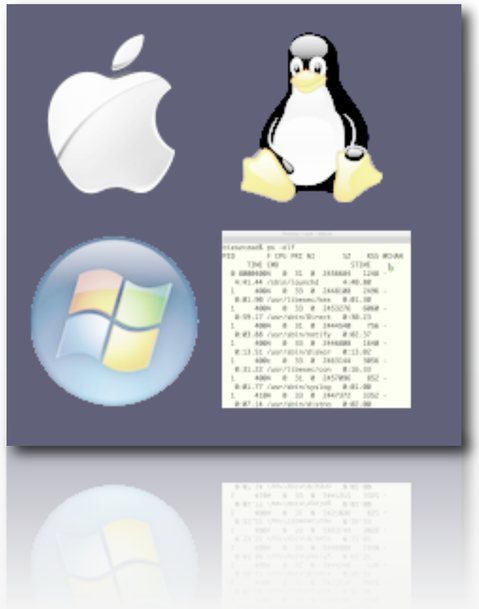


Lecture 9: Main Memory

COMP362 Operating Systems
Prof. AJ Bieszczad

Outline

- Background
- Address Binding
- Logical and Physical Address Spaces
- Contiguous Memory Allocation
- Memory Fragmentation
- Non-Contiguous Memory Allocation Based on Paging
- Page Table Implementations
- Non-Contiguous Memory Allocation Based on Segments
- Hybrid Memory Allocation

Background

- Programs are stored on external media (e.g., disks)
  - access to disk is slow, so this is not an option for a running process
  - program must be brought (from disk) into memory and placed within the space allocated to the process for it to be run
    - the location changes over time
  - buffers and caches are used to speed up the I/O
- Main memory and registers are the only storage that the CPU can access directly
  - register access is fast; one CPU clock (or less)
  - accessing main memory is much slower as it takes more clock cycles
    - still orders of magnitude faster than accessing a disk
- OS needs to protect memory to ensure correct operation
  - ➠protection of OS memory from user programs
  - ➠separation between user programs

An additional fast cashed memory is put between memory and CPU registers, so the often-used data can be accessed quickly from the cache rather then from memory.

Cashing is cascaded with the fastest and smallest L1 – level one, slower and larger L2 – level two, followed by L3 – level three and potentially more caches that are increasing in size but decreasing in speed to make them cost-effective.

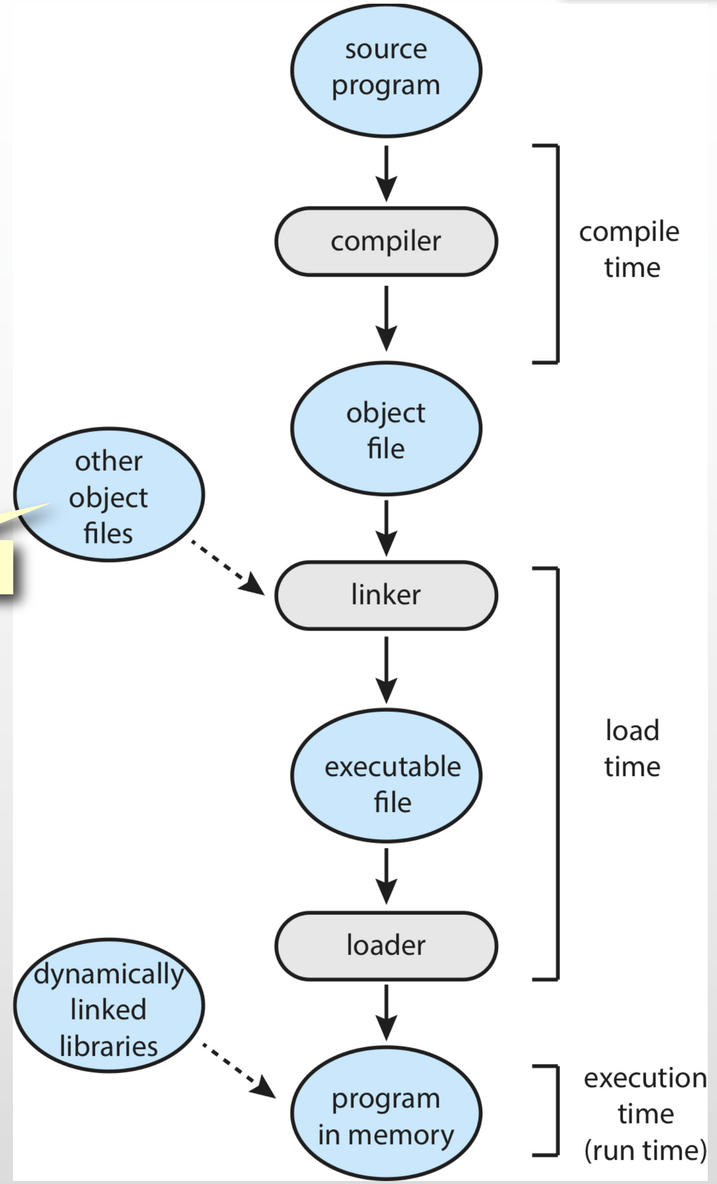
L-caches can be shared or not on multi-core CPUs. Usually, L1 and L2 and not shared, while L3 is shared.

L-caches can be universal for both instructions and data, or they can be specialized either for instructions or for data.

Binding of Instructions and Data to Memory

- Address (of both, instructions and data) from the program on the disk must be translated into an address in the memory
- The **binding** can happen at three different stages
  - **Compile time**:
    - if memory location is known a priori, then the compiler can generate absolute code
      - code requires recompilation if the starting location changes
  - **Load time**:
    - if memory location is not known at compile time, code must be relocatable
      - binding is done when loading program
      - program must stay in memory
  - **Execution time**:
    - binding delayed until run time
      - the process must be able to be moved during its execution from one memory location to another.

Hardware support for address maps required; e.g., base and limit registers



Dynamic Linking

- Linking postponed until execution time
- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system needed to check if routine is in processes' memory address
- Dynamic linking is particularly useful in **dynamic libraries**
  - also known as **shared libraries**
    - `.dll` in Windows
    - `.so` in Unix ("**s**hard **o**bject")
      - always start with "lib"; for example, `libfoo.so`
      - `.dylib` on Mac OS X

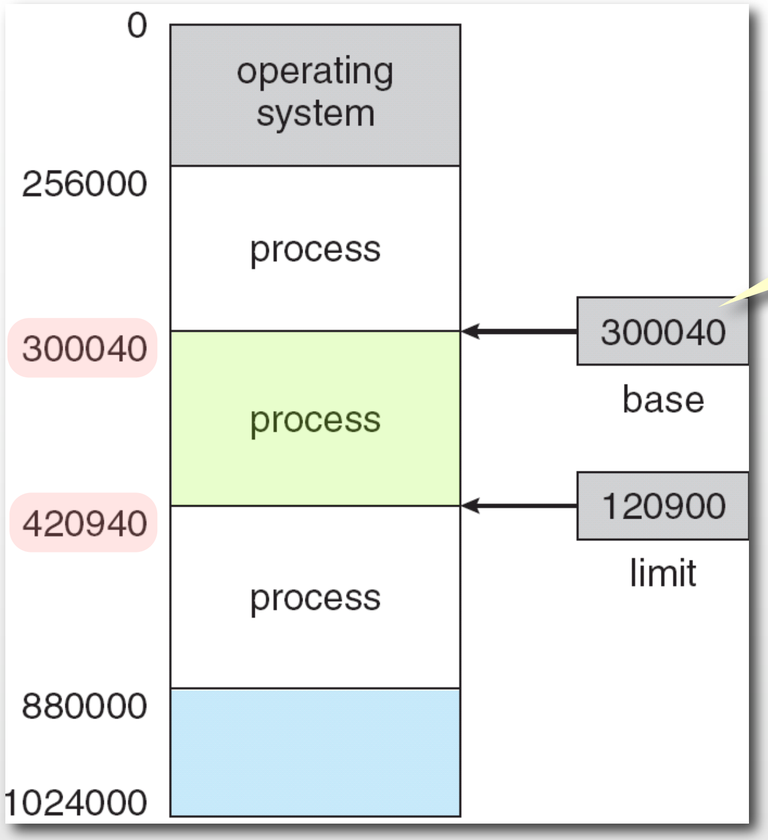
since they are shared between a number of programs using them

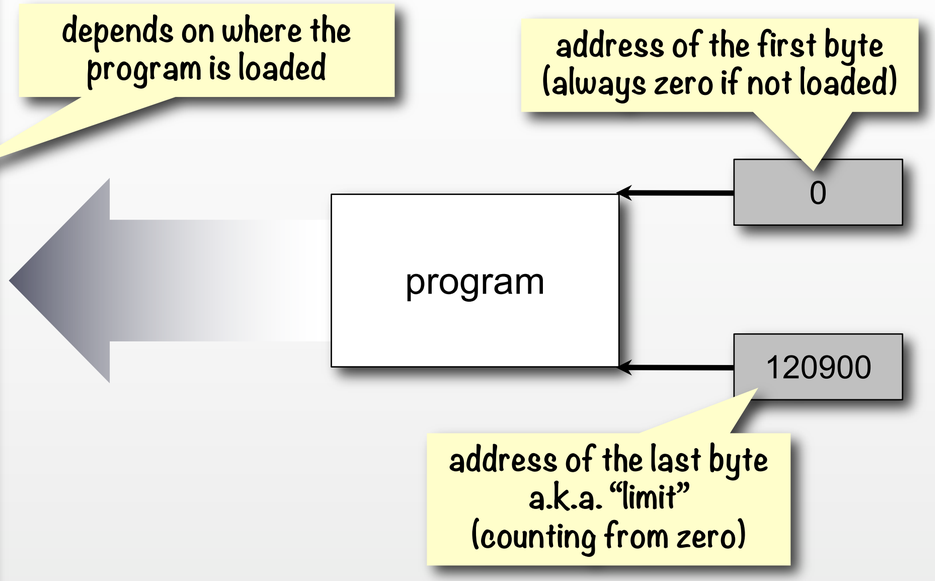
- In contrast to a dynamic library, a **static library** becomes part of the program after linking
  - it is cloned to every program that uses it
  - static libraries use `.a` extension (for **a**rchive; e.g., `libfoo.a`)
- In Linux, the use of standard C library is optimized
  - usually the dynamic version `libc.so` is used rather than static `libc.a`
    - this behavior can be changed by using `–static` flag of the gcc compiler

- You will learn how to create libraries, both static and dynamic, in the lab.

Base and Limit

- A pair of values defines the logical address space of a process
  - the **base** is the beginning of the space for the process
  - the **limit** is the scope of the space; i.e.,
    - base ≤ any address that process uses < base + limit

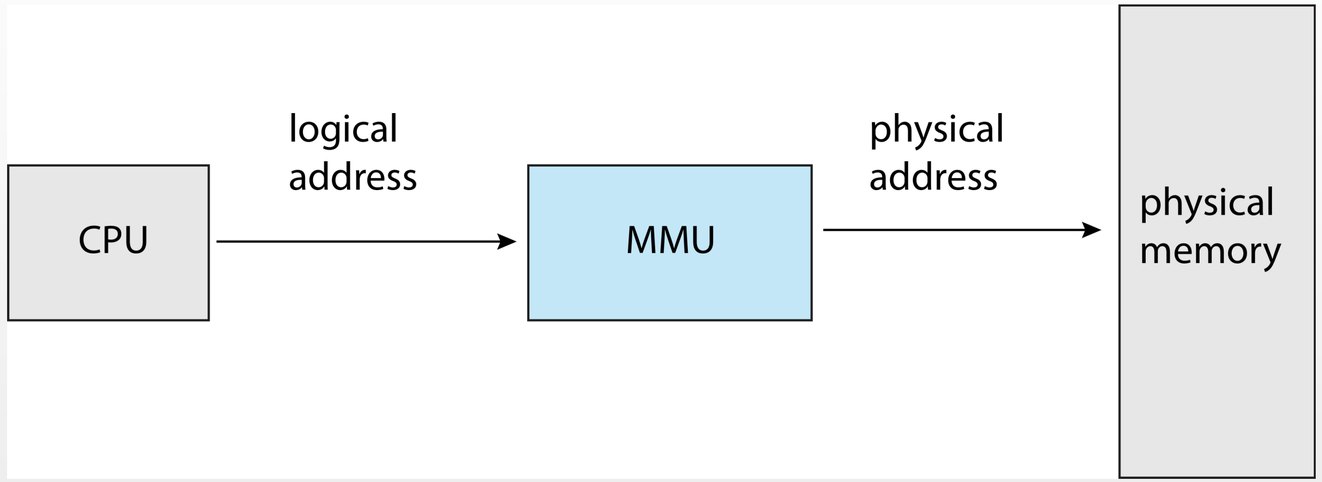




relocatable code can be loaded wherever there is room in the memory

Logical vs. Physical Address Space

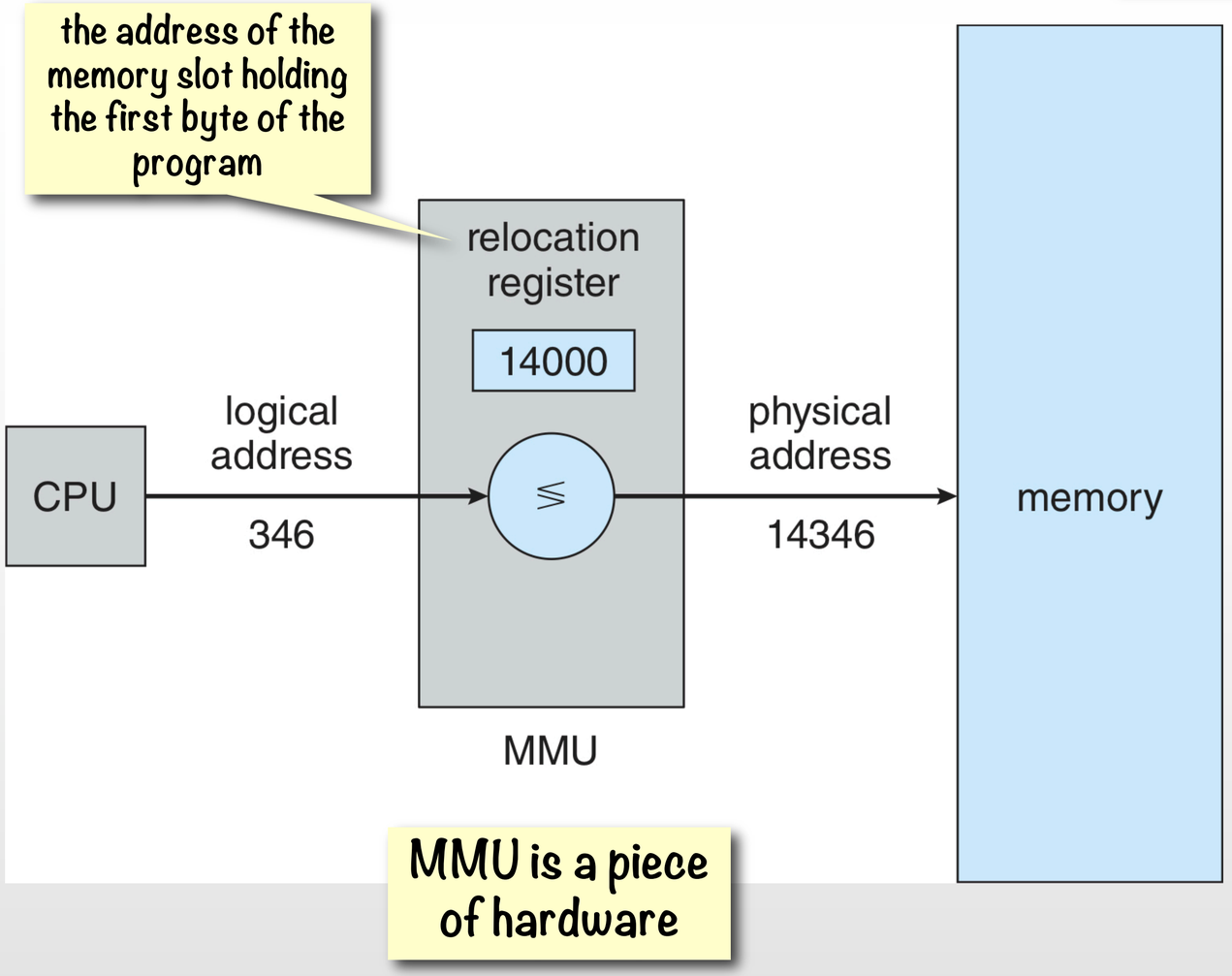
- The concept of a **logical (a.k.a. virtual) address space** that is bound to a separate **physical address space** is central to proper memory management
  - **Logical (a.k.a. virtual) address**
    - generated by the CPU; also referred to as virtual address
  - **Physical address**
    - address seen by the memory unit
    - needed to actually fetch or push data from and to memory



- Logical and physical addresses are the same in compile-time and load-time address-binding schemes
- Logical and physical addresses differ in execution-time address-binding scheme

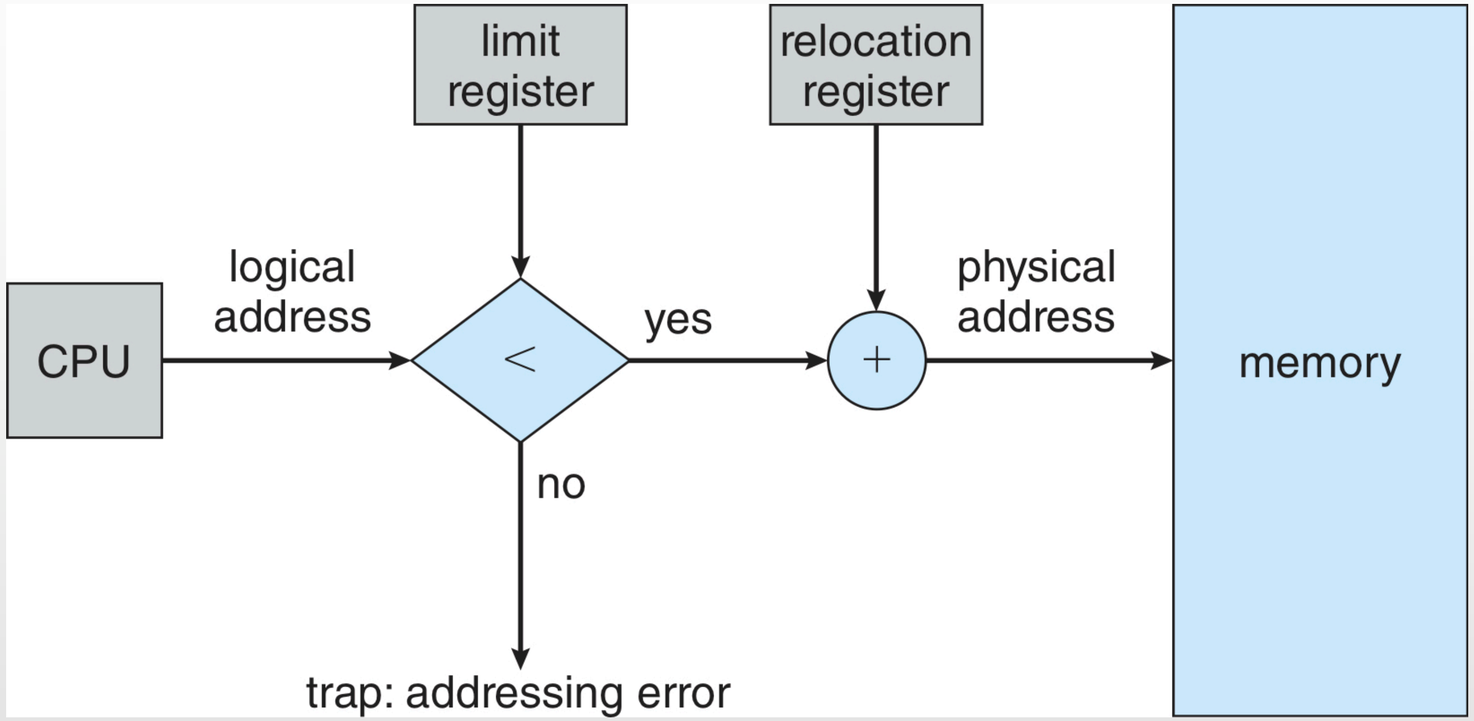
Memory-Management Unit (MMU)

- Hardware device that maps logical addresses to physical addresses
- In MMU scheme, the value in the **relocation register (a.k.a. base register)** is added to every address generated by a user process
- The user program deals with logical addresses; it never sees the physical addresses



Contiguous Allocation of Memory to a Program

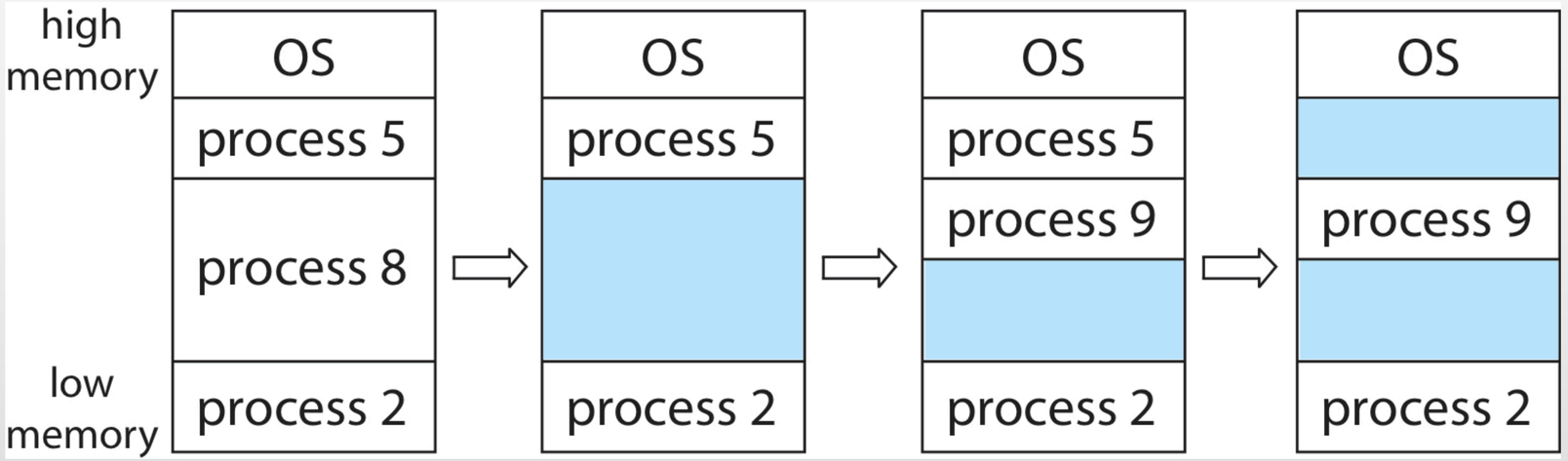
- Programs loaded into memory in their entirety
  - in run time binding they can be moved (“reloaded”)
  - the main memory usually divided into two partitions:
    - resident operating system, usually held in low memory close to interrupt vector
    - user processes then held in high memory
- **Relocation registers** provide protection of user processes from each other, and from compromising operating-system code and data
  - the **base register** contains the value of the smallest physical address of a program
    - the logical program address space is “relocated” to its physical counterpart
      - i.e., each logical address is modified by adding the base value to it
  - the **limit register** contains the range of logical addresses of a program
    - each logical address for a given program must be less than the limit register



- MMU translates all addresses transparently

Issues with Contiguous Allocation

- Multiple-partition allocation leads to empty spaces
  - **memory holes**
    - blocks of available memory
  - holes of various size are scattered throughout memory
    - since programs are loaded and unloaded (e.g., when they terminate)
  - when a process arrives, it must be allocated memory from a hole large enough to accommodate it
  - operating system maintains information about:
    - allocated partitions
      - e.g., the ownership
    - **free partitions** (blue "holes" in the diagram)



Algorithms for Contiguous Allocation

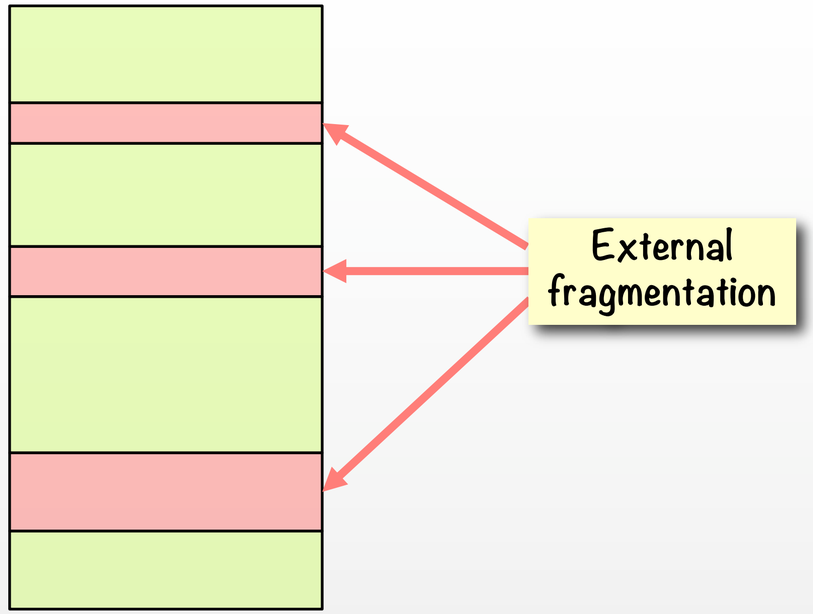
- Allocation approaches
  - **First-fit**
    - Allocate the first hole that is big enough
  - **Best-fit**
    - Allocate the smallest hole that is big enough; must search entire list, unless ordered by size
      - produces the smallest leftover hole
  - **Worst-fit**
    - Allocate the largest hole; must also search entire list
      - produces the largest leftover hole

- Experiments showed that first-fit and best-fit better than worst-fit in terms of speed and storage utilization
  - however, the issues with memory holes are affect each method

External Fragmentation

- The waste of space pertinent to all fitting algorithms is called **fragmentation**

- **External Fragmentation**
  - the total aggregate of memory space exists to satisfy a request, but it is not contiguous, so it's increasingly difficult to fit new programs in the remaining holes
    - e.g., the aggregate of the red regions in the diagram can hold a much larger program than any of the individual holes



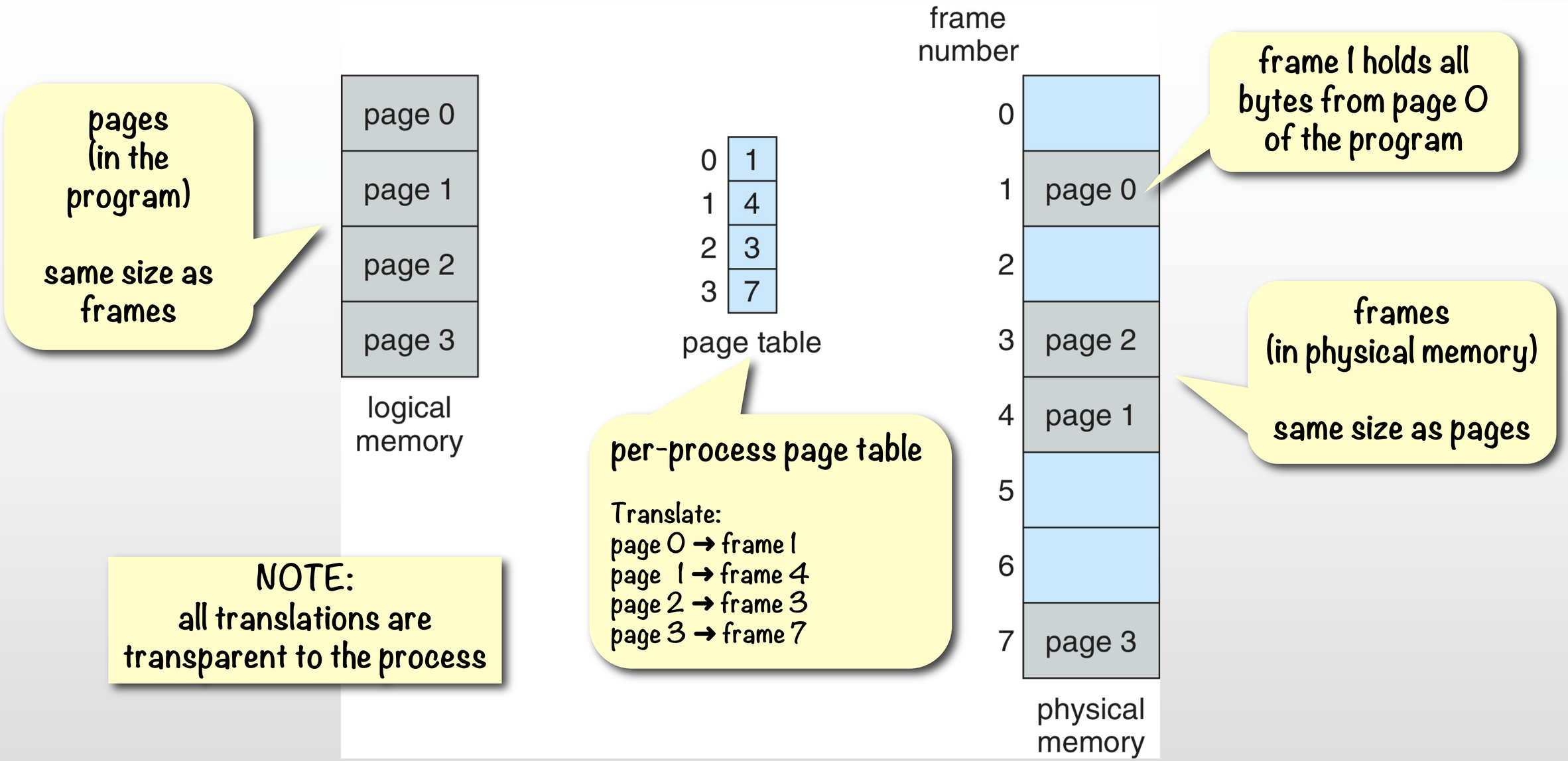
- Reduce external fragmentation by **compaction**
  - shuffle memory contents to place all free memory together in one large block
  - compaction is possible only if relocation is dynamic, and is done at execution time
  - I/O problem: what do we do with I/O operations that read/write from/to memory?
    - e.g., direct access to video memory in some games
    - latch job in memory while it is involved in I/O
    - do I/O only into OS buffers

Non-Contiguous Memory Allocation: Paging

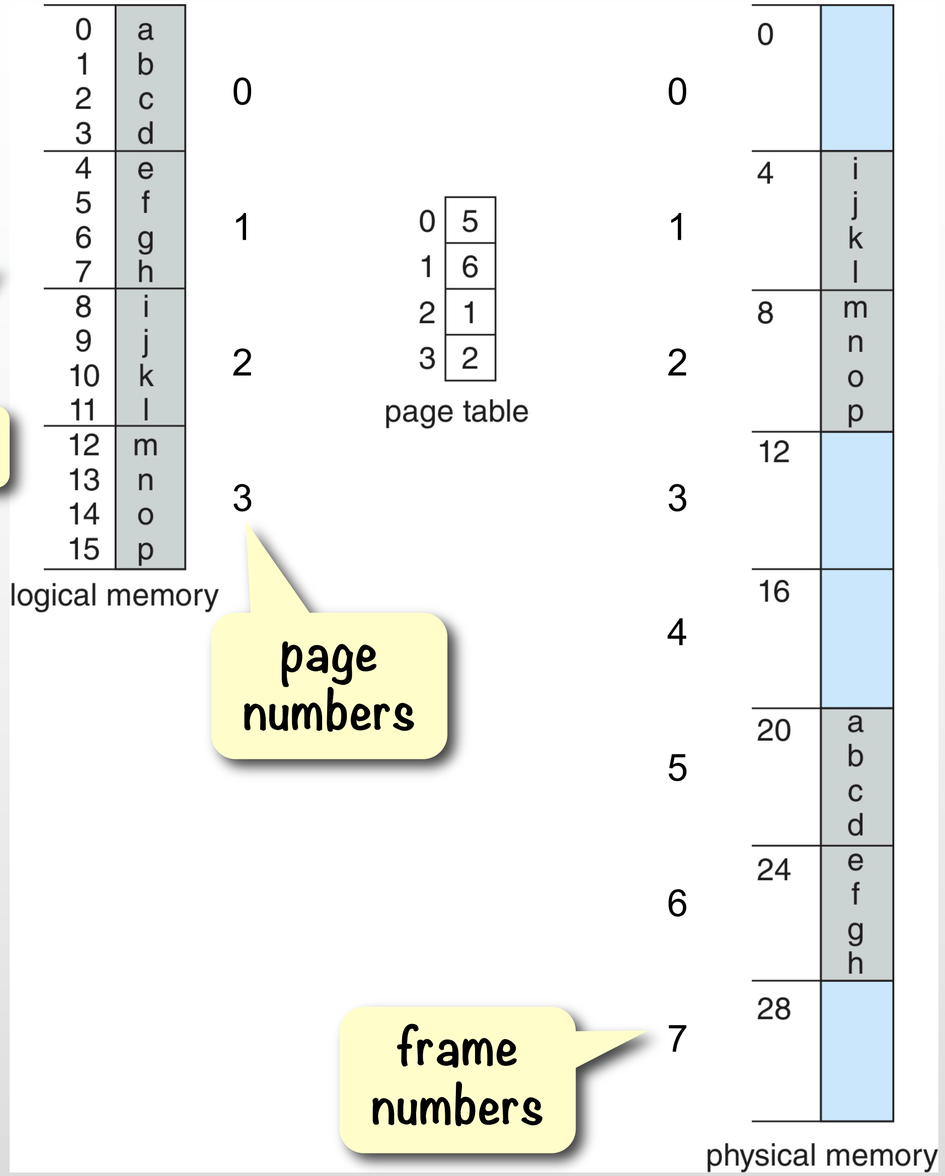
- Transferring the image of the whole process each time it is activated is inefficient and leads to severe fragmentation
  - Note that only some parts of the image might be needed at any given time!

- Divide physical memory into fixed-sized blocks called **frames**
  - size is power of 2, usually between 512 bytes and 8,192 bytes
    - because splitting a binary address into two parts is easy
- Divide logical memory (program) into blocks of same size called **pages**
  - rather than keeping all program in one large chunk of physical memory, keep the program pages in memory frames wherever the frames can be found
- Keep track of all **free frames**
- To run a program of size *n* pages, we need to find *n* free frames and load program pages in these frames
  - the frames don't need to be contiguous
- Set up a **page table** to translate logical to physical addresses

Frame Allocation Example

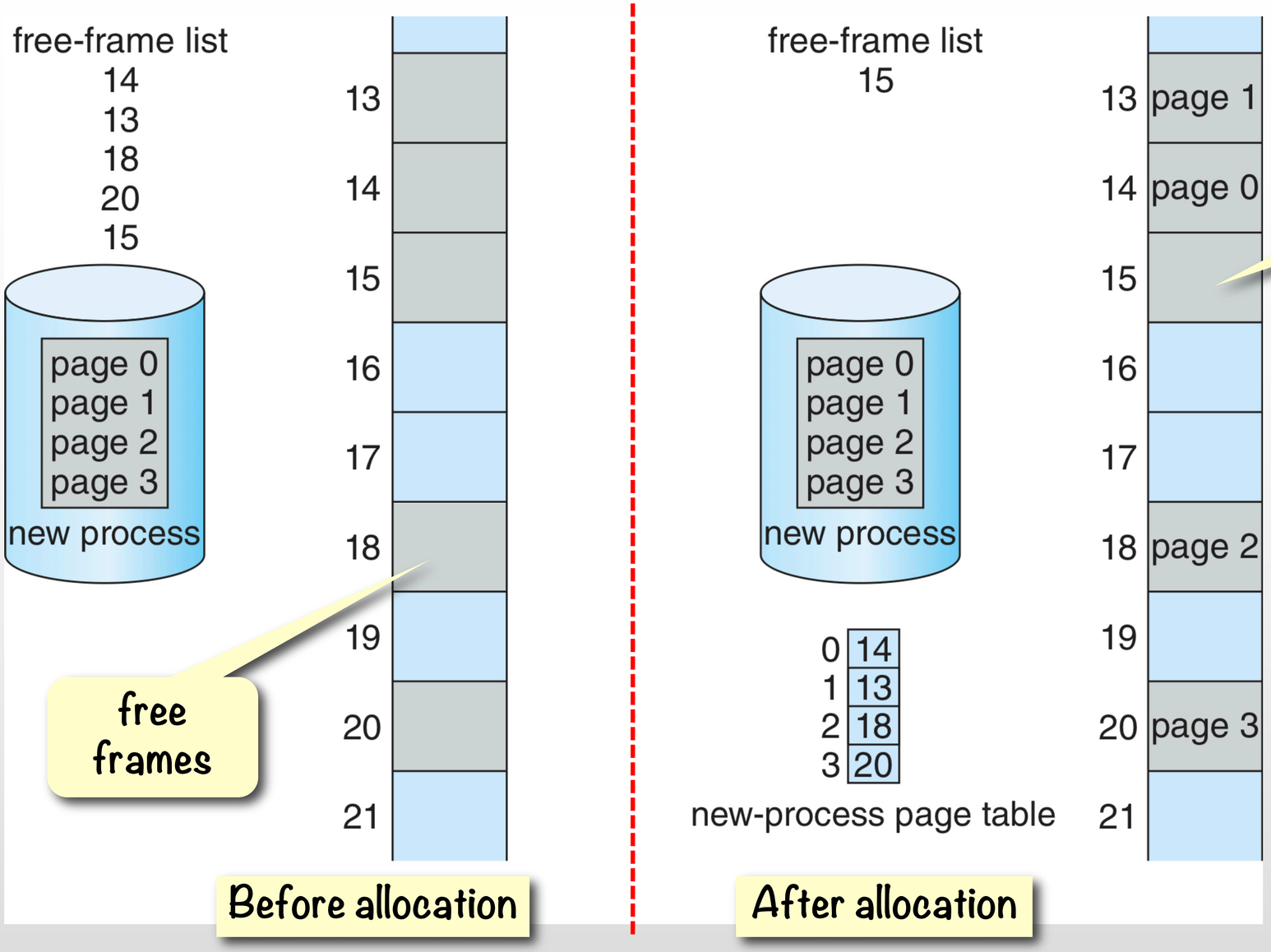


Frame Allocation Example



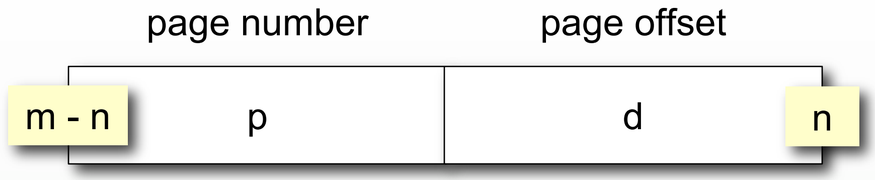
"process": 16 letters in 4 pages

Allocating Free Frames

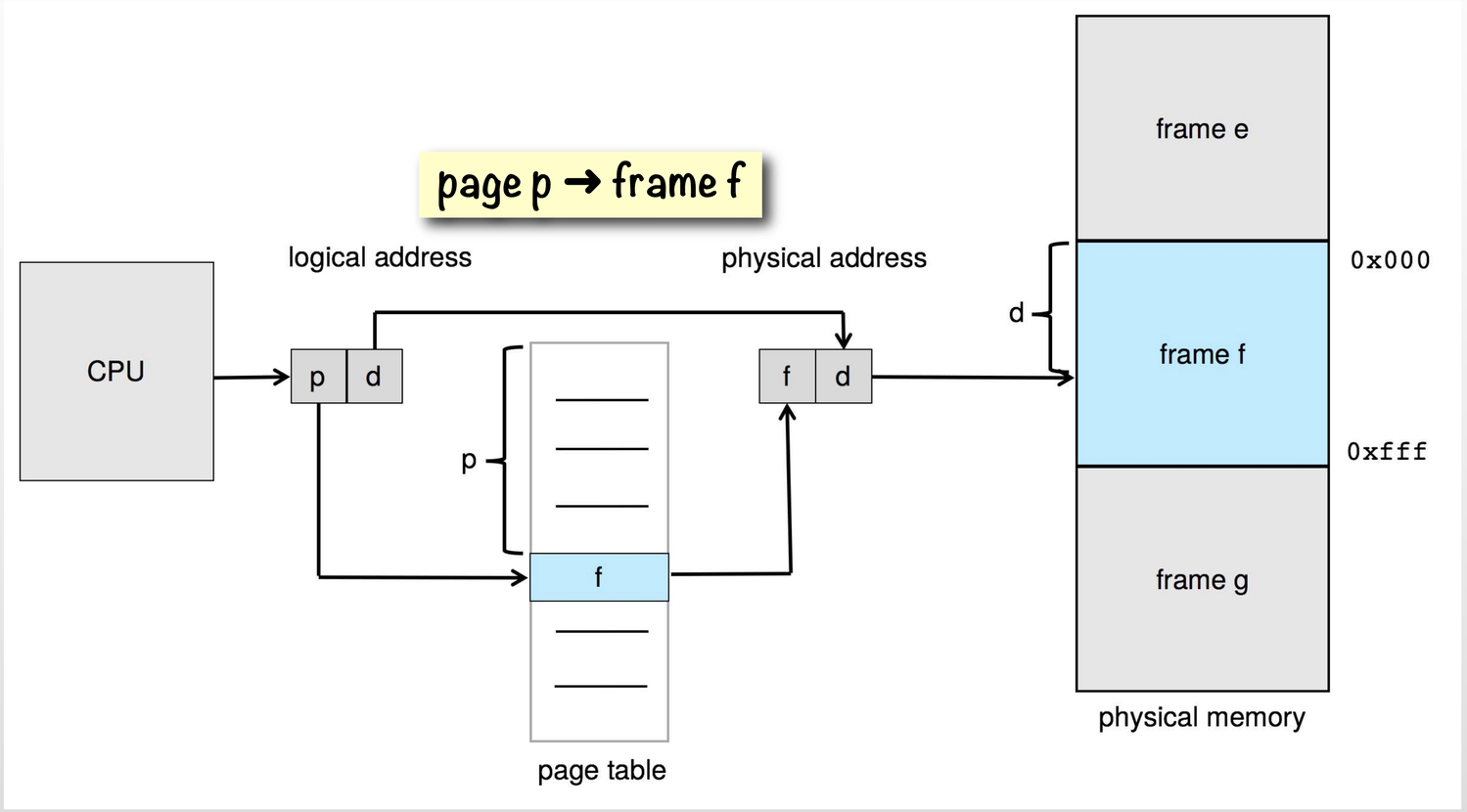


Address Translation Scheme in Paging

- Address generated by CPU is divided into two parts:
  - **page number** (p)
    - used as an index into a page table which contains base address of each frame in physical memory holding a page of the program
  - **page offset** (d)
    - combined with base address to define the physical memory address that is sent to the memory unit



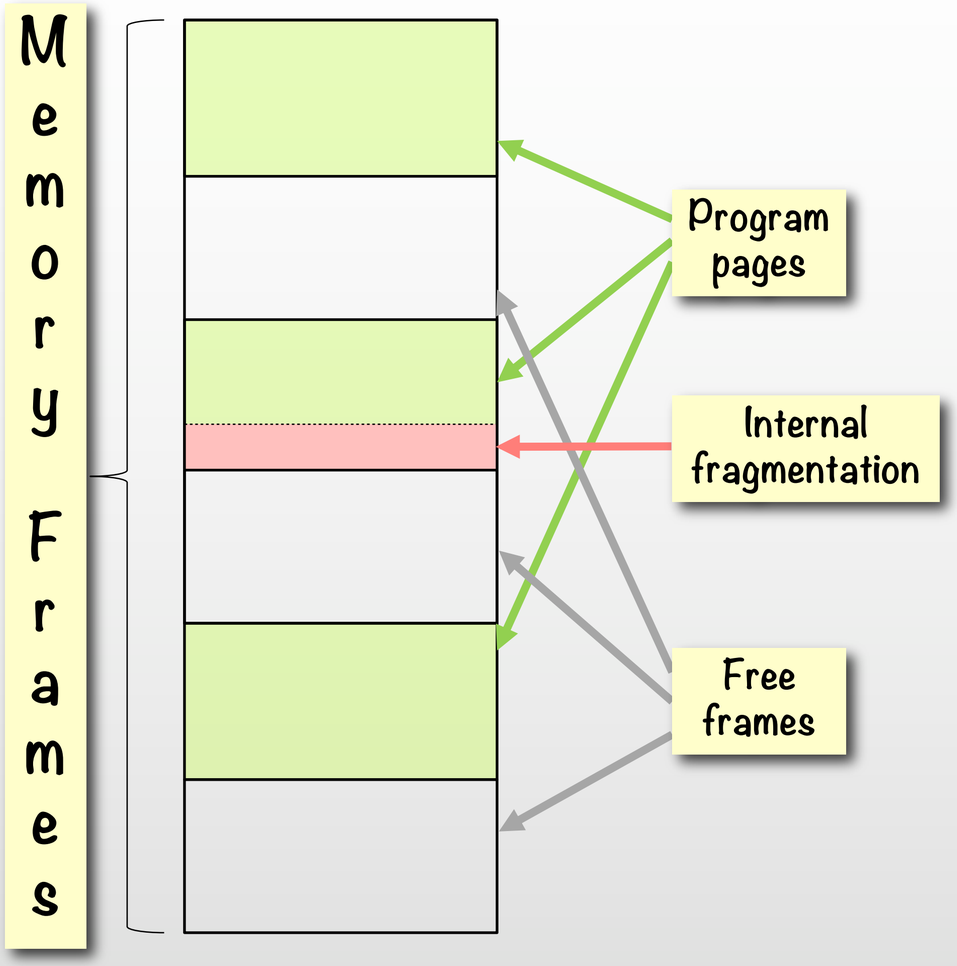
- For given logical address space 2^m and page size 2^n



Internal Fragmentation

- Inherent **internal fragmentation** due to pagination of programs
  - if the size of a program does not divide evenly by the size of the page, the last frame in the allocated space will be only partially filled
  - therefore, the allocated memory may be slightly larger than requested

- We can reduce internal fragmentation by adjusting page size
  - the smaller the page, the better fit we can find
  - however, a smaller page increases the size of the page table, since we need to map more pages for a given program



Implementation of Page Table

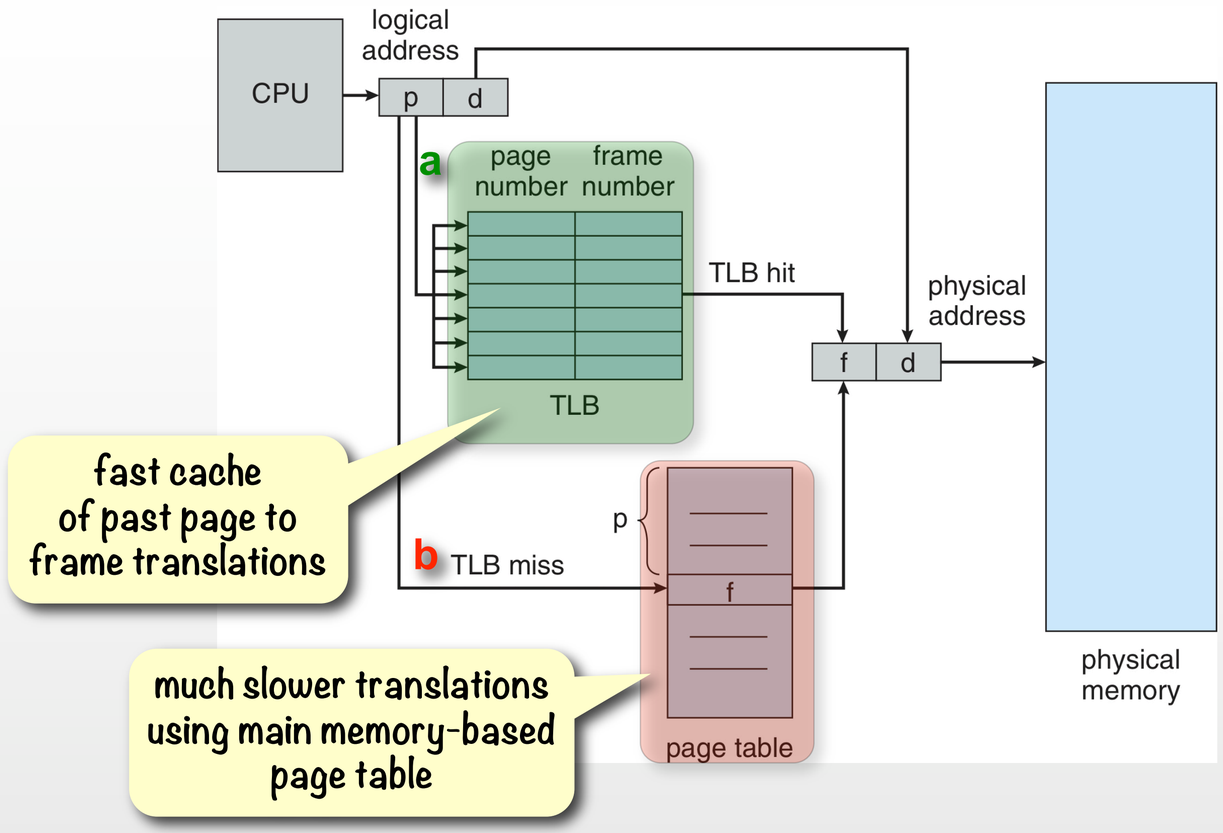
- A separate page table for each process, so we need to alter between tables on context switches
- If the page table is kept in main memory, we need:
  - **Page-table base register** (**PTBR**) points to the page table
  - **Page-table length register** (**PRLR**) indicates size of the page table
  - In this scheme every data/instruction access requires two memory accesses:
    - one for the page table and one for the data/instruction
    - this is not acceptable, since the translation is done for each address
- The access problem can be solved by the use of a special fast-lookup hardware cache called **translation look-aside buffers** (**TLBs**)
  - page to frame mapping is stored in that cache
  - search is done very fast, since there is no access to external memory, and the cache is searched with assistance from hardware (parallel search)
  - some TLBs store address-space identifiers (ASIDs) in each TLB entry
    - uniquely identifies each process to provide address-space protection for that process
      - if the ID of the process accessing the page does not match ASID, then an error is indicated
- Very fast associative memory used for TLBs
  - numerous TLBs can be placed between the CPU and the memory, or between the L caches

NOTE

TLBs are part of the MMU. They are different from L-caches!

L-caches cache everything between the CPU (or a core) and the memory, while TLBs cache page to frame translations

Paging Hardware With TLB



Assume
- memory access time is m time units
- we have to access the memory at least once.

Associative lookup = ε time units.

Hit ratio = α
- percentage of times that a page number is found in the associative registers related to number of associative registers.

Hence, (1 – α) is the miss ratio.

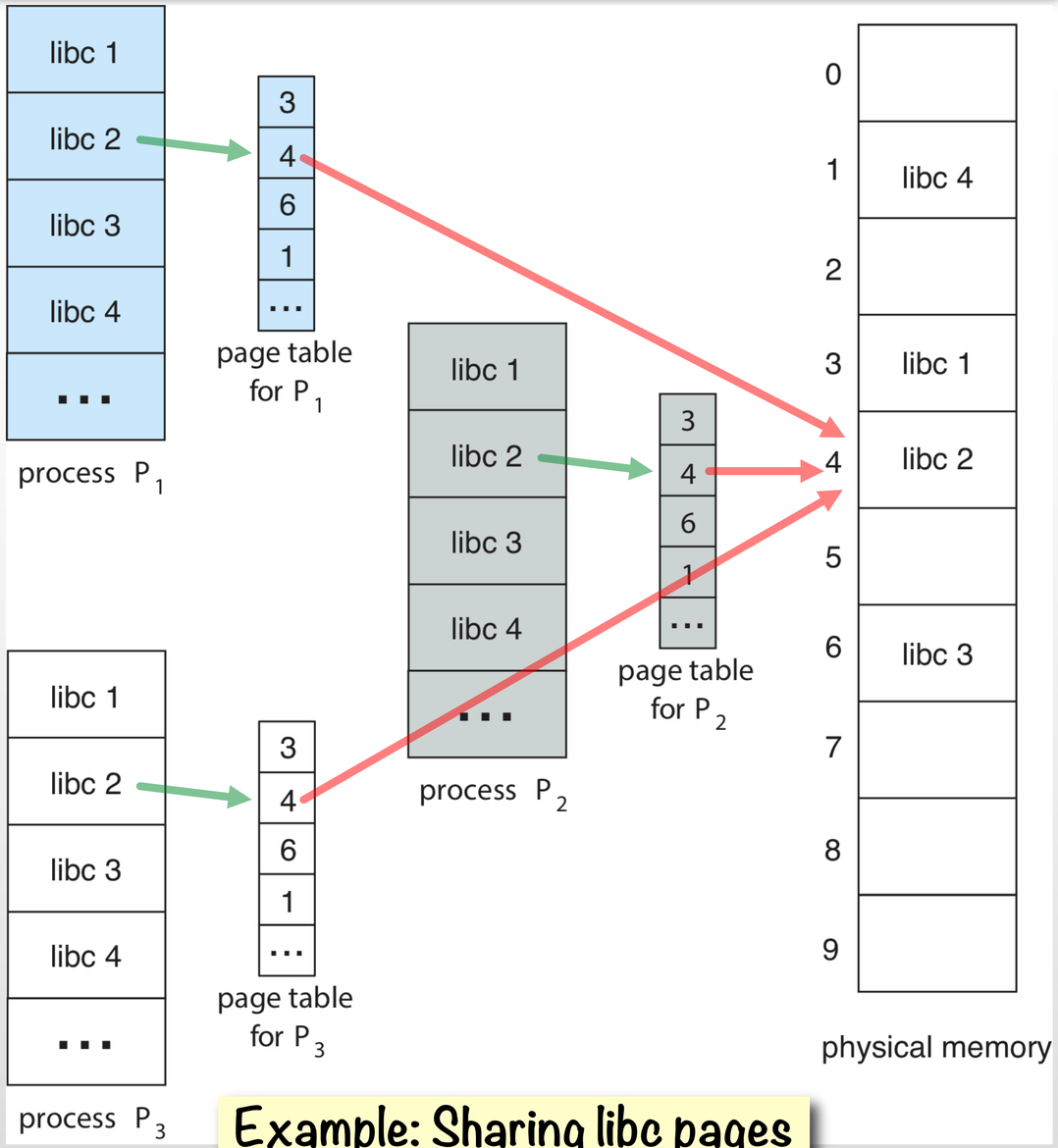
Effective Access Time (EAT)

$$EAT = m + \varepsilon\alpha + (m + \varepsilon)(1 - \alpha)$$

- TLB performs parallel search
  - if page number **p** is in one of the associative registers, get frame number **f** from the TLB (**a**)
  - otherwise get frame number **f** from the page table in main memory (**b**)
    - and save it in the TLB possibly replacing another number

Shared Pages

- Paging conveniently accommodates resource sharing
- Shared code
  - One copy of read-only (reentrant) code shared among processes
    - i.e., text editors, compilers, window systems
  - Shared code must appear in same location in the logical address space of all processes
- Private code and data
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space



Example: Sharing libc pages

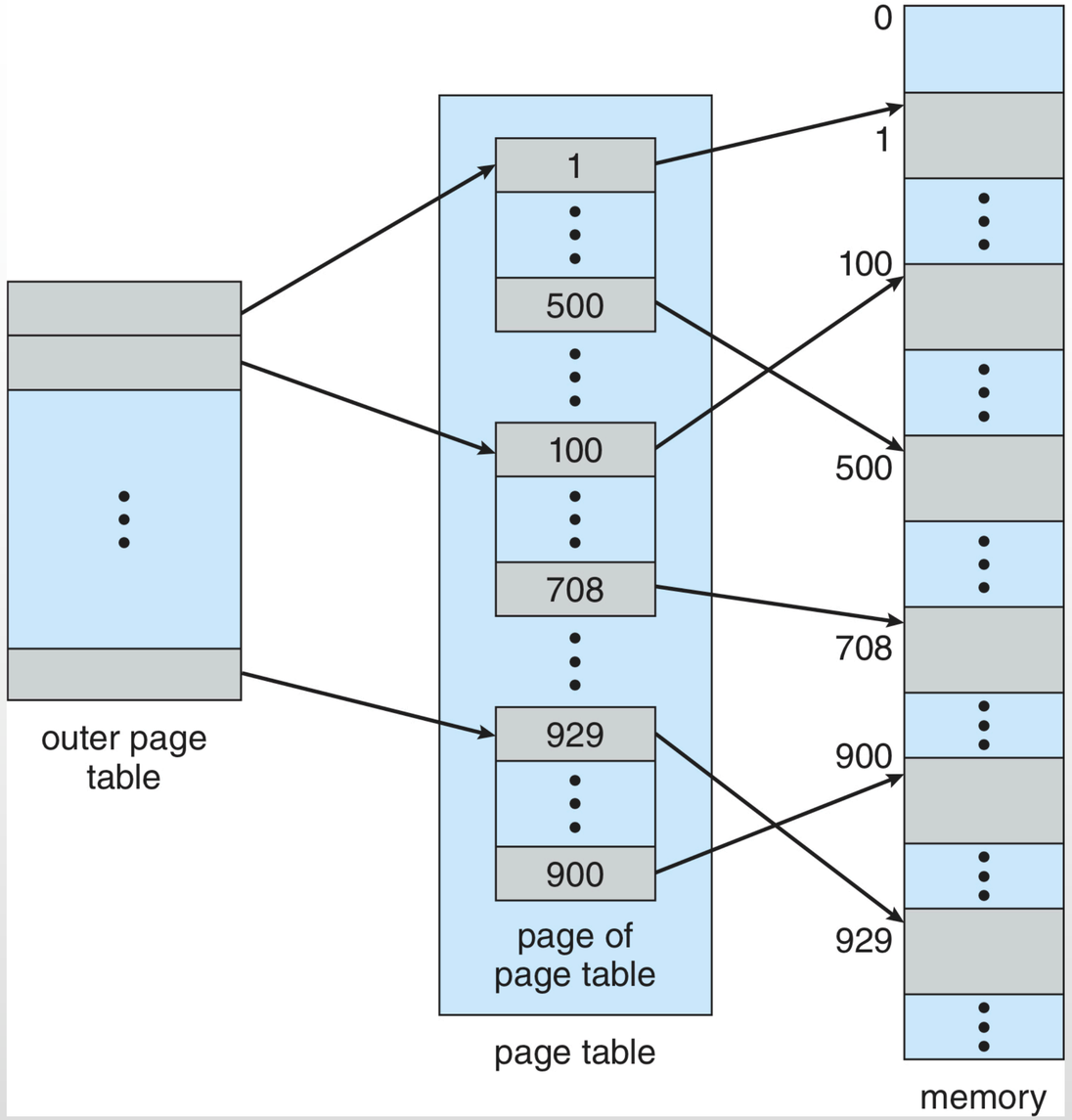
Structure of the Page Table

- Page tables might get large
  - e.g., with 32-bit addressing and 4 kB pages we may programs with maximum $2^{32}/2^{12} = 2^{20}$ pages, so if we have 4 bytes for each entry, we need $2^2 \times 2^{20} = 2^{22}$ bytes (2^{12} kB = 2^2 MB) for largest programs
    - ...not bad, but:
  - for 64-bit architecture, it would be much worse (up to 2^{24} TB of 4 kB pages!)

- To address the issue, we may use different page tables:
  - **Hierarchical Page Tables**
  - **Hashed Page Tables**
  - **Inverted Page Tables**

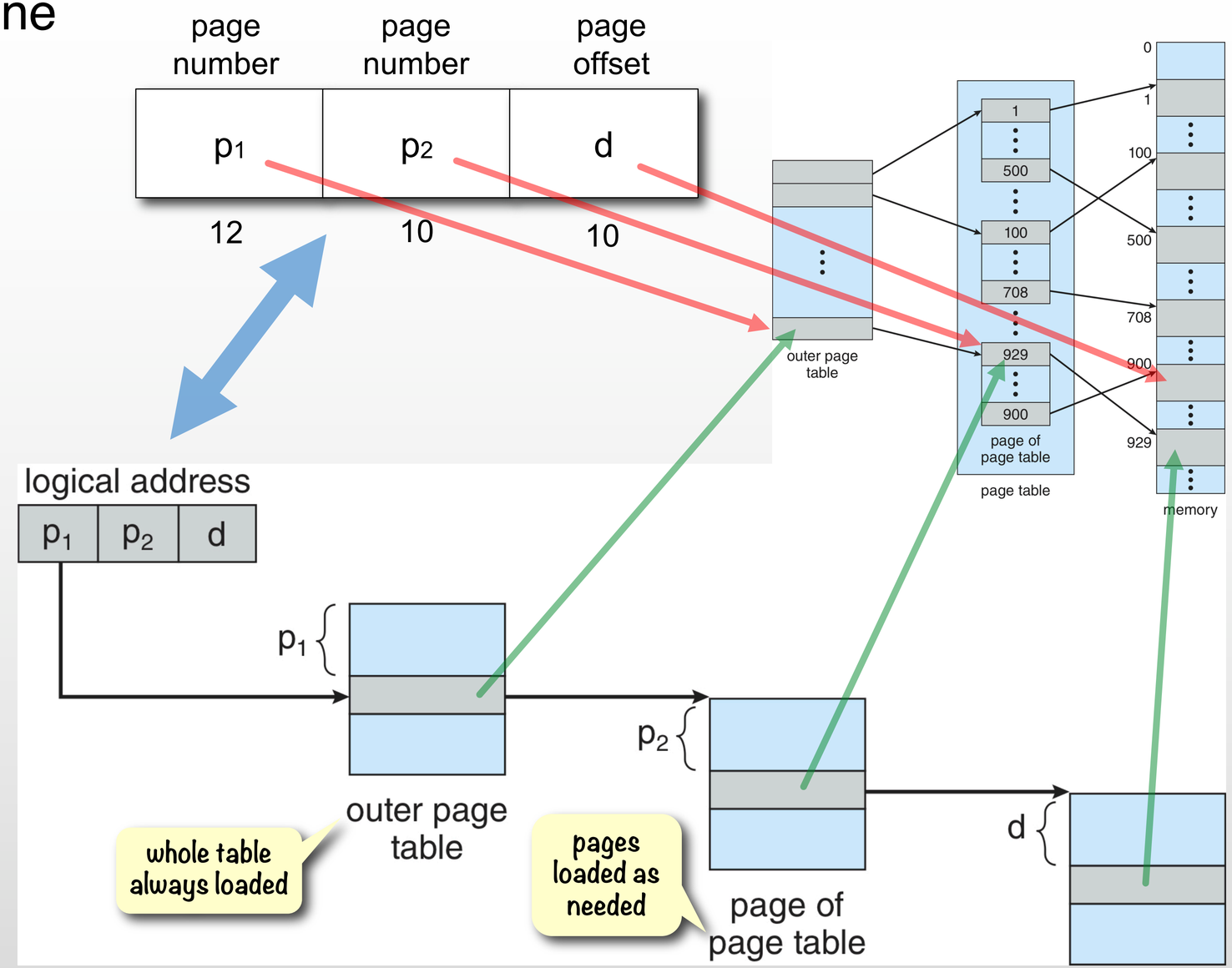
Hierarchical Paging

- Break up the logical address space into multiple page tables
  - i.e., divide the page table into pages too!
- An example of a simple technique is a **two-level page table**
- Only some pages of the page table need to be in memory, so the space required is smaller than keeping a monolithic table in memory
  - more about this will come in the next lecture on virtual memory



Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
    - offset for program page
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset
    - offset for the page of page table
- Thus, a logical address is as follows:
  - where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table



Three-level Paging Scheme

- The hierarchy can be built up

outer page	inner page	offset
p_1	p_2	d
42	10	12

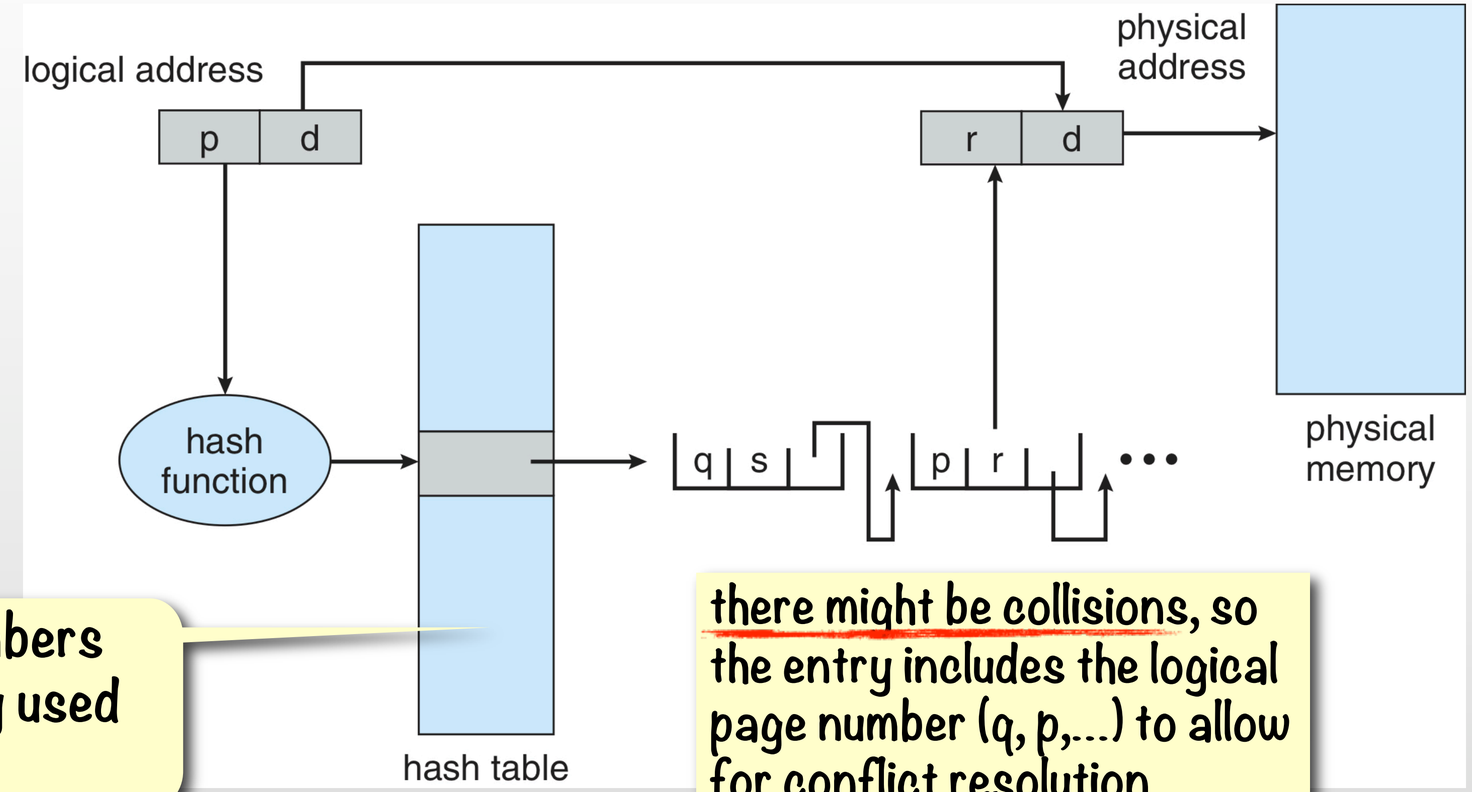
2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

- Four levels also used
  - ...and even higher levels possible

- Unfortunately, the access time is increasing with every level
  - recall that we keep the page table in memory, so that may take several cycles to access after TLB misses

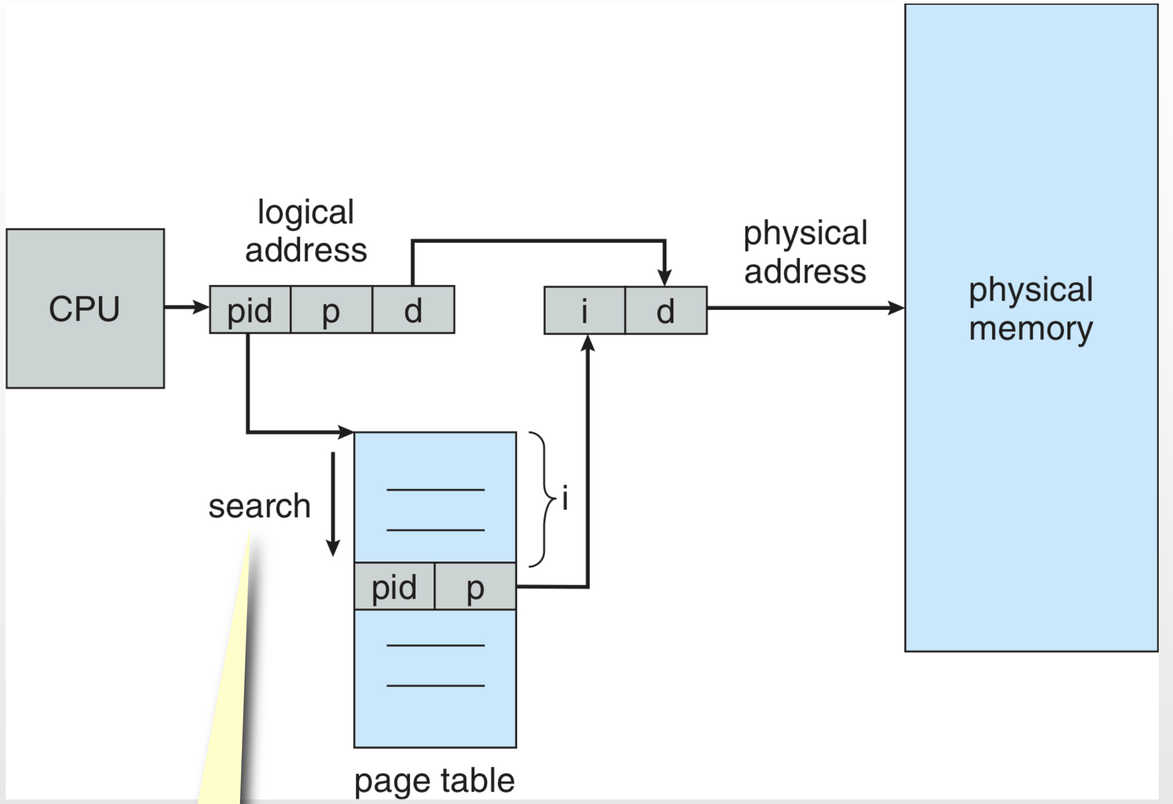
Hashed Page Tables

- Common in address spaces 32 bits and larger
- The virtual page number is hashed into a page table. This page table contains a chain of elements hashing to the same location.
- Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.



Inverted Page Table

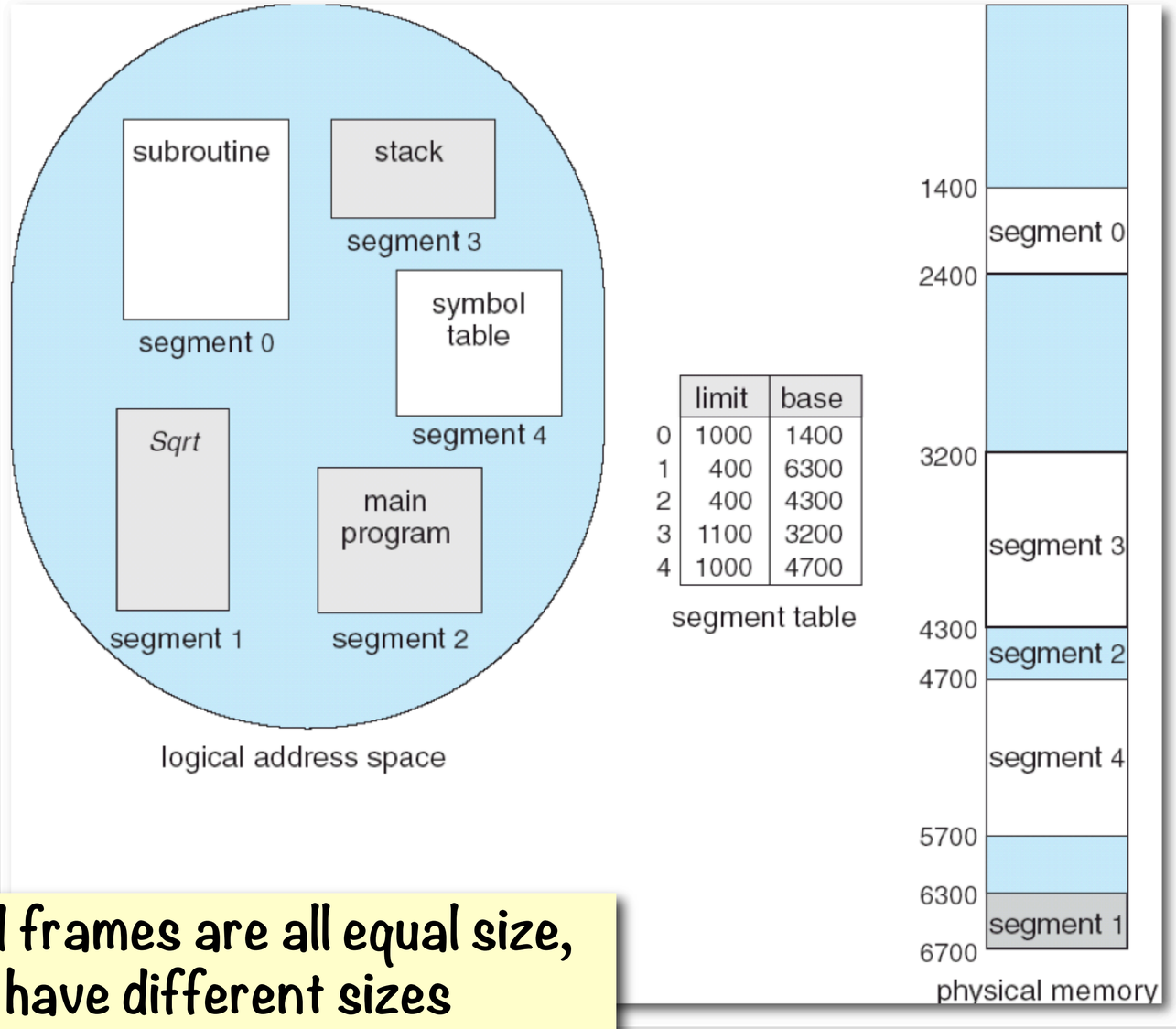
- In this scheme, there is one table for all processes with one entry for each real page of memory
  - rather than individual pages for all processes
- Entry consists of the virtual address of the page stored in that real memory location (p), and information about the process that owns that page (pid)
- The offset of the entry in the table is the base
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs



we can also use hash table to limit the search to one — or at most a few — page-table entries

Non-Contiguous Memory Allocation: Segmentation

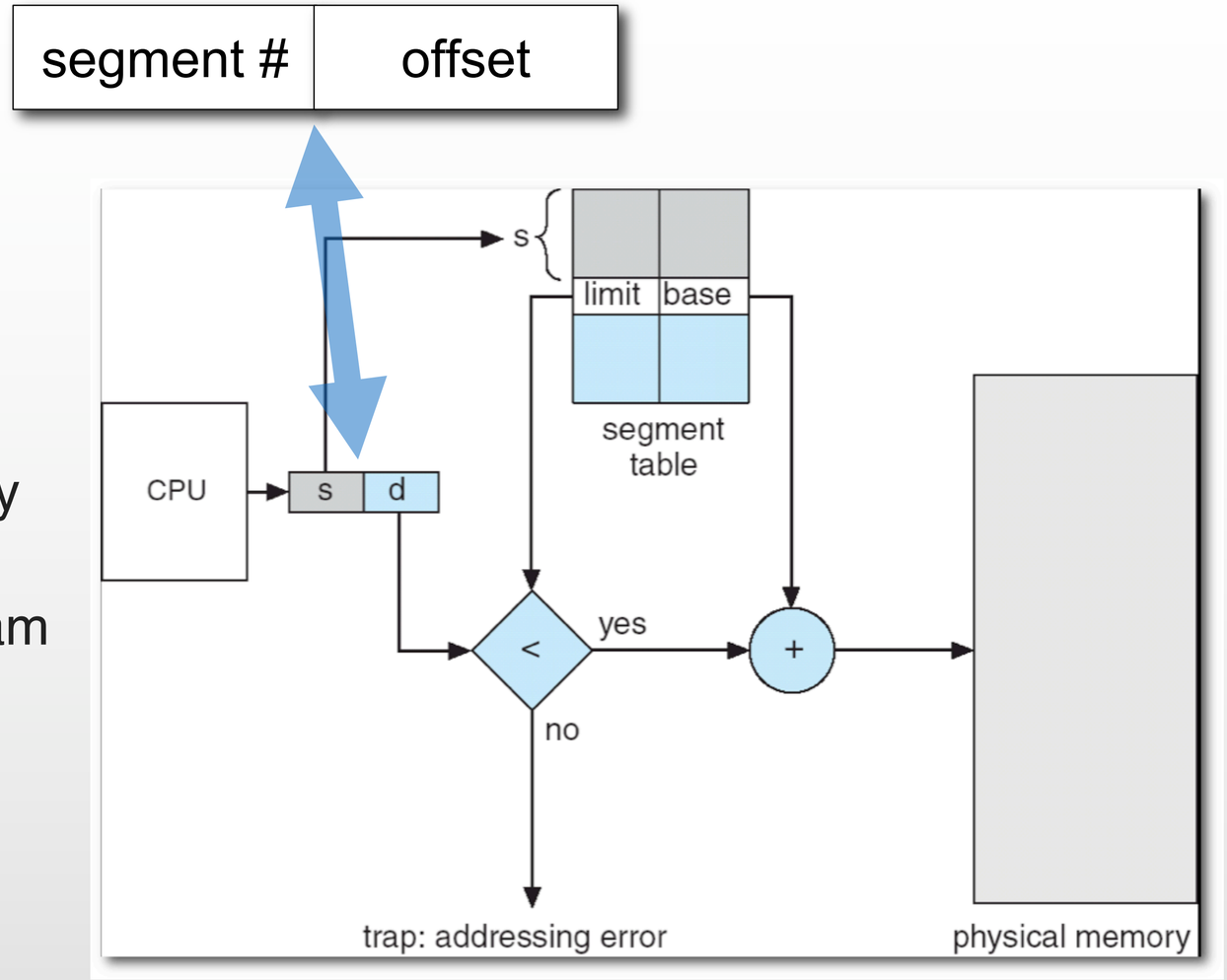
- Memory-management scheme that supports user view of memory:
  - a program is a collection of segments
- A segment is a logical unit such as:
  - main program,
  - procedure,
  - function,
  - method,
  - object,
  - local variables, global variables,
  - common block,
  - stack,
  - symbol table, arrays
- The user view is mapped by OS onto the physical memory space
  - just like for pages, but…



while pages and frames are all equal size, segments have different sizes

Segmentation Architecture

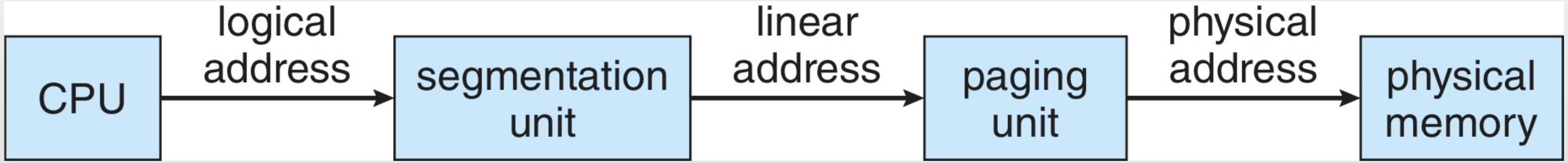
- Logical address consists of a two-tuple
- Segment table
  - maps two-dimensional physical addresses
  - each table entry has:
    - base – contains the starting physical address where the segments reside in memory
    - limit – specifies the length of the segment
- **Segment-table base register (STBR)**
  - points to the segment table's location in memory
- **Segment-table length register (STLR)**
  - indicates number of segments used by a program
    - segment number s is legal if s < STLR
- Protection
  - with each entry in segment table associate:
    - validation bit = 0 ⇒ illegal segment
    - read/write/execute privileges
- Protection bits associated with segments
  - code sharing occurs at segment level



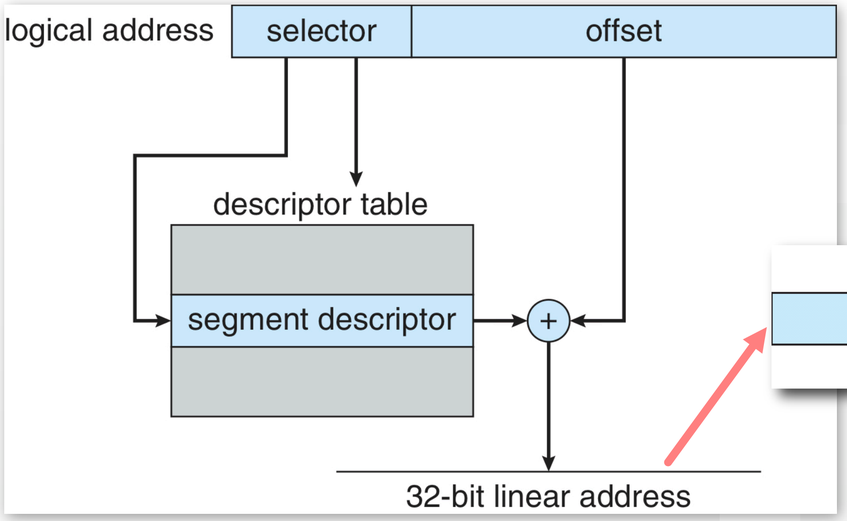
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem

Hybrid Segmentation-Paging Addressing

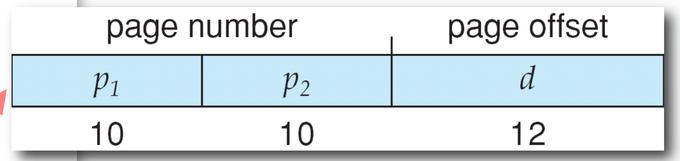
- Supports both segmentation and segmentation with paging
- CPU generates logical address
  - given to segmentation unit
    - which produces linear addresses
  - linear address given to paging unit
    - which generates physical address in main memory
    - paging units form equivalent of MMU

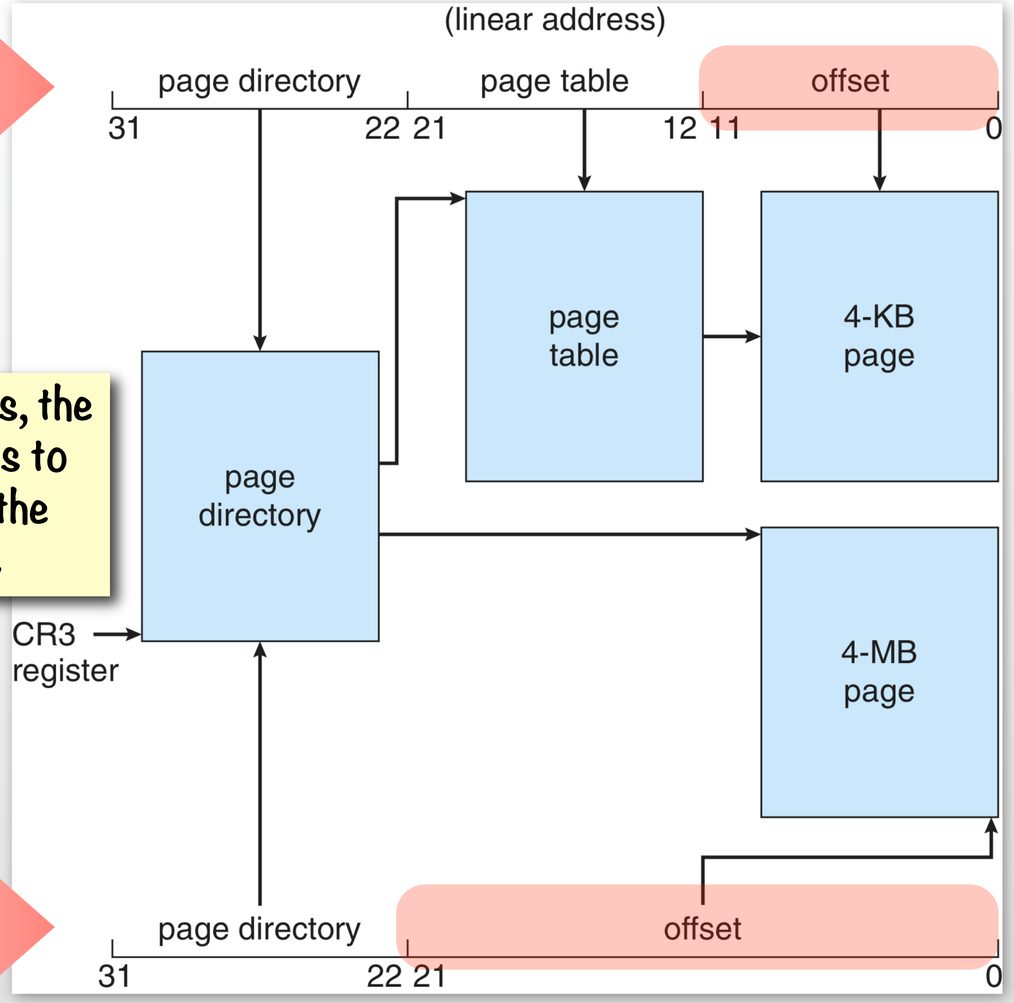


Hybrid Segmentation-Paging Addressing



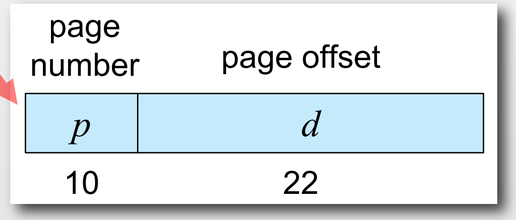
4 kB (2^{12}) pages





On Intel architectures, the CR3 register points to the page table for the current process

support for 4 kB (2^{12}) pages (arrow up) and 4 MB (2^{22}) pages (arrow down) through Page Size flag



4 MB (2^{22}) pages

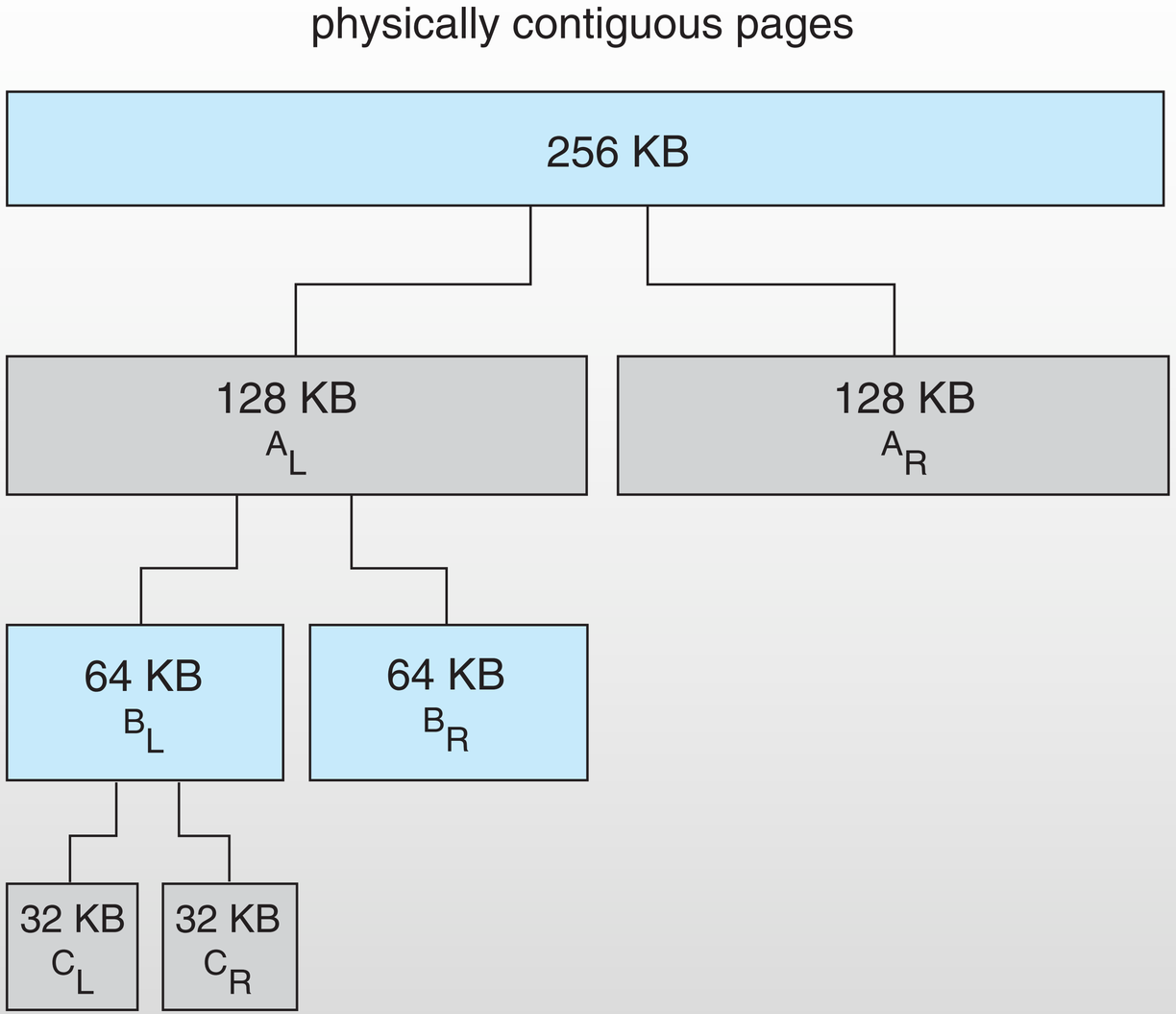
Allocating Memory for Kernel

- Kernel memory is often allocated from a free-memory pool different from the list used to satisfy ordinary user-mode processes
  - `malloc()` vs. `kmalloc()`

- Two primary reasons for this:
  - the kernel requests memory for data structures of varying sizes, some of which are less than a page in size
    - the kernel must use memory conservatively and attempt to minimize waste due to fragmentation
    - especially important because many operating systems do not subject kernel code or data to the paging system
  - certain hardware devices interact directly with physical memory and consequently may require memory residing in physically contiguous pages

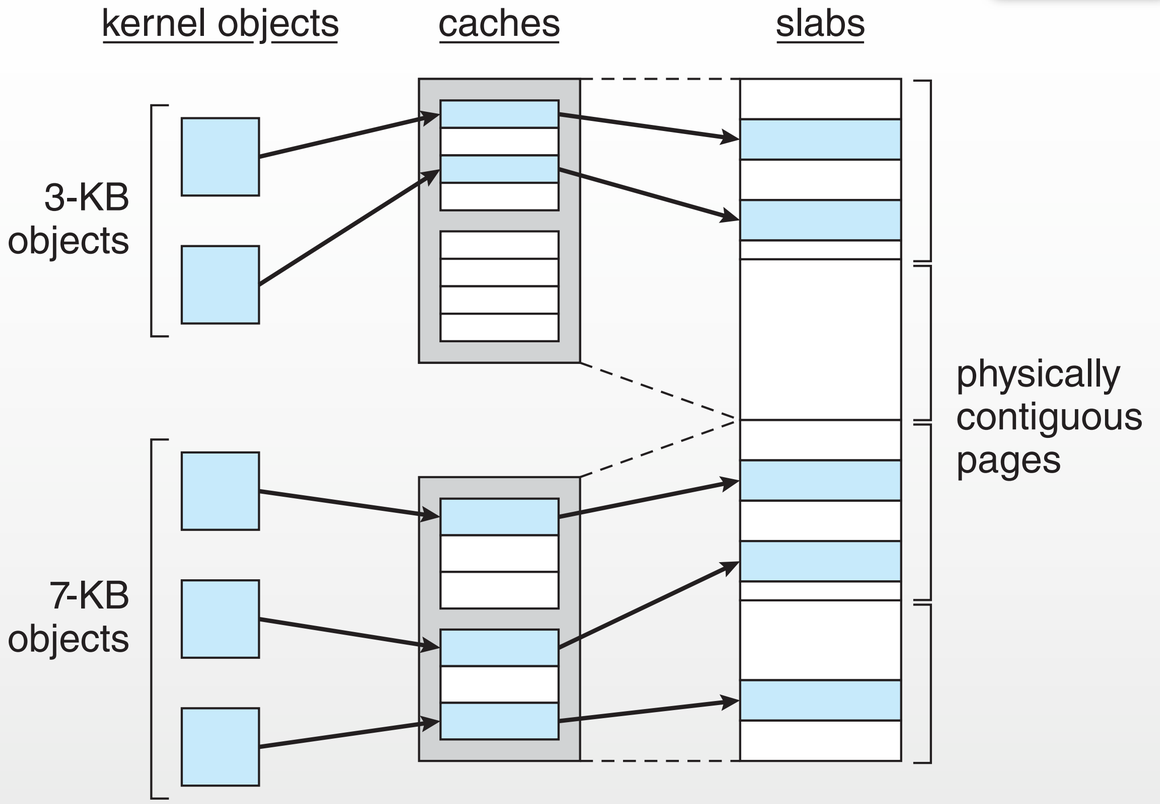
Linux Buddy System

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using power-of-2 allocator
  - Satisfies requests in units sized as power of 2
    - hence “buddies”
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
    - Continue until appropriate sized chunk available



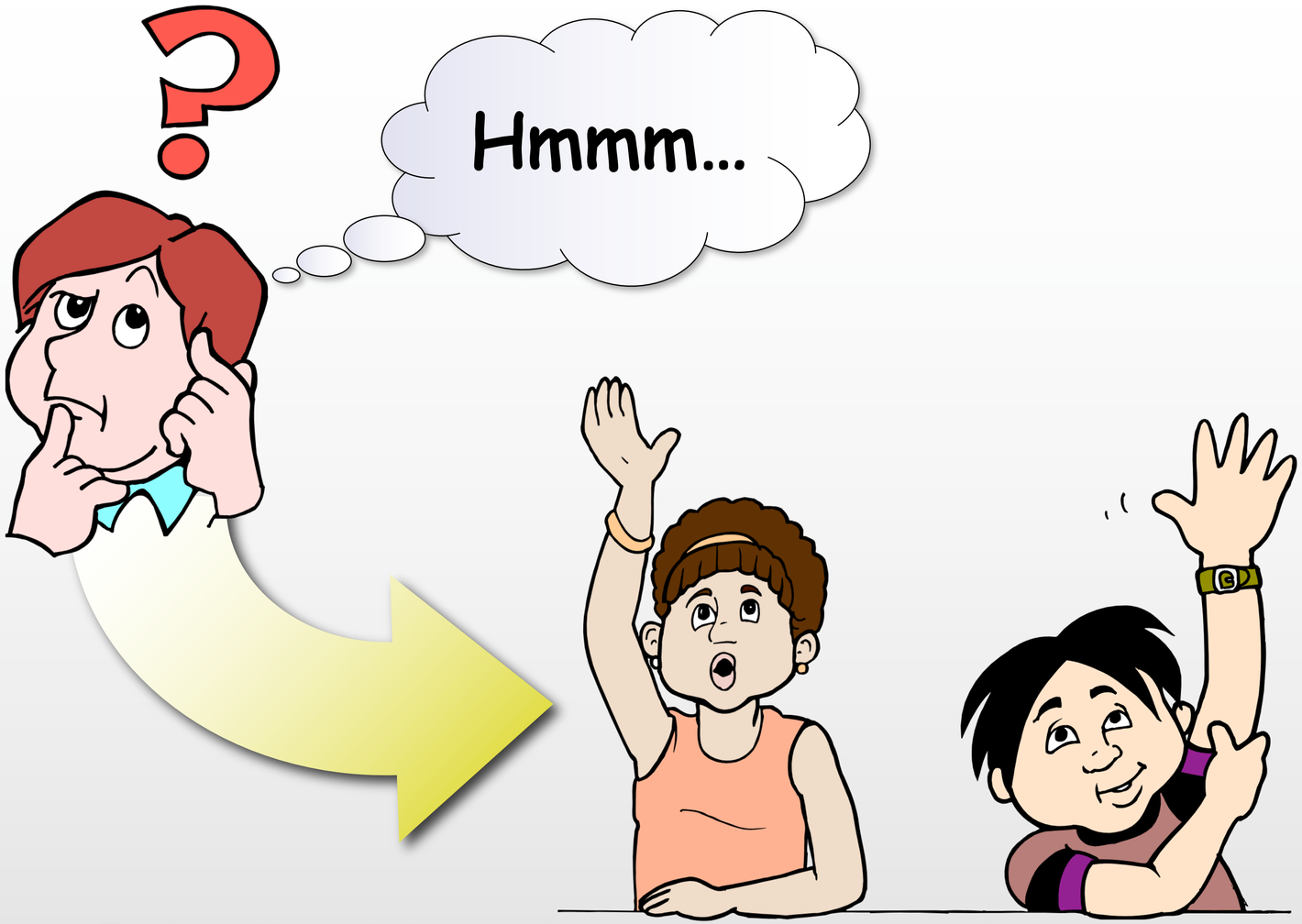
Linux Slab Allocation

- A **slab** is made up of one or more physically contiguous pages.
- A **cache** consists of one or more slabs.
- Each cache is populated with **objects** that are instantiations of the kernel data structure the cache represents.
  - for example, a separate cache for the data structure representing process descriptors, a separate cache for file objects, a separate cache for semaphores, and so forth.
- A slab may be in one of three possible states:
  - full - all objects in the slab are marked as used
  - empty - all objects in the slab are marked as free
  - partial - the slab consists of both used and free objects
    - the slab allocator first attempts to satisfy the request with a free object in a partial slab
    - if none exists, a free object is assigned from an empty slab
    - if no empty slabs are available, a new slab is allocated from contiguous physical pages and assigned to a cache; memory for the object is allocated from this slab.



Slab Allocation Benefits

- The slab allocator provides two main benefits:
  - No (or at least less) memory is wasted due to fragmentation
    - because each unique kernel data structure has an associated cache, and each cache is made up of one or more slabs that are divided into chunks the size of the objects being represented
    - when the kernel requests memory for an object, the slab allocator returns the exact amount of memory required to represent the object
  - Memory requests can be satisfied quickly
    - the slab allocation scheme is particularly effective for managing memory when objects are frequently allocated and deallocated, as is often the case with requests from the kernel
      - saved time on memory allocation and release
    - objects are created in advance and thus can be quickly allocated from the cache
    - when the kernel has finished with an object and releases it, it is marked as free and returned to its cache



COMP362 Operating Systems

Prof. AJ Bieszczad