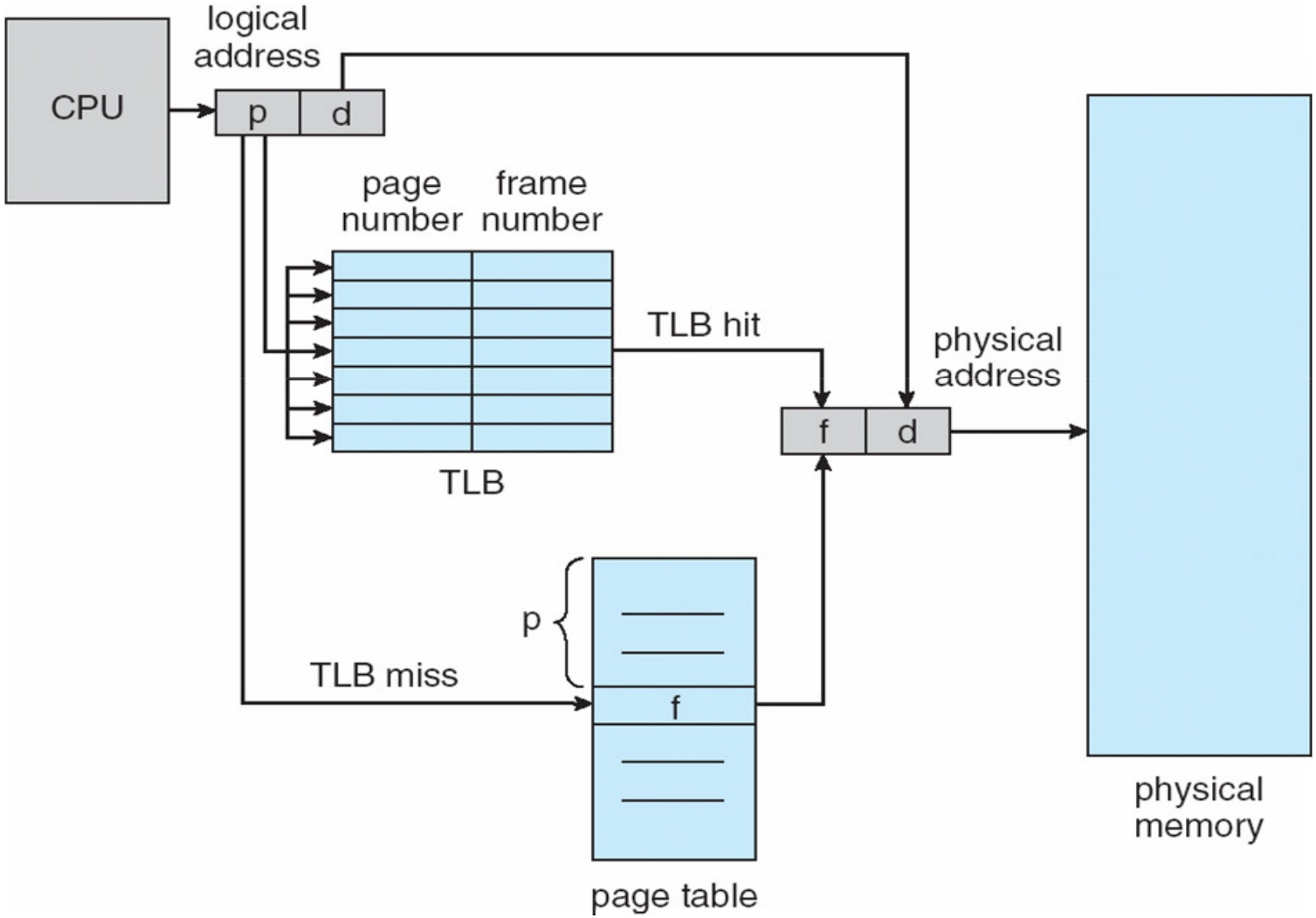
- For a 64-bit physical address space, inner pages consist of 8-byte (2^3) size entries
  - If the page size is 4KB (2^{12}), inner pages are represented by a 9-bit page offset (2^9 entries * 2^3 entry size (64 bits) = page size of 4 KB)
- In practice, far fewer than 64 bits are used for address representation in current modern architectures
  - For example, the x86-64 architecture implements a 48-bit address space using four levels of paging hierarchy



TLB: Translation Look-aside Buffer

- The page table memory access problem is solved by the use of a special small fast-lookup hardware cache called **translation look-aside buffer**
  - TLBs are typically small (between 64 to 1,024 entries in size)
- Each entry in the TLB consists of a **key** and a **value**
  - Given an item to the TLB, the item is compared with all keys simultaneously and if the item is found, the corresponding value field is returned
  - TLB lookup is part of the instruction pipeline, adding no performance penalty
- A TLB contains only a few page table entries
  - If the page number is found (**TLB hit**), its frame number is immediately available and is used to access memory
  - If the page number is not found (**TLB miss**), a memory reference to the page table must be made and the page and frame number are then added to the TLB for faster access next time

Paging with TLB



Effective Access Time (EAT)

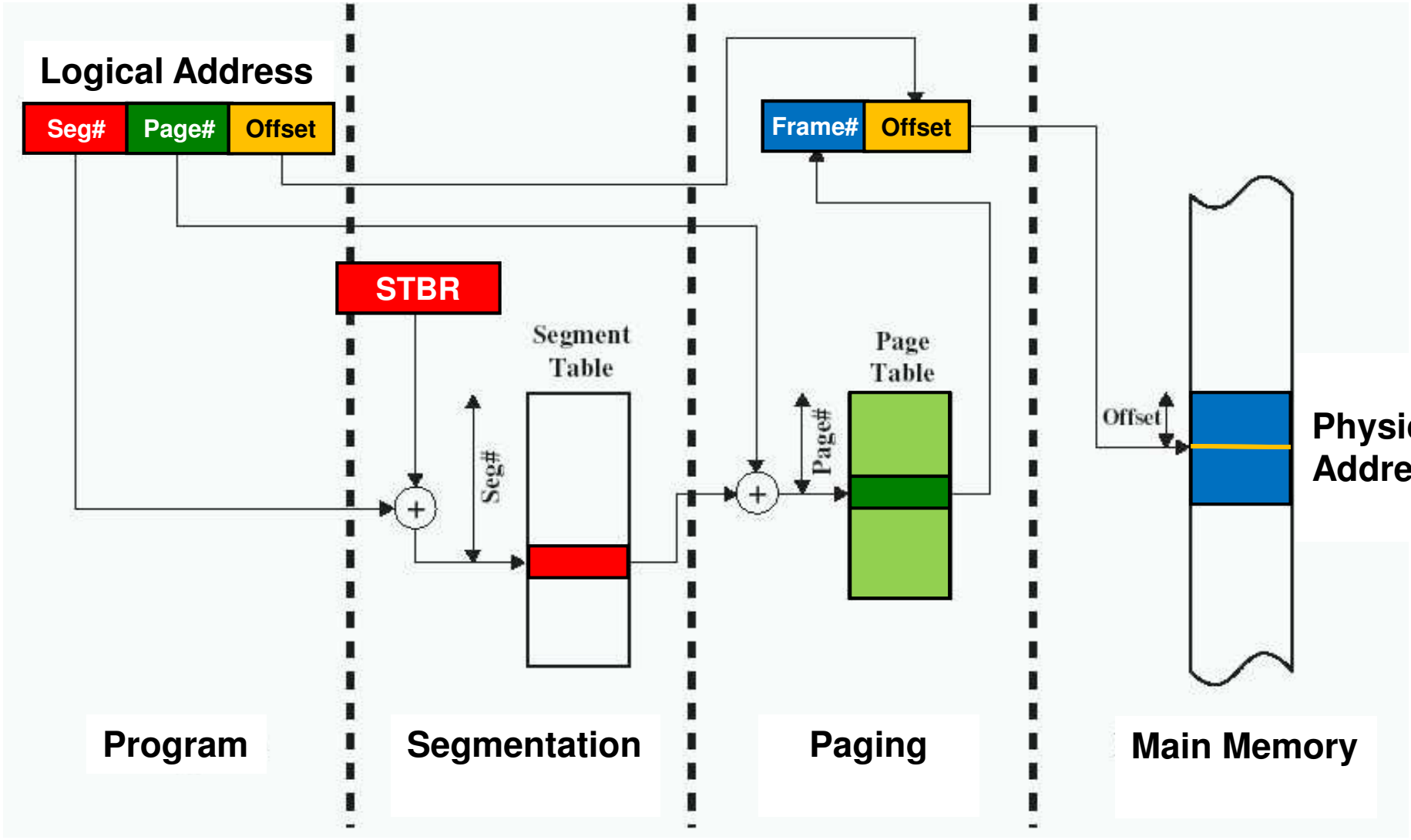
- EAT = (Hit Ratio x Hit Time) + (Miss Ratio x Miss Time)
- Consider the following scenario:
  - Hit Ratio = 80%
  - TLB access time = 20 ns
  - Memory access time = 100 ns
  - EAT = 0.8 * (20 + 100) + 0.2 * (20 + 100 + 100) = 96 + 44 = 140 ns
- Consider a more realistic scenario:
  - Hit Ratio = 98%
  - TLB access time = 20 ns
  - Memory access time = 100 ns
  - EAT = 0.98 * (20 + 100) + 0.02 * (20 + 100 + 100) = 117.6 + 4.4 = 122 ns

Paging and Segmentation

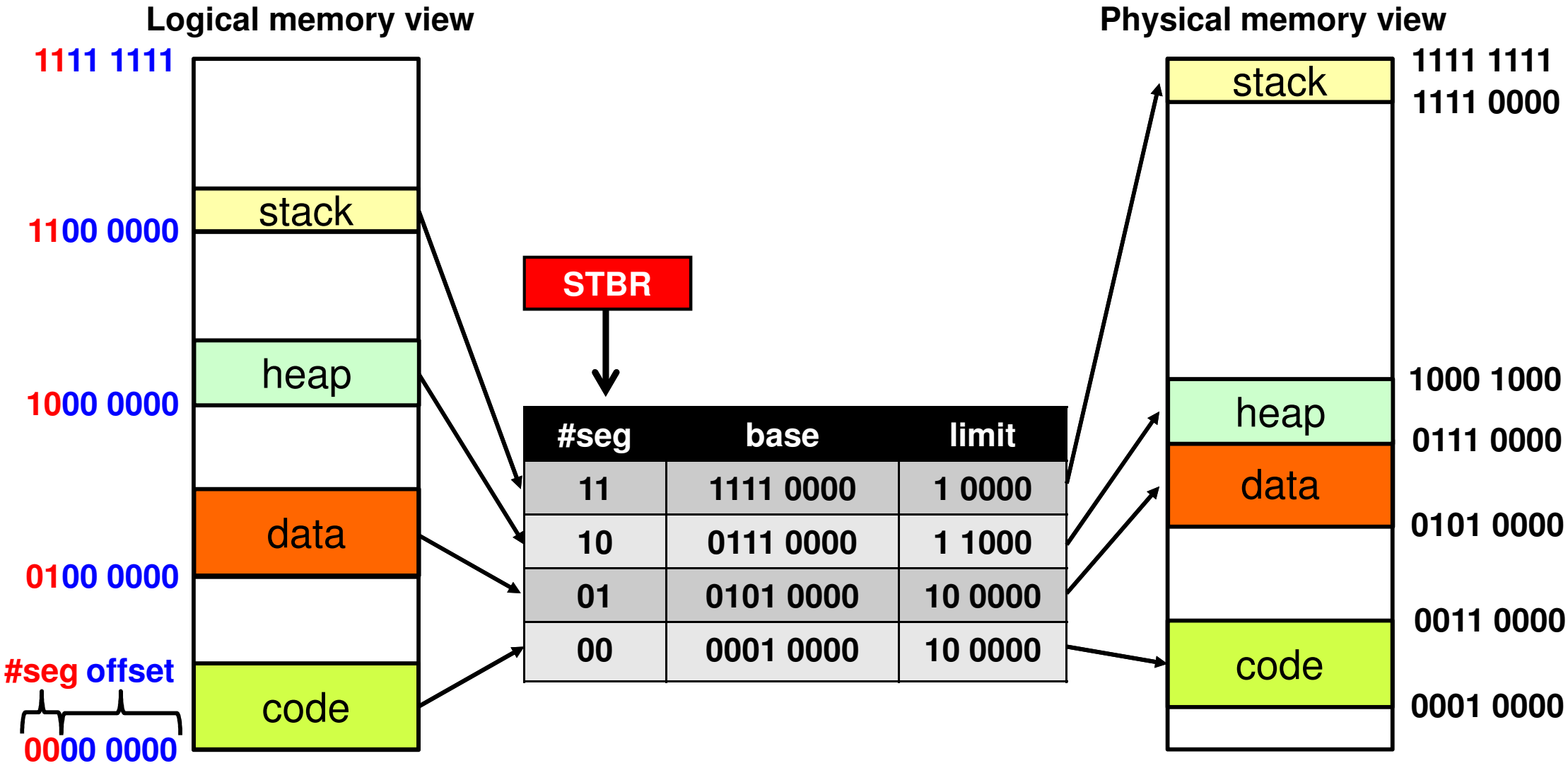
- We can also combine paging and segmentation in two levels of mapping
  - Process is view as a set of variable-size logical segments
  - Each segment is then divided into fixed-size pages
- Logical addresses consist of tuples of the form:

  `<segment-number, page-within-segment, offset-within-page>`
- Implementation
  - One segment table per process plus one page table per segment
  - Avoids external fragmentation
- Sharing can happen at both levels
  - Share a complete segment by having same base in two segment tables
  - Share a frame by having same frame reference in two page tables

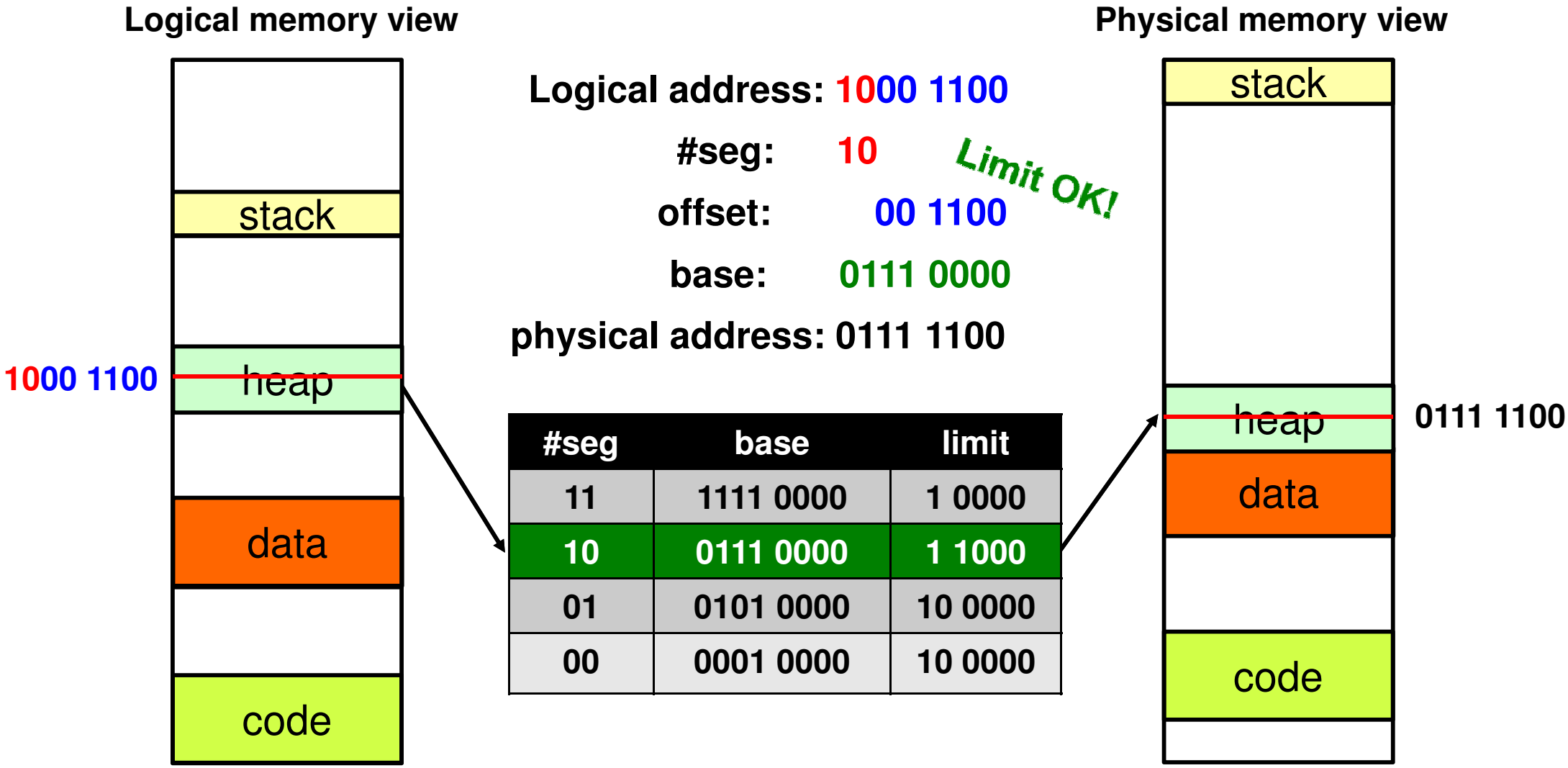
Paging and Segmentation



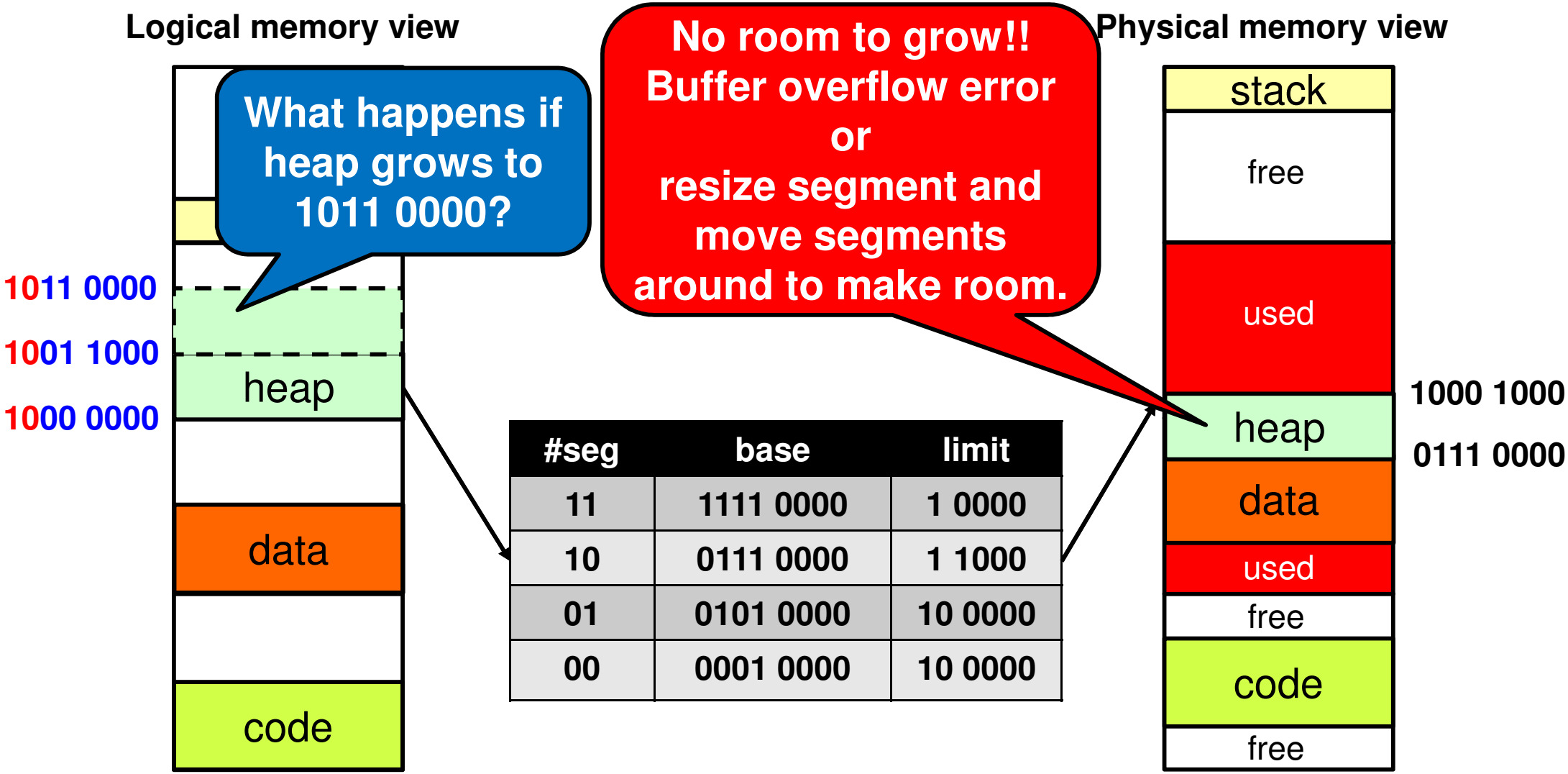
Recap: Segmentation



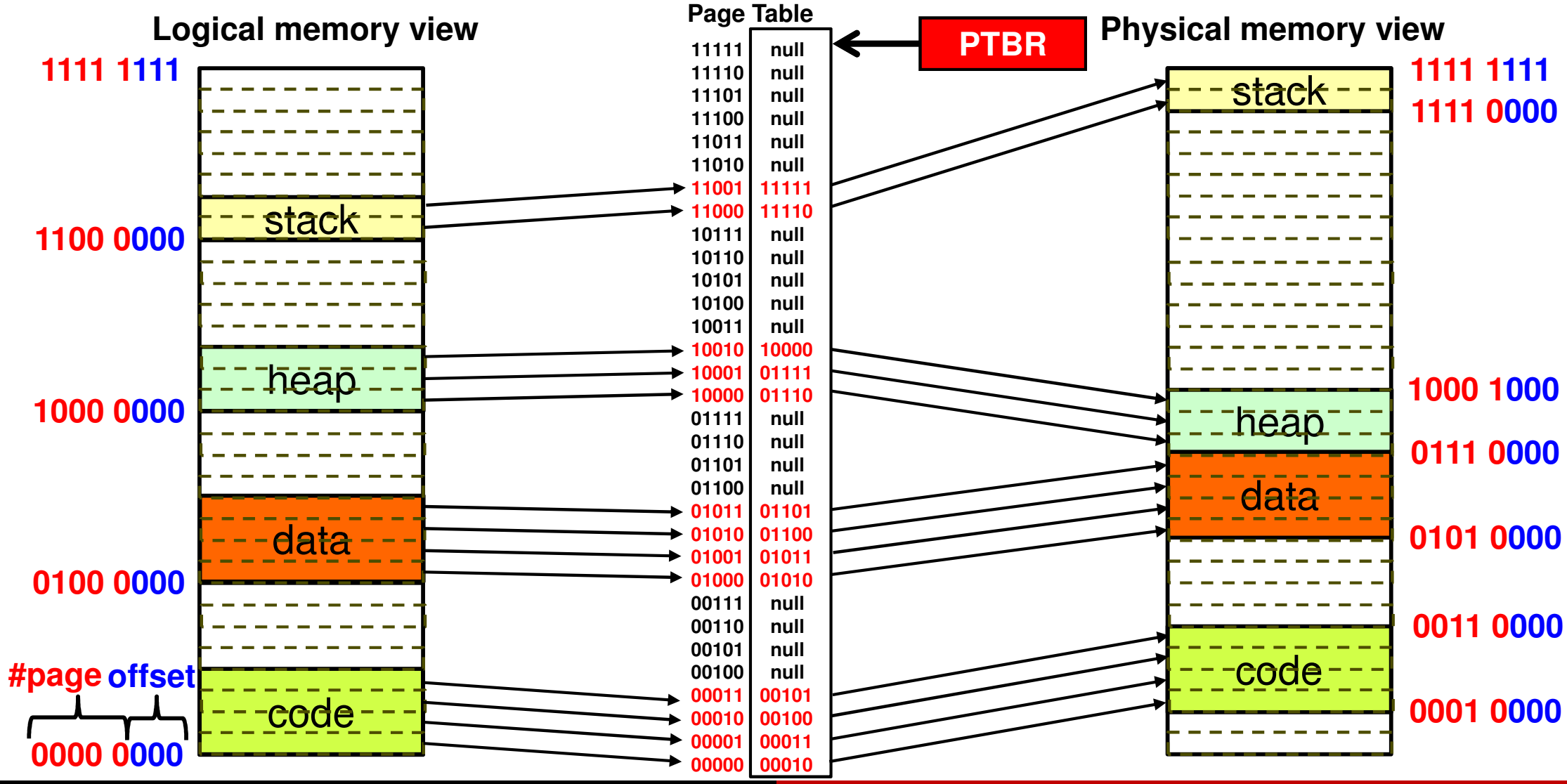
Recap: Segmentation



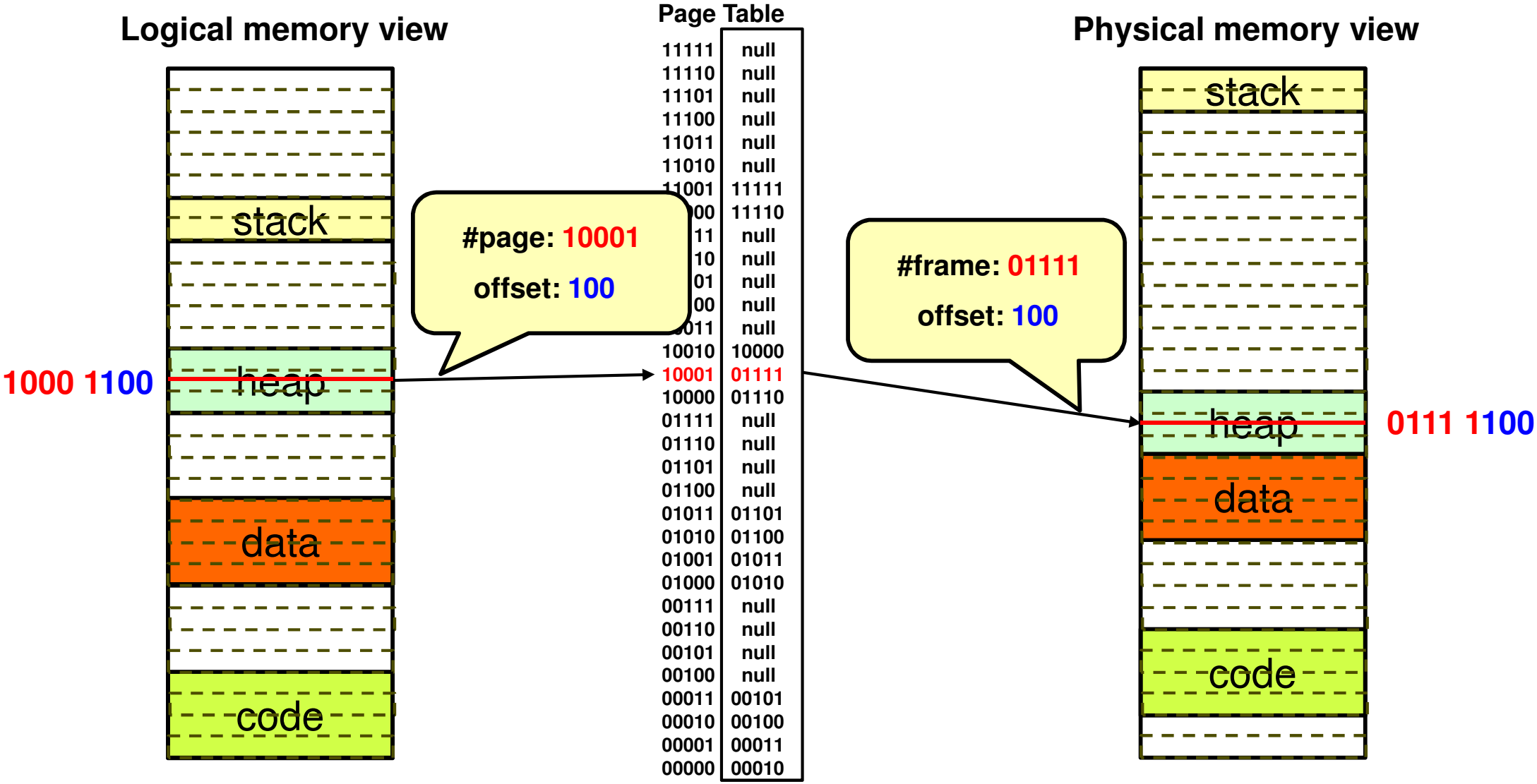
Recap: Segmentation



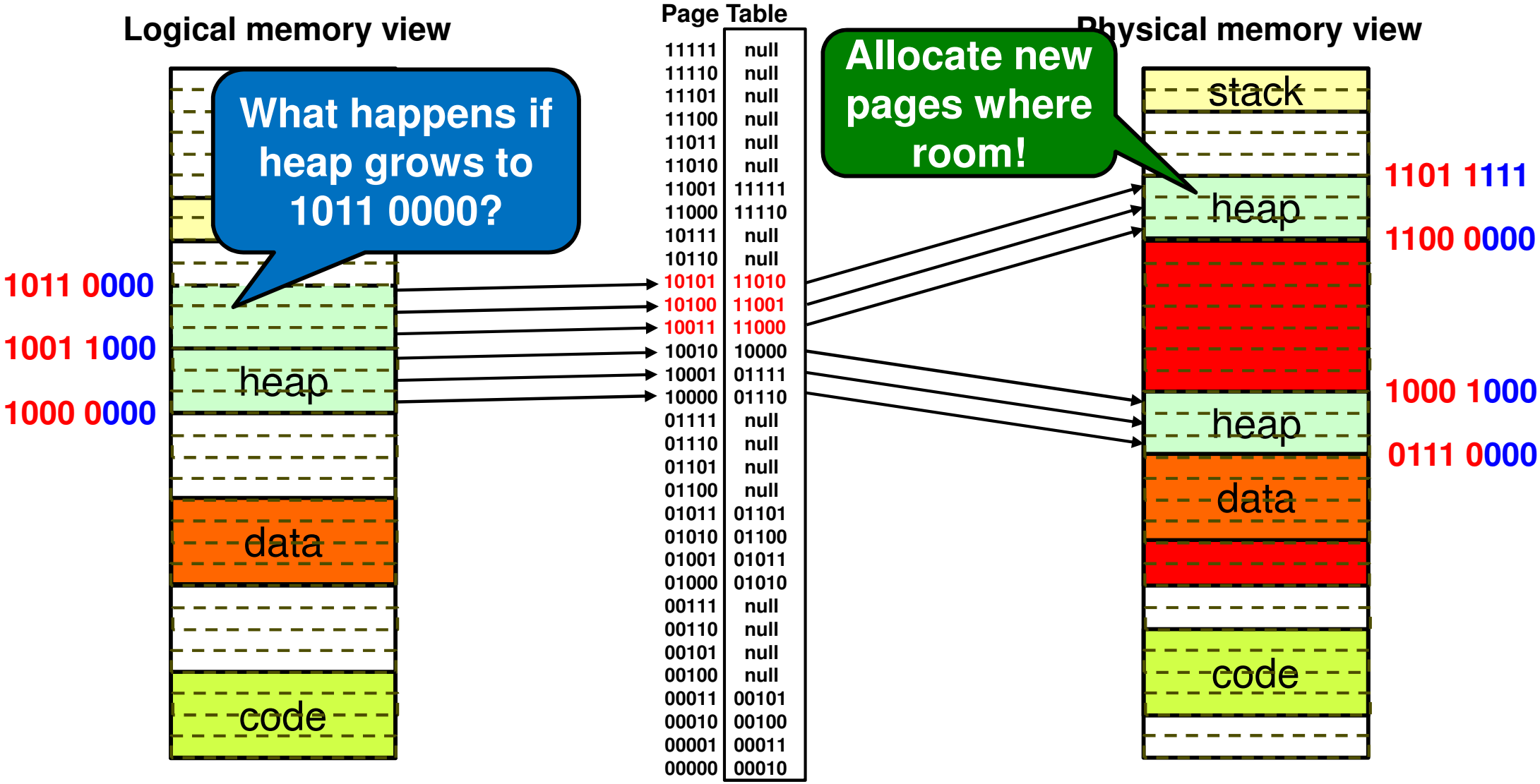
Recap: Paging



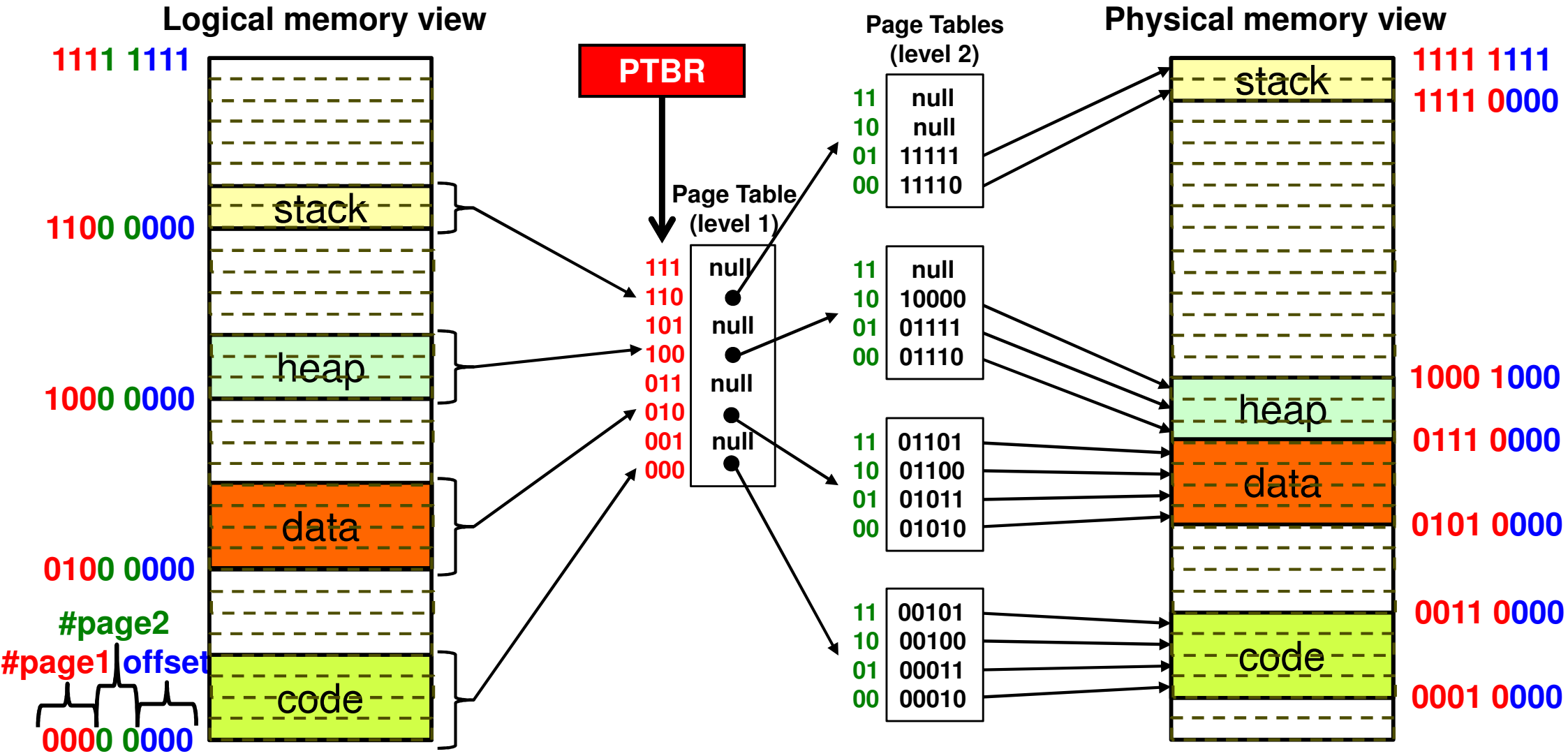
Recap: Paging



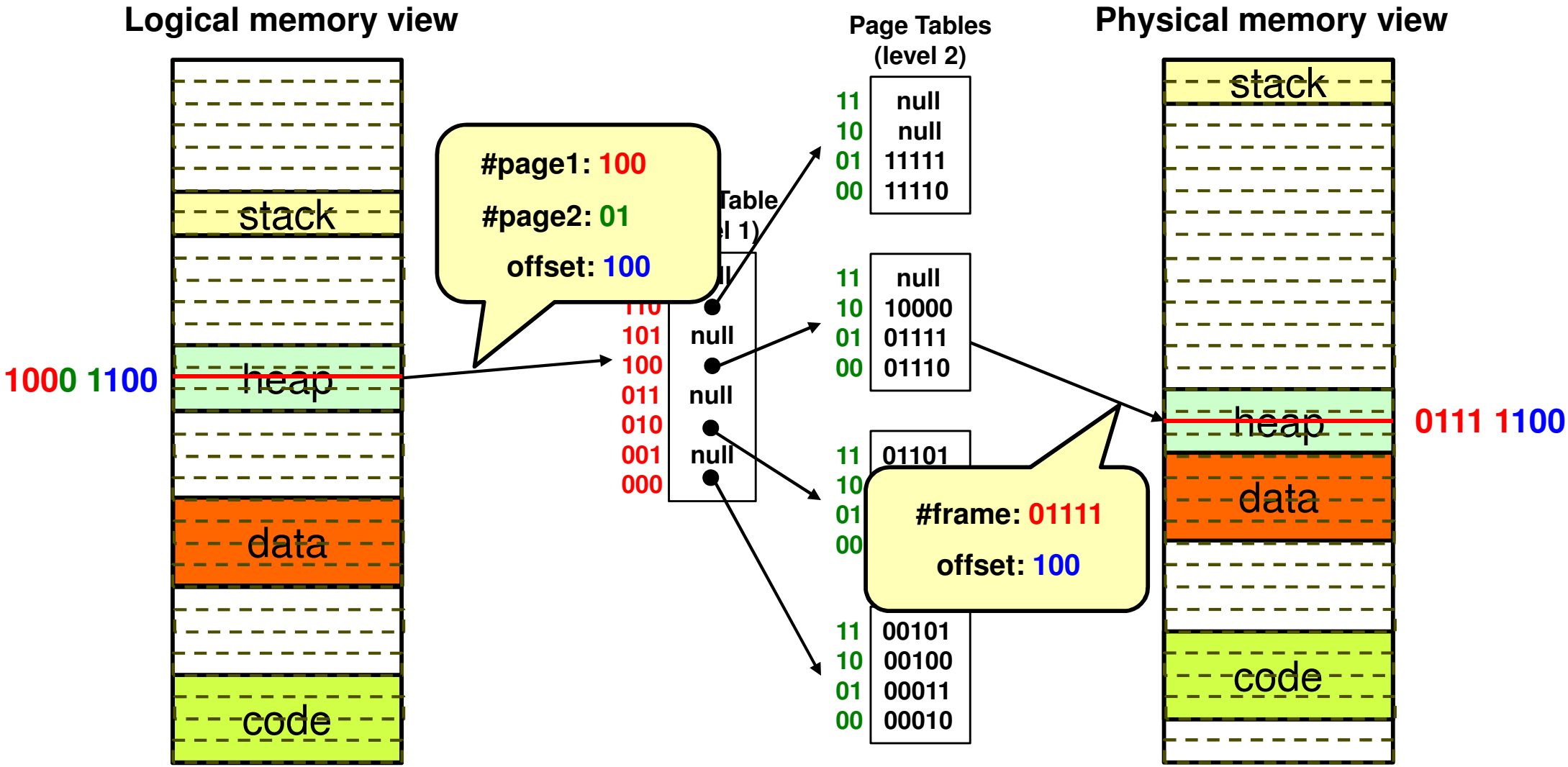
Recap: Paging



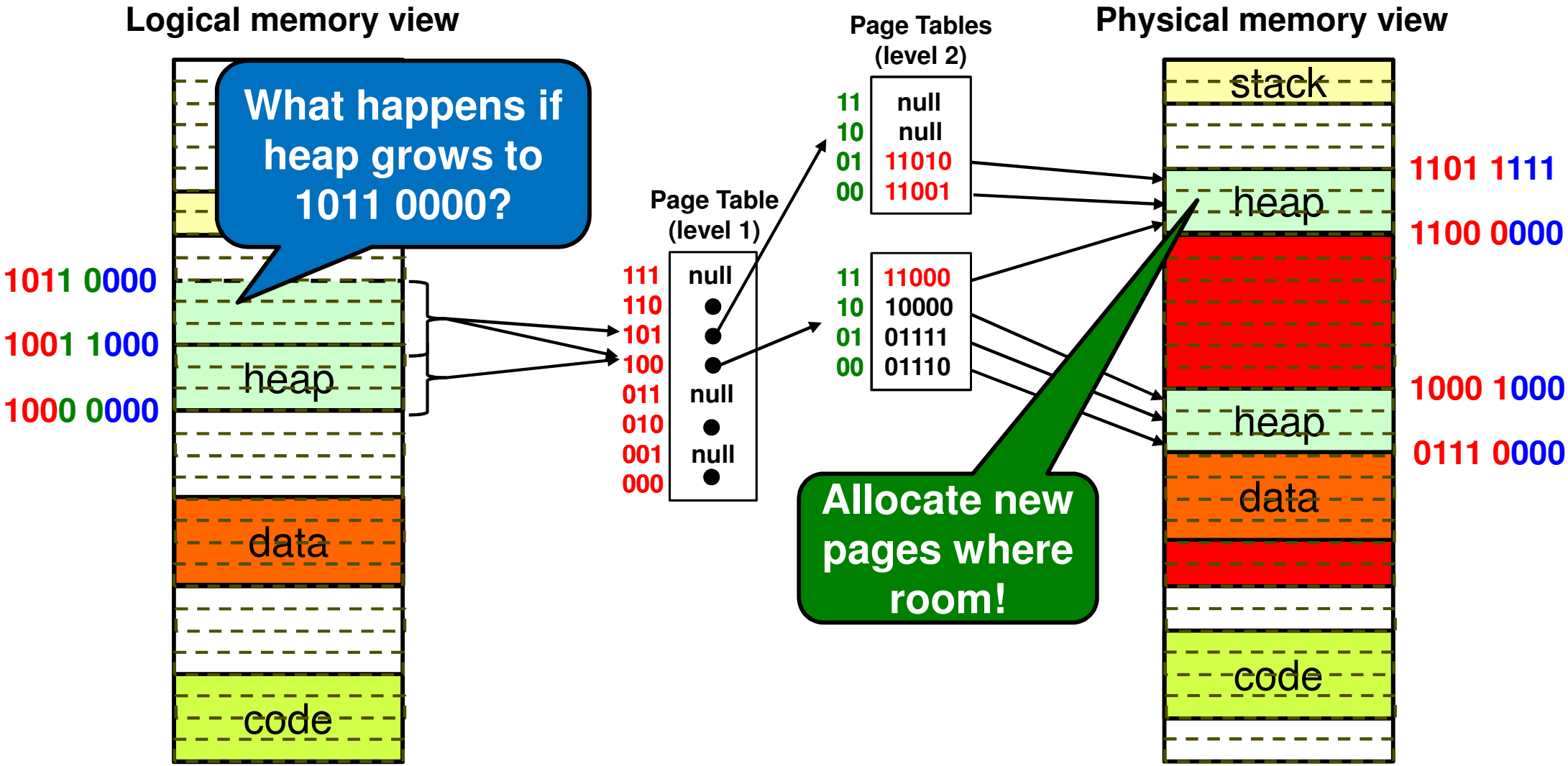
Recap: Two-Level Paging



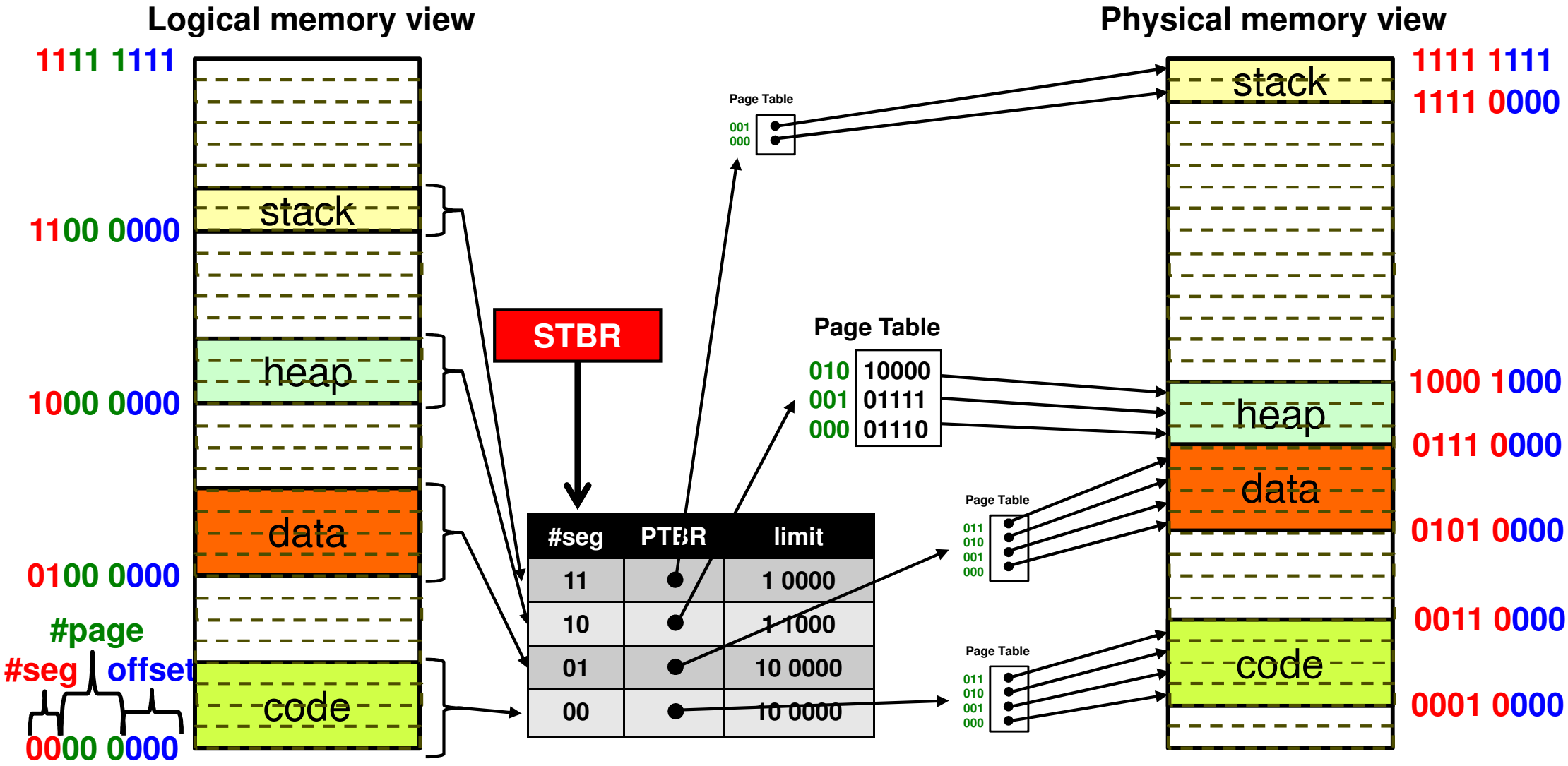
Recap: Two-Level Paging



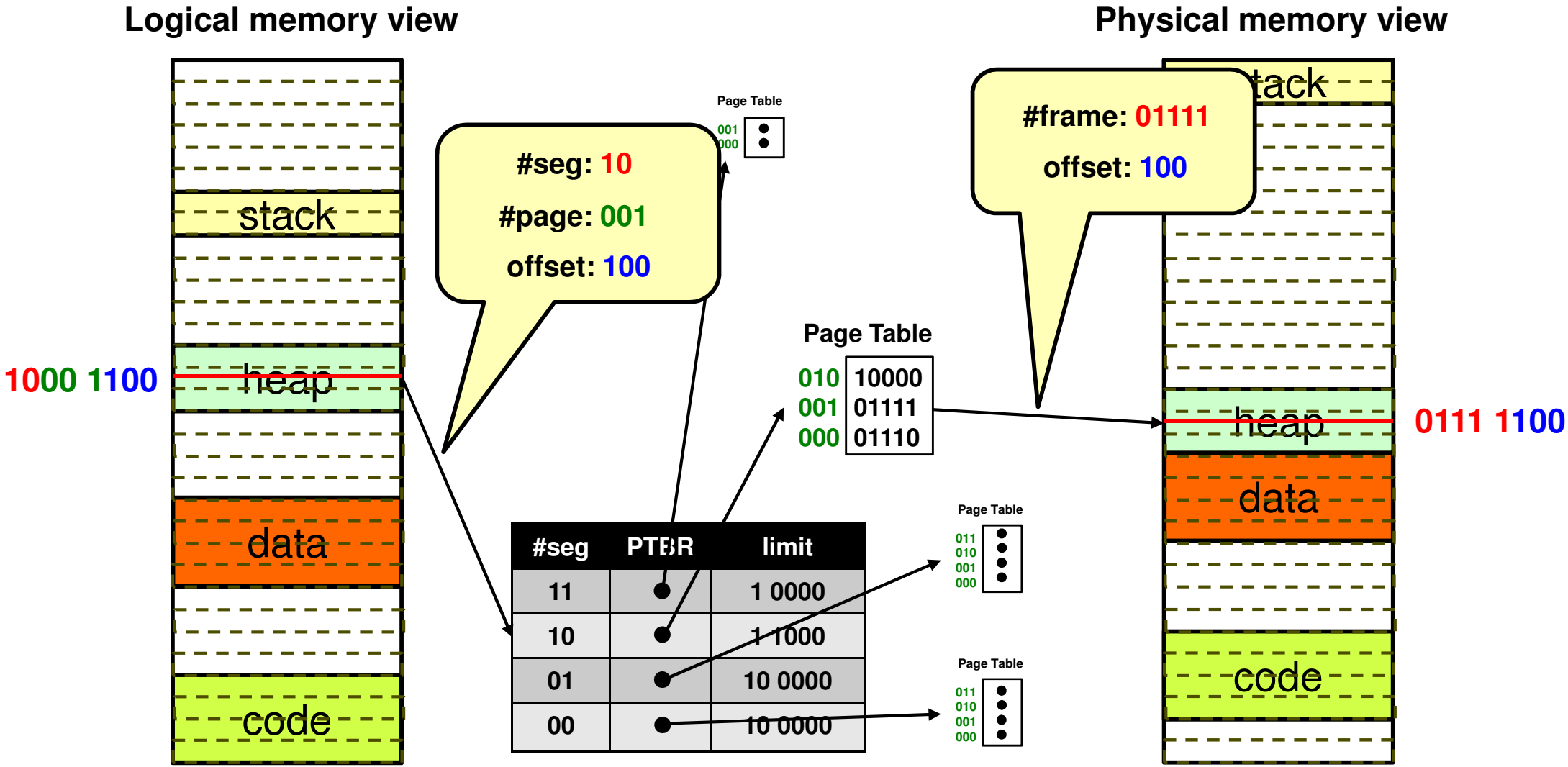
Recap: Two-Level Paging



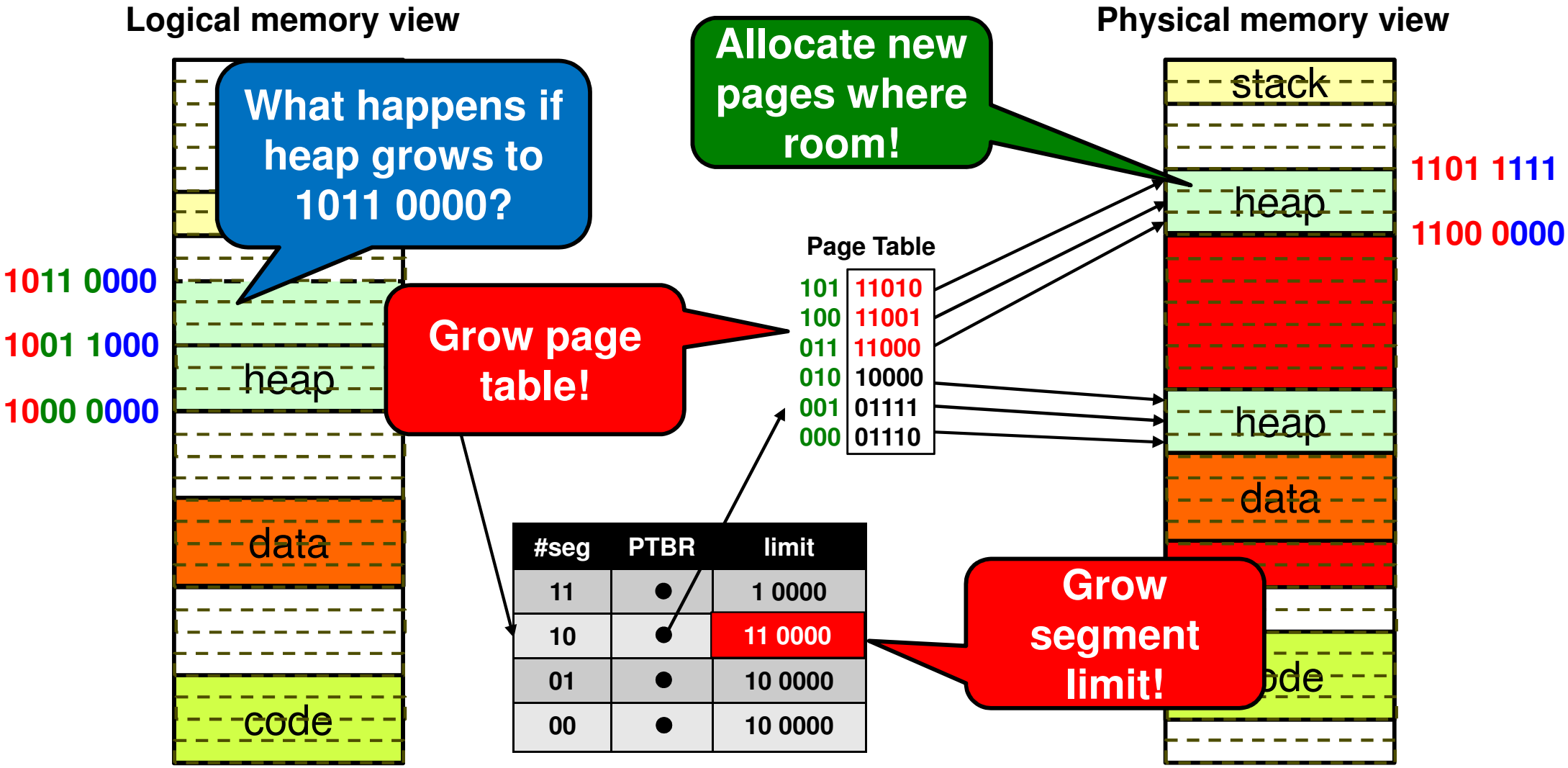
Recap: Paging and Segmentation



Recap: Paging and Segmentation



Recap: Paging and Segmentation



Recap: Paging with TLB