Operating System Concepts

Lecture 24: Segmentation and Paging

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

MWF 12:00-12:50 VVC 2 215

Today's class

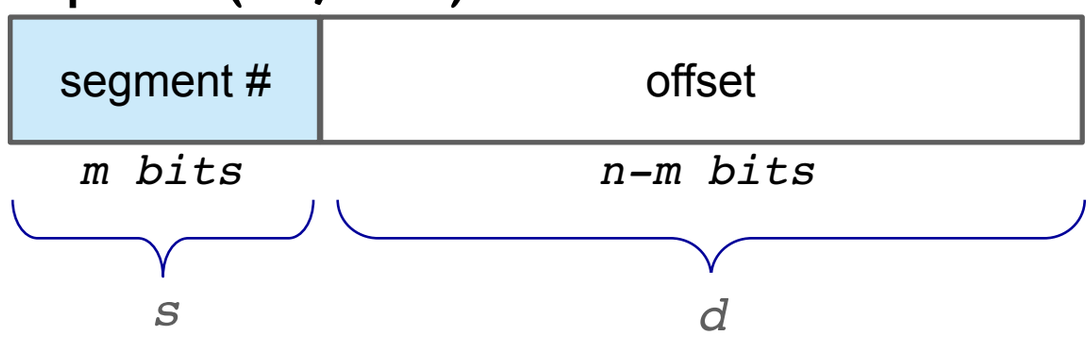
- Memory-management strategies
  - contiguous memory allocation
  - segmentation
  - paging
- Hardware support
  - mapping tables in addition to registers

Segmentation

- process generates contiguous virtual addresses from `0` to $\texttt{MAX}_{\texttt{proc}}$
- OS lays the process down on **arbitrary size segments** and hardware translates virtual addresses to physical addresses in memory
  - using **a segment table** that keeps track of the segment locations in memory

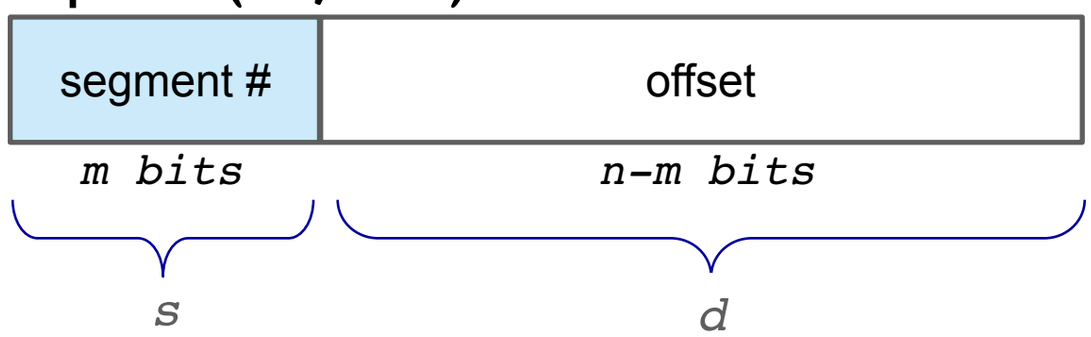
Segmentation

- process generates contiguous virtual addresses from `0` to $\texttt{MAX}_{\texttt{proc}}$
- OS lays the process down on **arbitrary size segments** and hardware translates virtual addresses to physical addresses in memory
  - using **a segment table** that keeps track of the segment locations in memory
- the virtual address is identified by a pair `(s, d)`
  - s is the segment number
  - d is the offset within the segment



Segmentation

- process generates contiguous virtual addresses from `0` to $\texttt{MAX}_{\texttt{proc}}$
- OS lays the process down on **arbitrary size segments** and hardware translates virtual addresses to physical addresses in memory
  - using **a segment table** that keeps track of the segment locations in memory
- the virtual address is identified by a pair `(s, d)`
  - s is the segment number
  - d is the offset within the segment

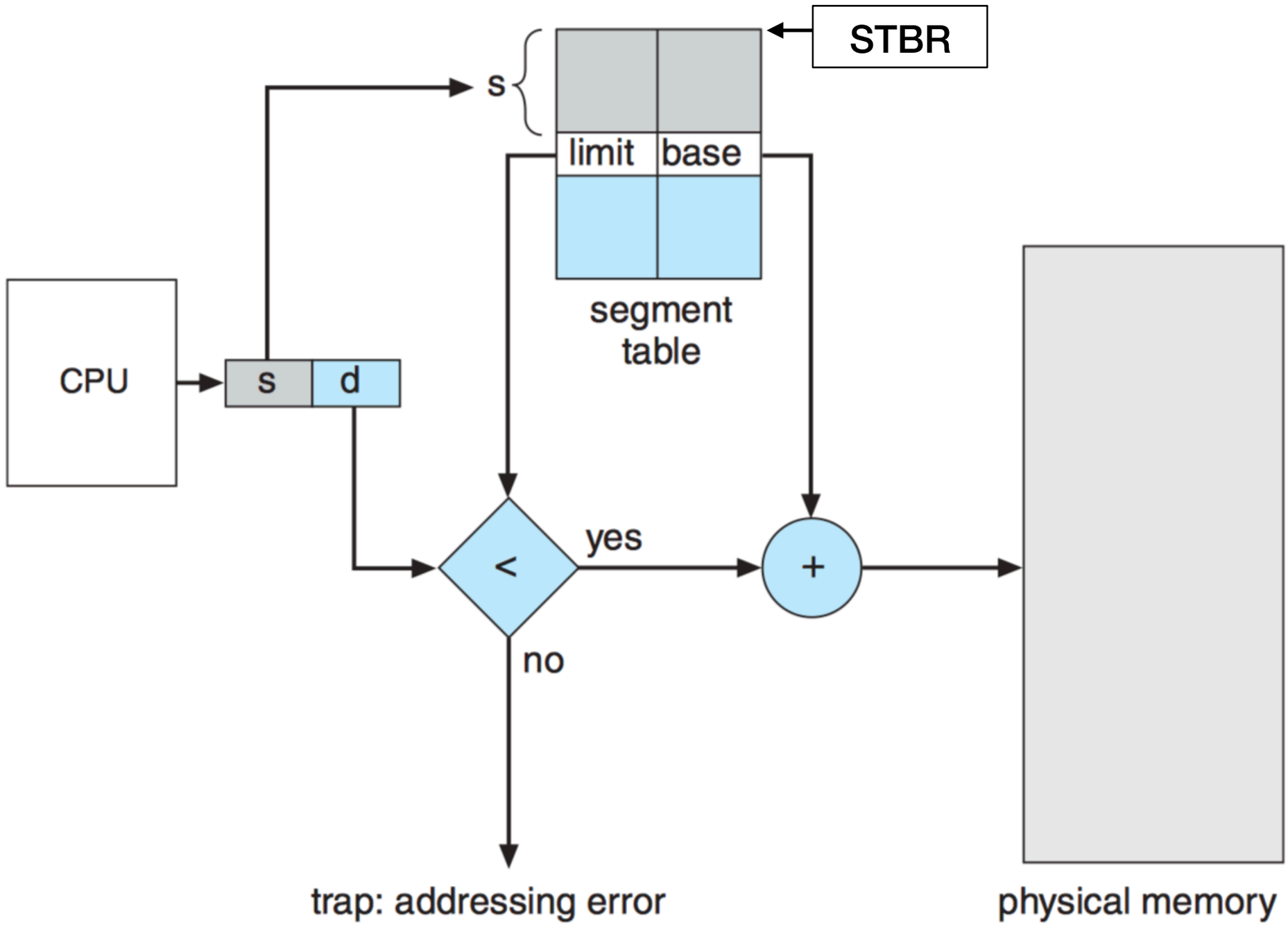


- enable sharing code and data between processes (for example when linking libraries)

How to find addresses when segments are not allocated contiguously in memory?

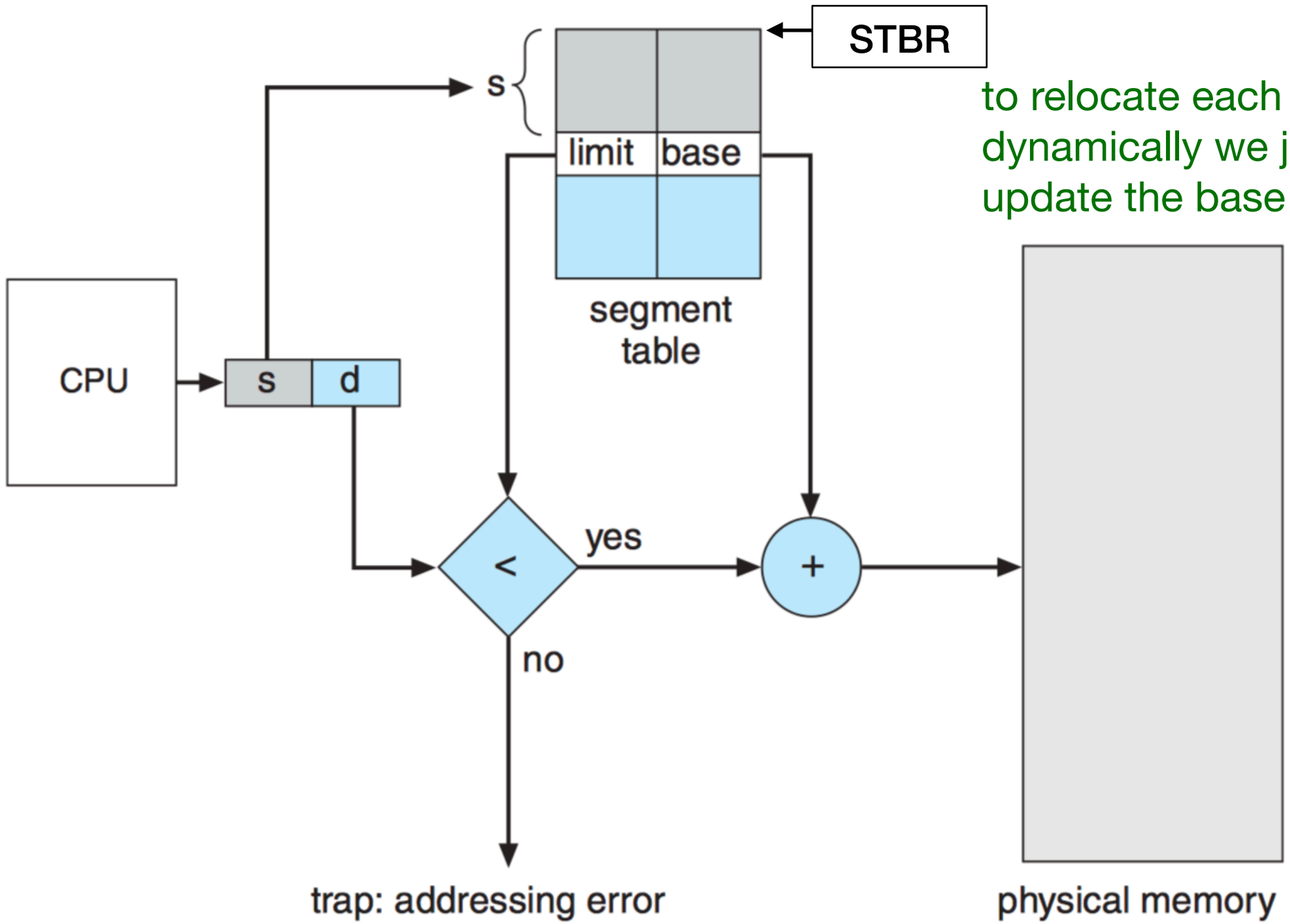
add a segment table containing base & limit register values

How to find addresses when segments are not allocated contiguously in memory?



add a segment table containing base & limit register values

How to find addresses when segments are not allocated contiguously in memory?

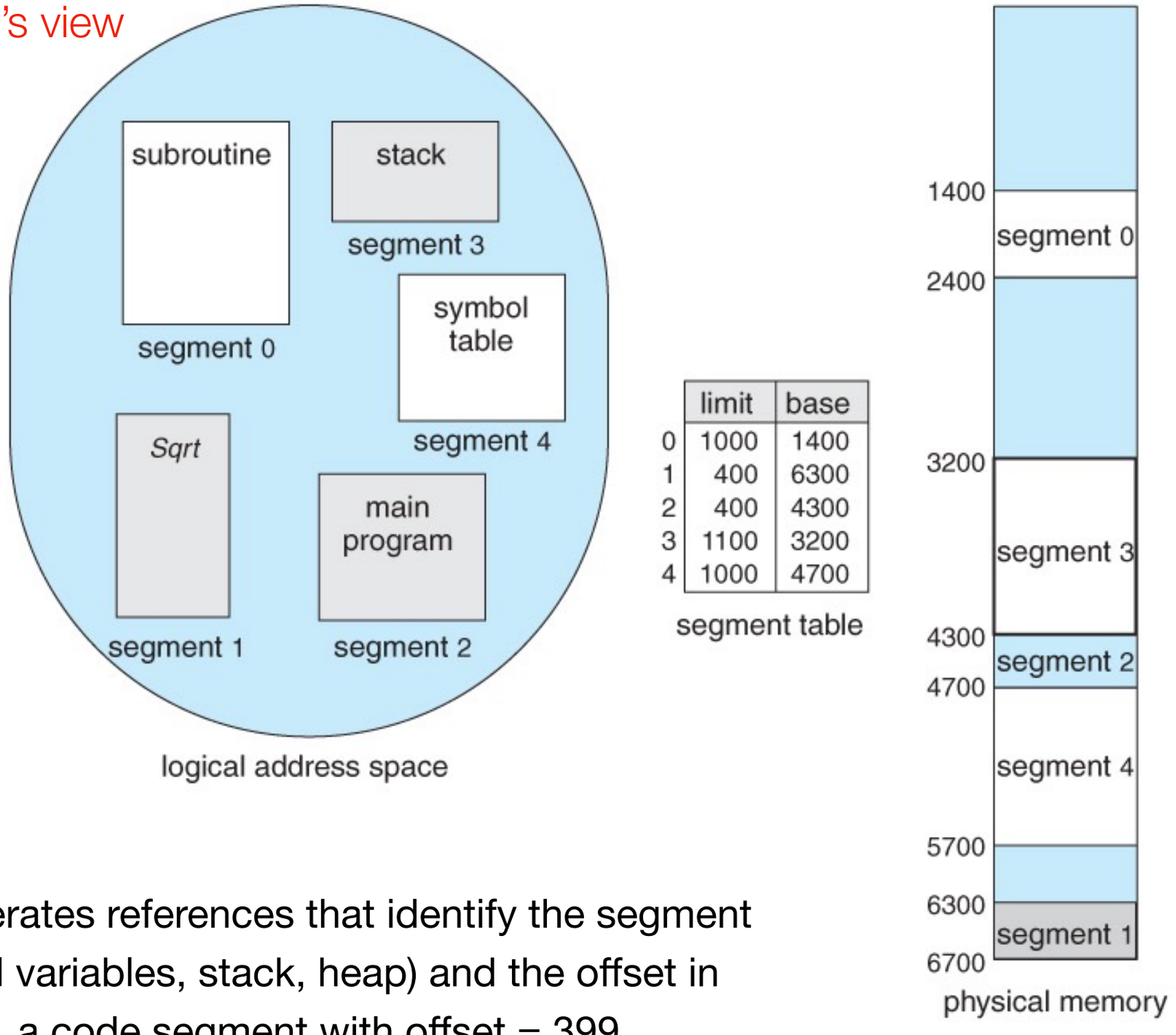


to relocate each segment dynamically we just need to update the base register

add a segment table containing base & limit register values

Hardware support for segmentation

programmer's view



the compiler generates references that identify the segment (e.g., code, global variables, stack, heap) and the offset in the segment, e.g., a code segment with offset = 399

Is segmentation enough?

- segmentation leads to poor memory utilization
  - we might not use much of a large segment, but we must keep the whole thing in memory
  - suffers from external fragmentation
  - allocation/deallocation of arbitrary size segments is complex
- how can we improve memory management? **using fixed-size allocation units**

Motivation for paging

- 90/10 rule
  - processes spend 90% of their time accessing 10% of their space in memory
  - so keep only those parts in memory that are being used

Motivation for paging

- 90/10 rule
  - processes spend 90% of their time accessing 10% of their space in memory
  - so keep only those parts in memory that are being used
- logical memory of a process is contiguous, but pages need not be allocated contiguously in memory

Motivation for paging

- 90/10 rule
  - processes spend 90% of their time accessing 10% of their space in memory
  - so keep only those parts in memory that are being used
- logical memory of a process is contiguous, but pages need not be allocated contiguously in memory

- paging eliminates external fragmentation by dividing memory into **fixed-size** pages
  - the `getpagesize()` system call returns the page size (e.g., 4096 bytes)
  - this command in linux returns the page size as well: `getconf PAGESIZE`

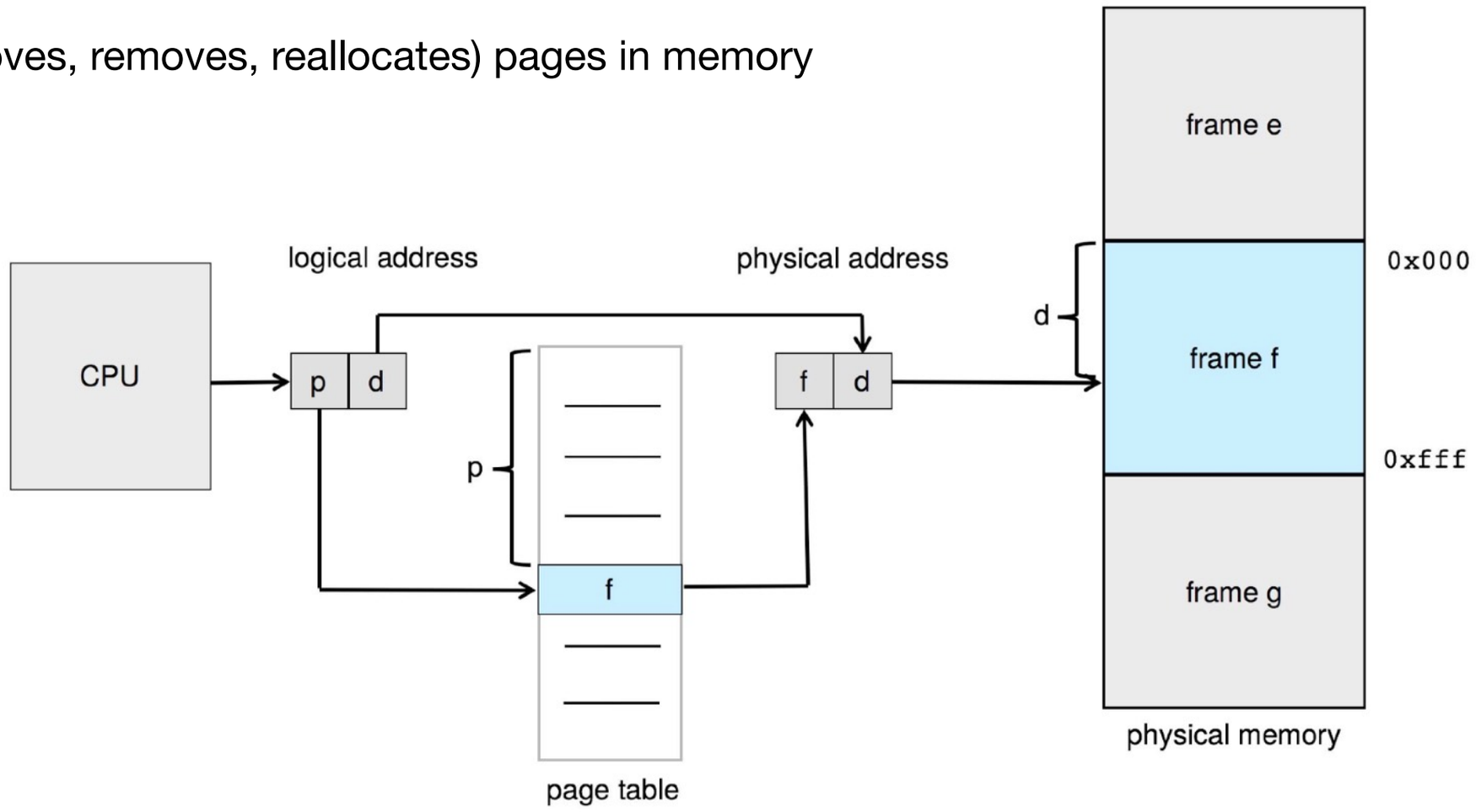
Motivation for paging

- 90/10 rule
  - processes spend 90% of their time accessing 10% of their space in memory
  - so keep only those parts in memory that are being used
- logical memory of a process is contiguous, but pages need not be allocated contiguously in memory

- paging eliminates external fragmentation by dividing memory into **fixed-size** pages
  - the `getpagesize()` system call returns the page size (e.g., 4096 bytes)
  - this command in linux returns the page size as well: `getconf PAGESIZE`
- paging causes internal fragmentation though
  - on average half a page is lost per process

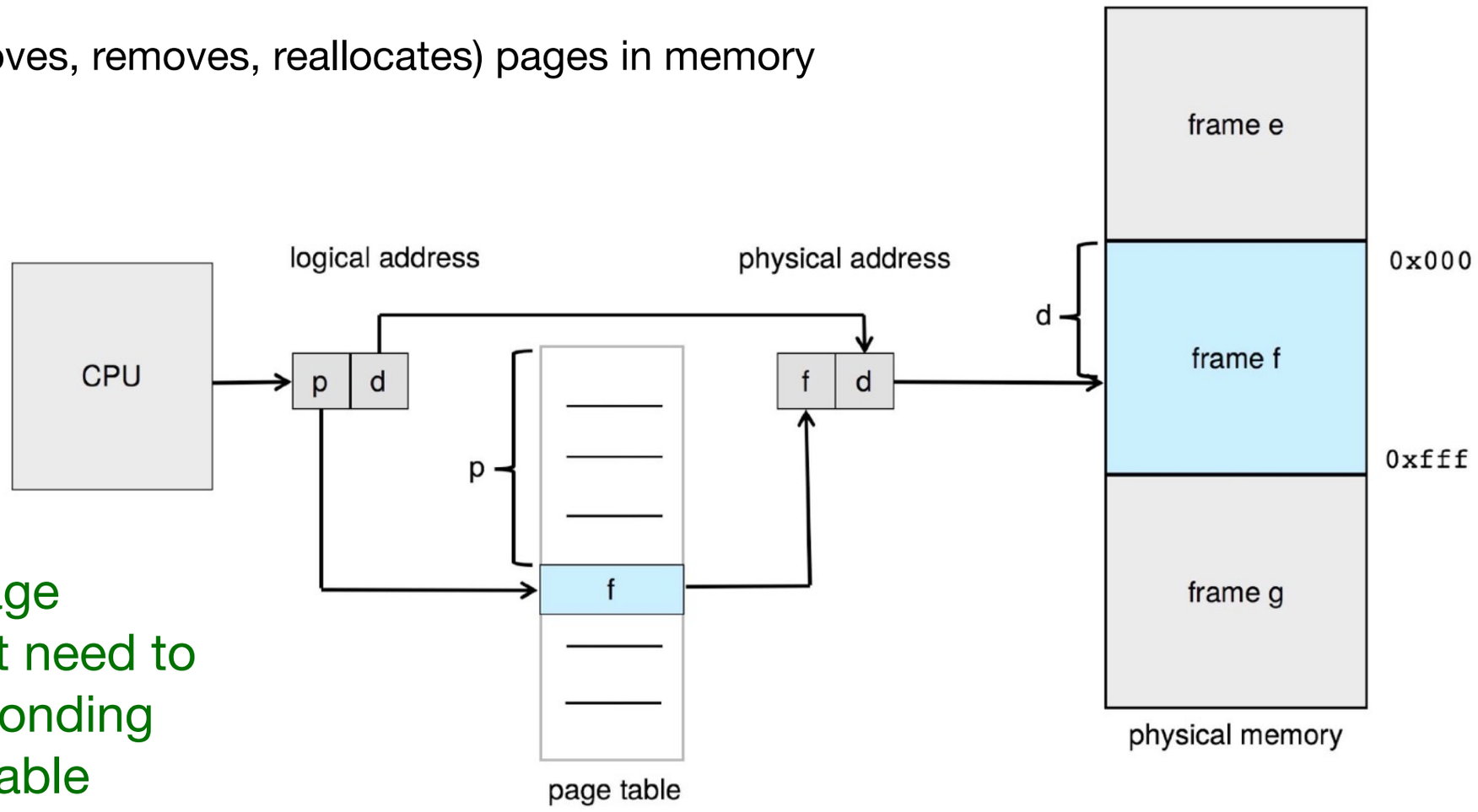
How to find addresses when pages are not allocated contiguously in memory?

- OS divides processes into fixed sized pages
- selectively allocates pages to **page frames** in memory
  - the virtual address contains the page number (p) and the page offset (d)
  - p is used as an index into the **page table**, allowing to extract the frame number (f)
  - the page table keeps track of the page frame in memory in which the page is located
- keeps track of free frames in a **frame table** which has an entry for each physical page frame
- OS manages (moves, removes, reallocates) pages in memory



How to find addresses when pages are not allocated contiguously in memory?

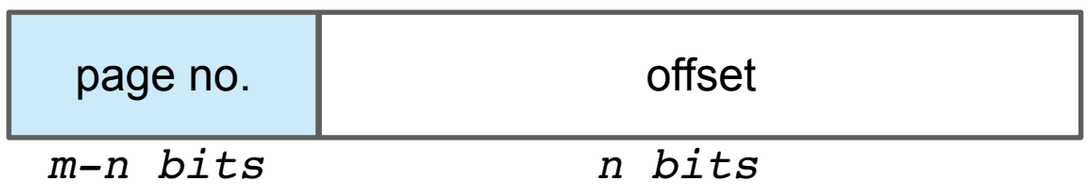
- OS divides processes into fixed sized pages
- selectively allocates pages to **page frames** in memory
  - the virtual address contains the page number (p) and the page offset (d)
  - p is used as an index into the **page table**, allowing to extract the frame number (f)
  - the page table keeps track of the page frame in memory in which the page is located
- keeps track of free frames in a **frame table** which has an entry for each physical page frame
- OS manages (moves, removes, reallocates) pages in memory



to relocate each page dynamically we just need to update the corresponding frame in the page table

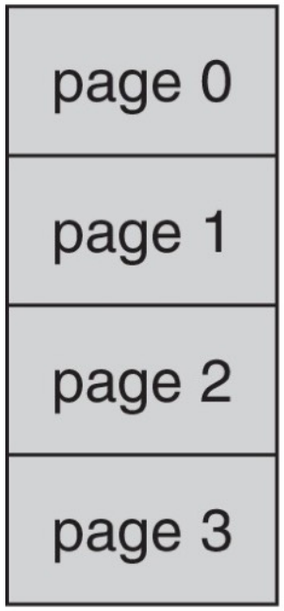
Paging details

- page sizes (and frame sizes) are typically a power of 2 between 512 bytes and 8192 bytes per page
  - makes the translation of virtual addresses into physical addresses easier
- for example, given a memory of size 2^m bytes and a page of size 2^n then
  - the high order m-n bits of a virtual address select the page
  - the low order n bits select the offset in the page



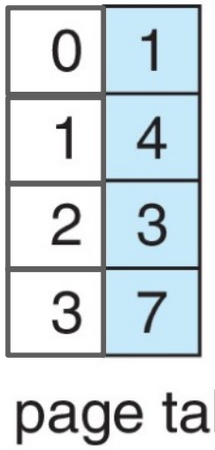
Example

mem. size: 65,536 = 2^{16}
page size: 8,196 = 2^{13}



logical memory

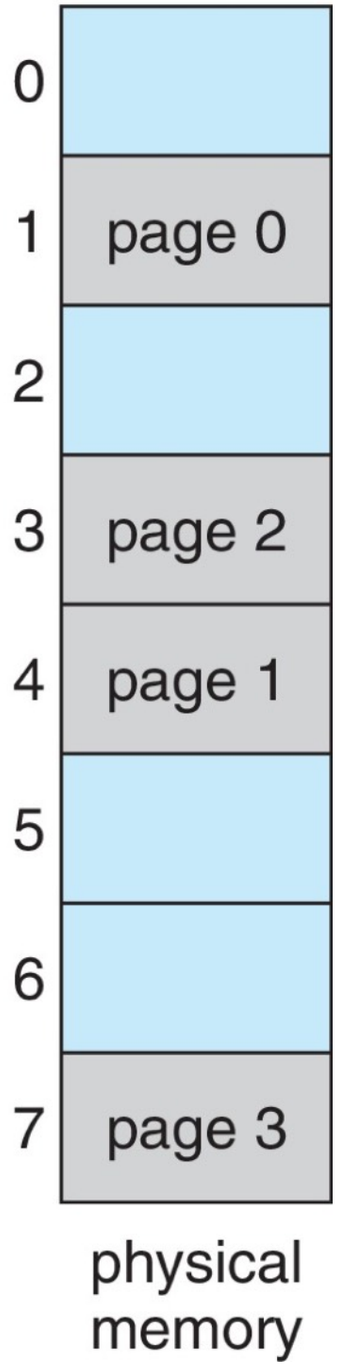
a process is comprised of fixed size pages



page table

(page #, frame #)

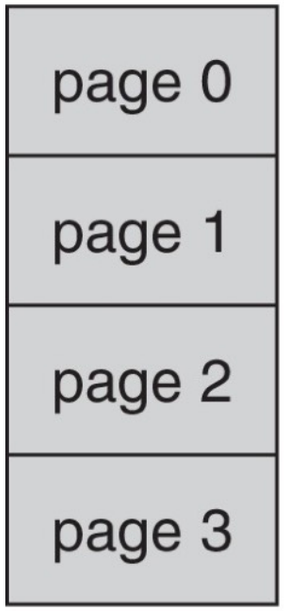
page table is like a set of relocation registers, one for each frame



physical memory

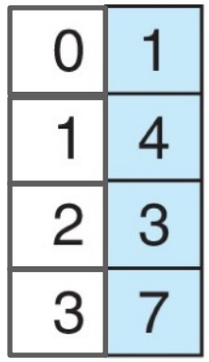
Example

mem. size: 65,536 = 2^{16}
page size: 8,196 = 2^{13}



logical memory

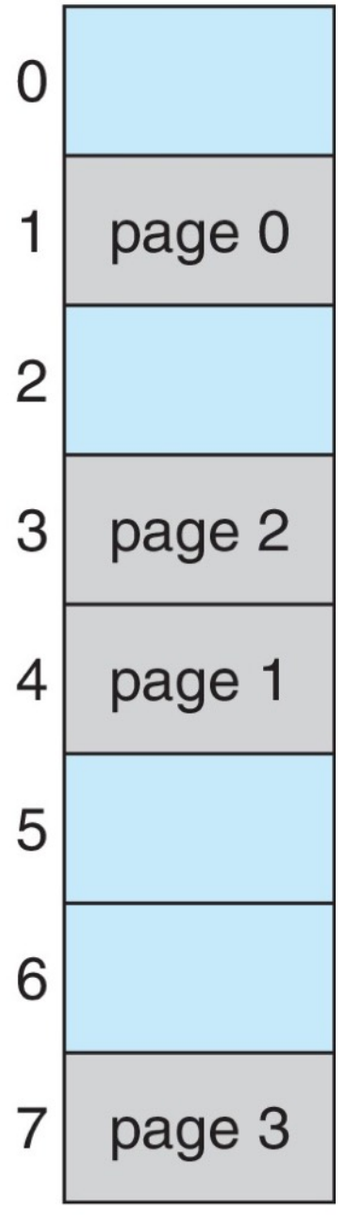
a process is comprised of fixed size pages



page table

(page #, frame #)

page table is like a set of relocation registers, one for each frame



physical memory

How many bits for an address (assuming we can address 1 byte increments)? **16 in this example**

Example of address translation

- how many frames do we have? 8 frames
- what is the page size? 2^{13} bytes
- assuming we can address 1 byte increments, how many bits for an address?
  16bits (3 bits for page table, 13 bits for offset)
- assuming we can address 1 word (4 bytes) increments, how many bits for an address?
  14bits (3 bits for page table, 11 bits for offset)

Calculating internal fragmentation

- recall that
  - worst-case internal fragmentation: 1 frame – 1 byte
  - average-case internal fragmentation: 1/2 frame size
- let the page size be 2,048 bytes and the process size be 72,766 bytes (i.e., 35 pages + 1,086 bytes)
  - internal fragmentation of 2,048 - 1,086 = 962 bytes

Calculating internal fragmentation

- recall that
  - worst-case internal fragmentation: 1 frame – 1 byte
  - average-case internal fragmentation: 1/2 frame size
- let the page size be 2,048 bytes and the process size be 72,766 bytes (i.e., 35 pages + 1,086 bytes)
  - internal fragmentation of 2,048 - 1,086 = 962 bytes
- can we say small frame sizes are desirable?
  - No, each page table entry takes memory to track

Where to store the page table?

- put it in memory?
  - page-table base register (PTBR) points to the page table
  - page-table length register (PTLR) indicates size of the page table
  - every data/instruction access requires two memory accesses: one for the page table and one for the data/instruction

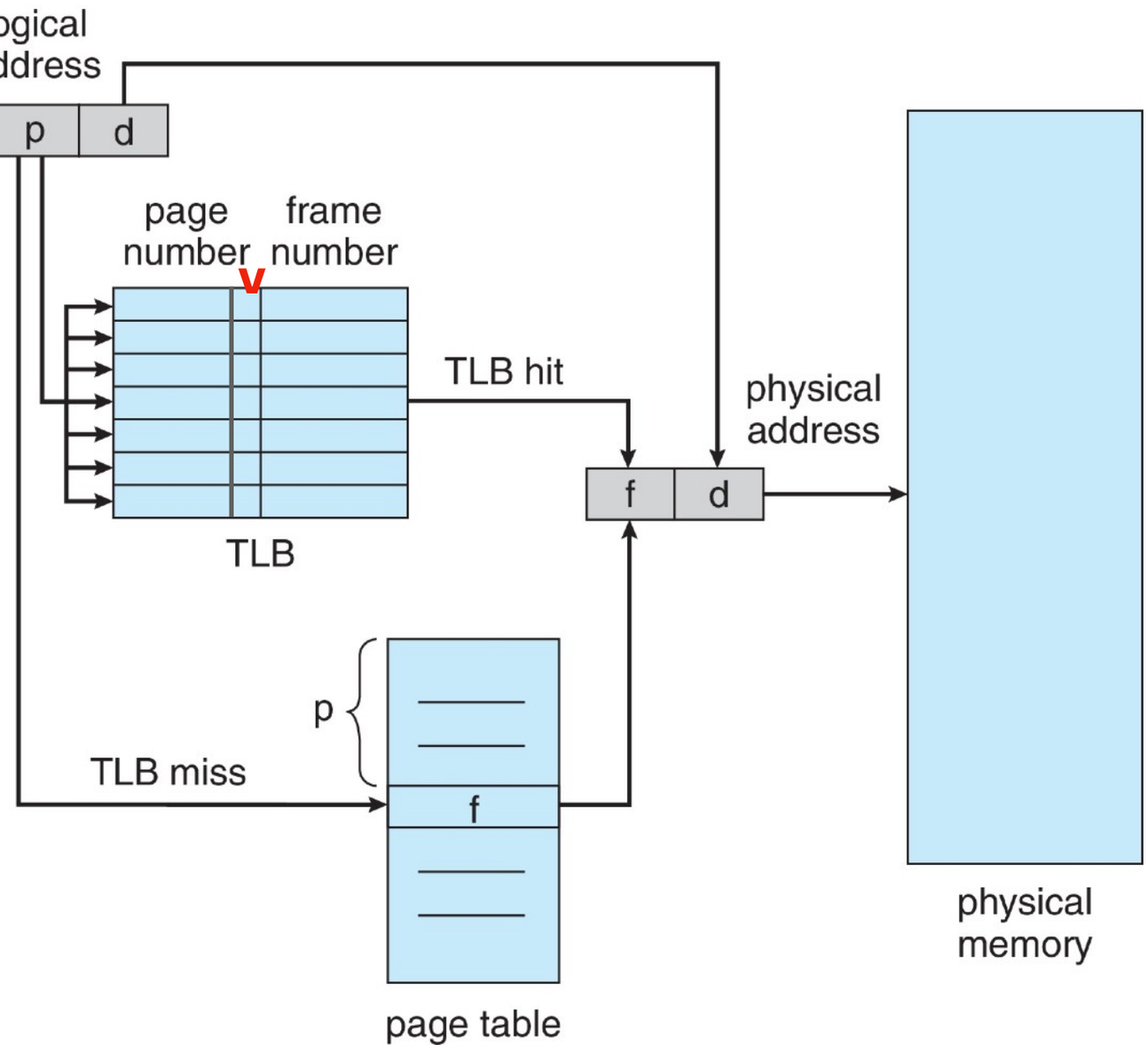
- **translation look-aside buffer** (TLB) is a fast-lookup hardware cache in MMU (fully associative memory) that stores page numbers (key) and the frame (value) in which they are stored
  - if memory accesses have **locality**, address translation has locality too
  - typical TLB sizes range from 64 to 1,024 entries (TLB is small)
  - TLB is typically on chip, access time of a few nanoseconds instead of several hundred of nanoseconds for main memory

Paging with TLB

on a TLB miss, value is loaded into the TLB for faster access next time

- replacement policies must be considered

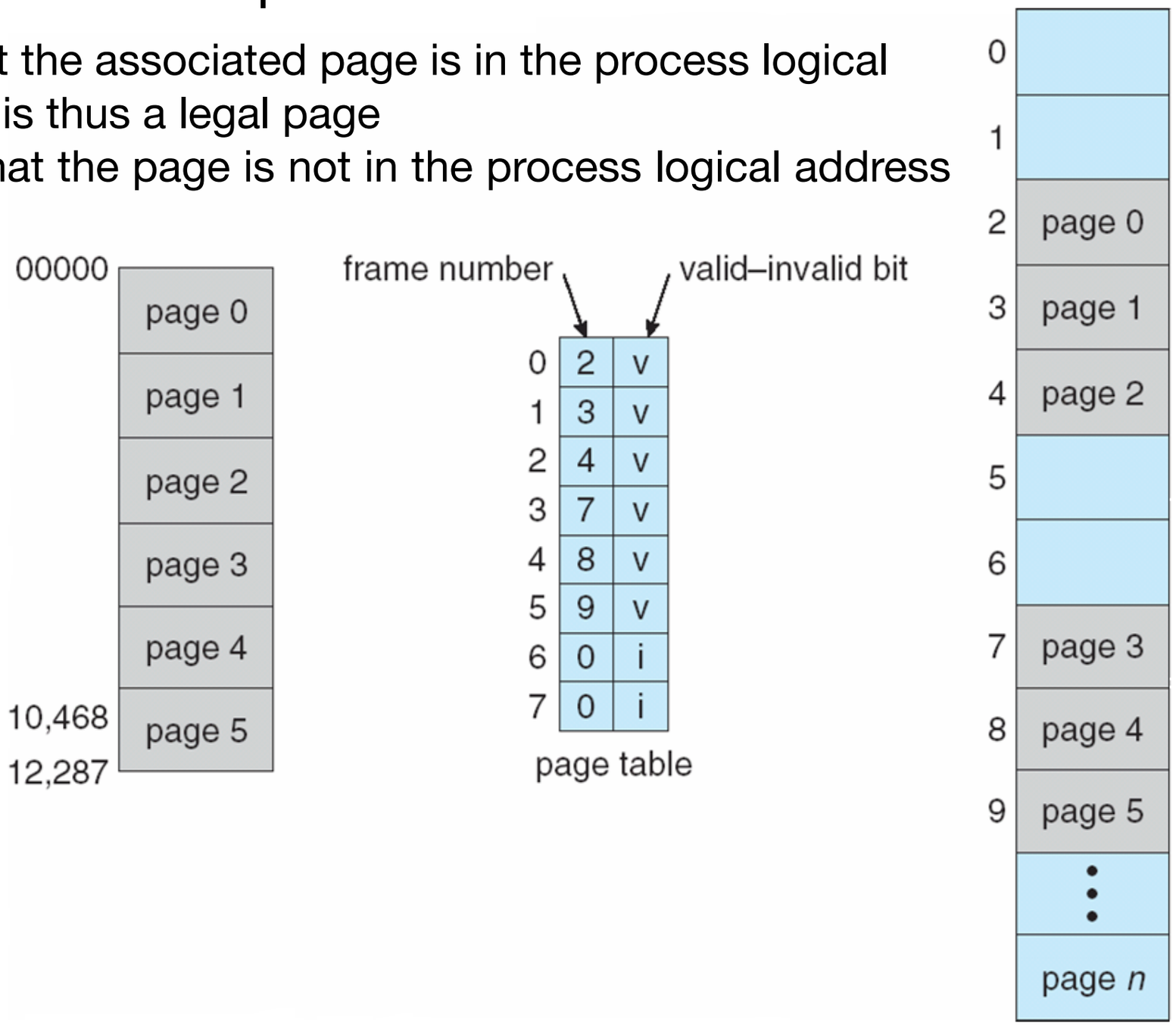
v: valid/invalid bit used for protection



Valid/invalid bit in TLB

v: valid/invalid bit used for protection

- “valid” indicates that the associated page is in the process logical address space, and is thus a legal page
- “invalid” indicates that the page is not in the process logical address space



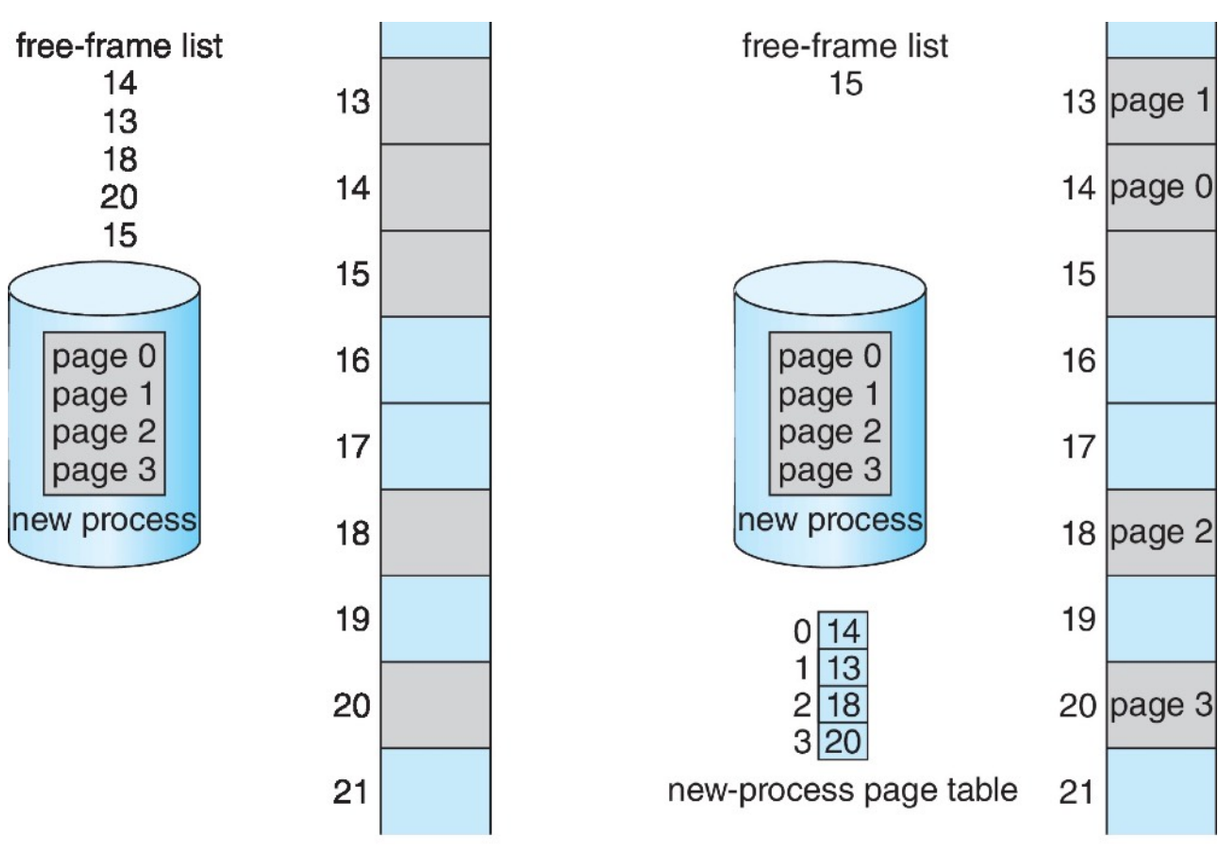
Effective memory access cost with TLB

- the effective access time (EAT) if the page table is in memory?
  EAT $= 2 \times C_{ma}$
- the effective access time with a TLB?
  EAT $= \rho_{hit}(C_{ma} + C_{TLB}) + (1 - \rho_{hit})(2 \times C_{ma} + C_{TLB})$
  - where ρ_{hit} is the hit ratio (i.e., percentage of times that a page number is found in the TLB)
- a large TLB improves hit ratio, decreases average memory cost
- example: let $C_{ma} = 10$ns, $C_{TLB} = 0$ns
  - if ρ_{hit}=80% then EAT=12ns (20% slowdown in access time)
  - if ρ_{hit}=99% then EAT=10.1ns (1% slowdown in access time)

How to allocate memory when starting a process?

suppose the process needs k pages

- if k page frames are free, then allocate these frames to pages; otherwise, free frames that are no longer needed and allocate them to the process
- the OS puts each page in a frame and then puts the frame number in the corresponding entry in the page table
- OS marks all TLB entries as invalid (flushes the TLB) and starts the process
- as process executes, OS loads TLB entries as each page is accessed, replacing an existing entry if the TLB is full



before allocation **after allocation**

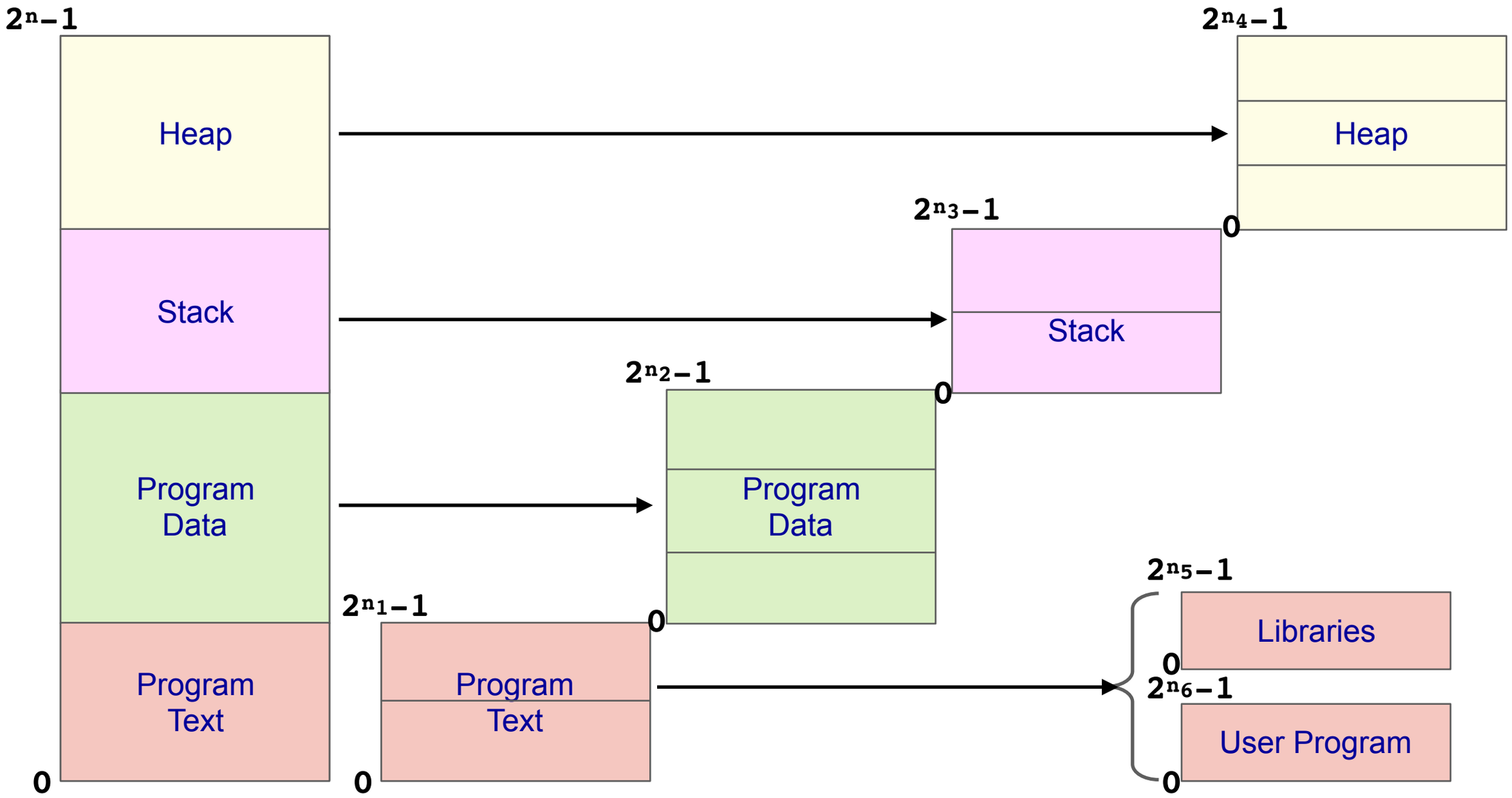
What happens on a context switch?

- OS saves the PTBR value to the PCB
- OS saves the process's page table in its PCB
  (i.e., copying the TLB to the PCB)
- OS marks all TLB entries as invalid (i.e., flushes the TLB)
- OS restores the PTBR value of the new process
- OS restores the TLB if it was saved

Sharing

- increases the level of multiprogramming by sharing code and data between different processes
  - shared code must be reentrant, that means the processes that are using it cannot change it
  - OS keeps track of available reentrant code in memory and reuses them if a new process requests the same program
- allows running many processes given a limited amount of memory
- sharable units: segments or pages
- paging allows sharing of memory across processes, since memory used by a process no longer needs to be contiguous

How to implement sharing?



Segmentation versus paging

- paging is a big improvement over segmentation
  - eliminates the problem of external fragmentation and therefore the need for compaction
  - enables processes to run when they are only partially loaded in main memory

Segmentation versus paging

- paging is a big improvement over segmentation
  - eliminates the problem of external fragmentation and therefore the need for compaction
  - enables processes to run when they are only partially loaded in main memory
- however, paging has its cost:
  - causes internal fragmentation
  - translating from a virtual address to a physical address is more time-consuming
    - segment table tends to be much smaller than page table
  - paging requires hardware support in the form of a TLB to be efficient enough
  - paging requires more complex OS to maintain the page table