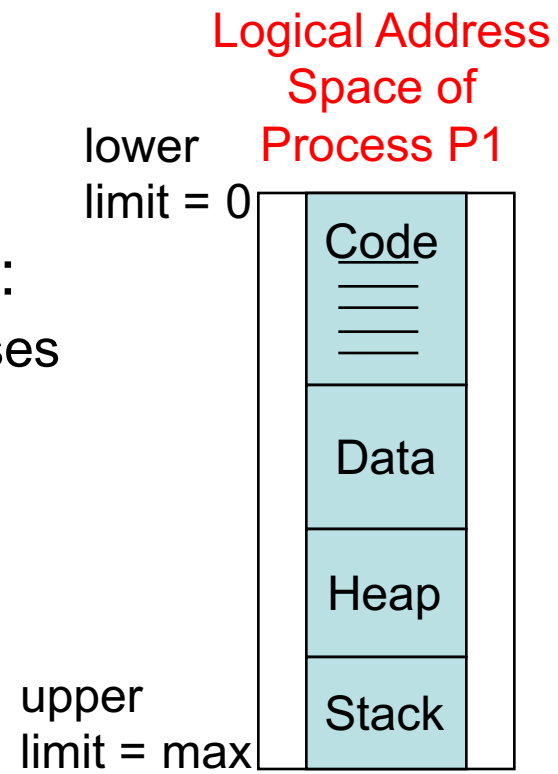
Chapter 8: Paging

CSCI 3753 Operating Systems

William Mortl & Dr. Rick Han

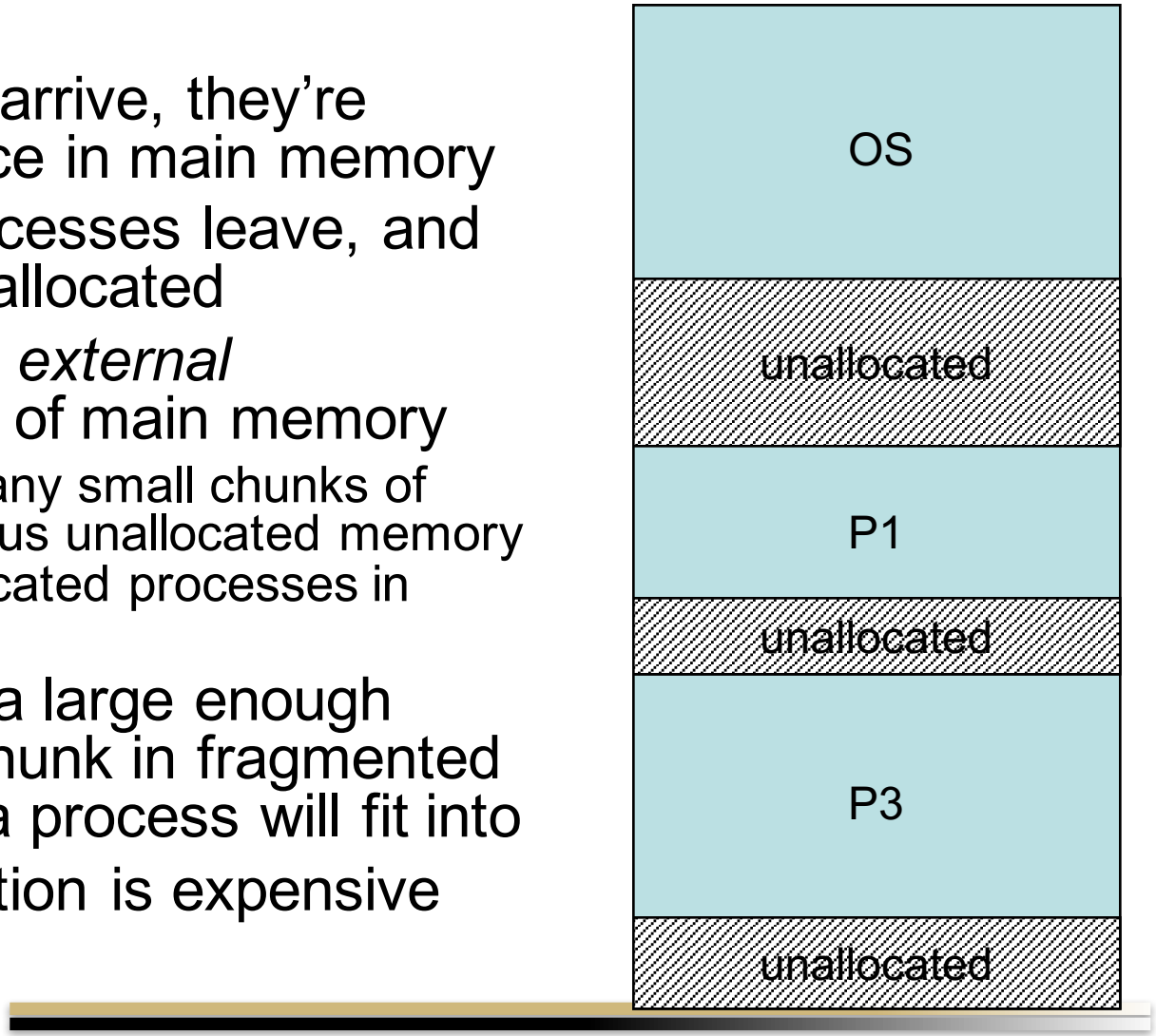
Recap

- Memory management of memory hierarchy
- Give processes the illusion of each having their own private virtual/logical address space
  - Memory management unit (MMU):
    1. translate virtual to physical addresses
    2. check bounds, e.g. base & limit registers
  - run-time binding: V->P at each assembly instruction
  - static/dynamic linking
- Swapping and swap space



Fragmentation

- as processes arrive, they're allocated space in main memory
- over time, processes leave, and memory is deallocated
- This results in *external fragmentation* of main memory
  - There are many small chunks of non-contiguous unallocated memory between allocated processes in memory
- OS must find a large enough unallocated chunk in fragmented memory that a process will fit into
- De-fragmentation is expensive



Paging

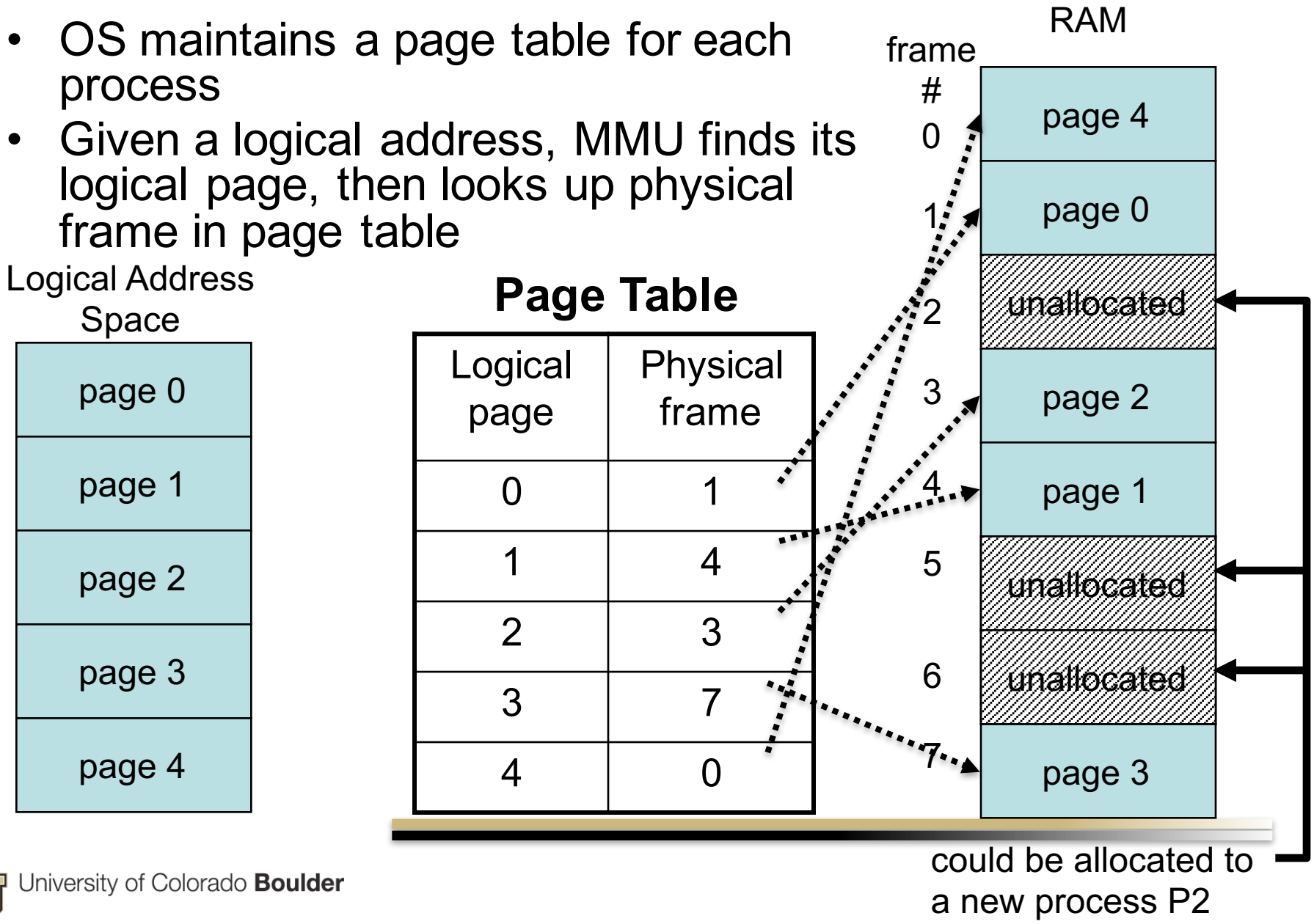
- A better solution to external fragmentation is to divide the logical address space into fixed-size *pages*
  - each logical/virtual page is mapped to a similarly sized physical frame in main memory
    - thus main memory is divided into fixed-size frames
  - *the frames don't have to be contiguous in memory*
  - At run time, given a virtual address:
    1. figure out what virtual page it belongs to
    2. then determine which physical page in memory stores that virtual page,
    3. then offset into that physical page to obtain the precise address

Paging

- Each process is now split into pages that are scattered non-contiguously throughout memory
- this solves the external fragmentation problem:
  - when a new process needs to be allocated, and there are enough free pages, then find any set of free pages in memory
- Need a *page table* to keep track of where each logical page of a process is located in main memory
  - to keep track of the mapping of each logical page to a physical memory frame

Paging

- OS maintains a page table for each process
- Given a logical address, MMU finds its logical page, then looks up physical frame in page table



Paging

- user's view of memory is still as one contiguous block of logical address space
  - MMU performs run-time mapping of each logical address to a physical address using the page table
- Typical page size is 4-8 KB
  - Example: a 4 GB 32-bit address space with 4 KB/page (2^{12}) => $2^{32}/2^{12}$ = 1 million entries in page table
    - Your page table would need to be >= 20 bits/entry
  - example: if a page table allows 32-bit entries, and if page size is 4 KB, then can address 2^{44} bytes = 16 TB of memory
  - There is support for huge/large pages ~ 4 MB each

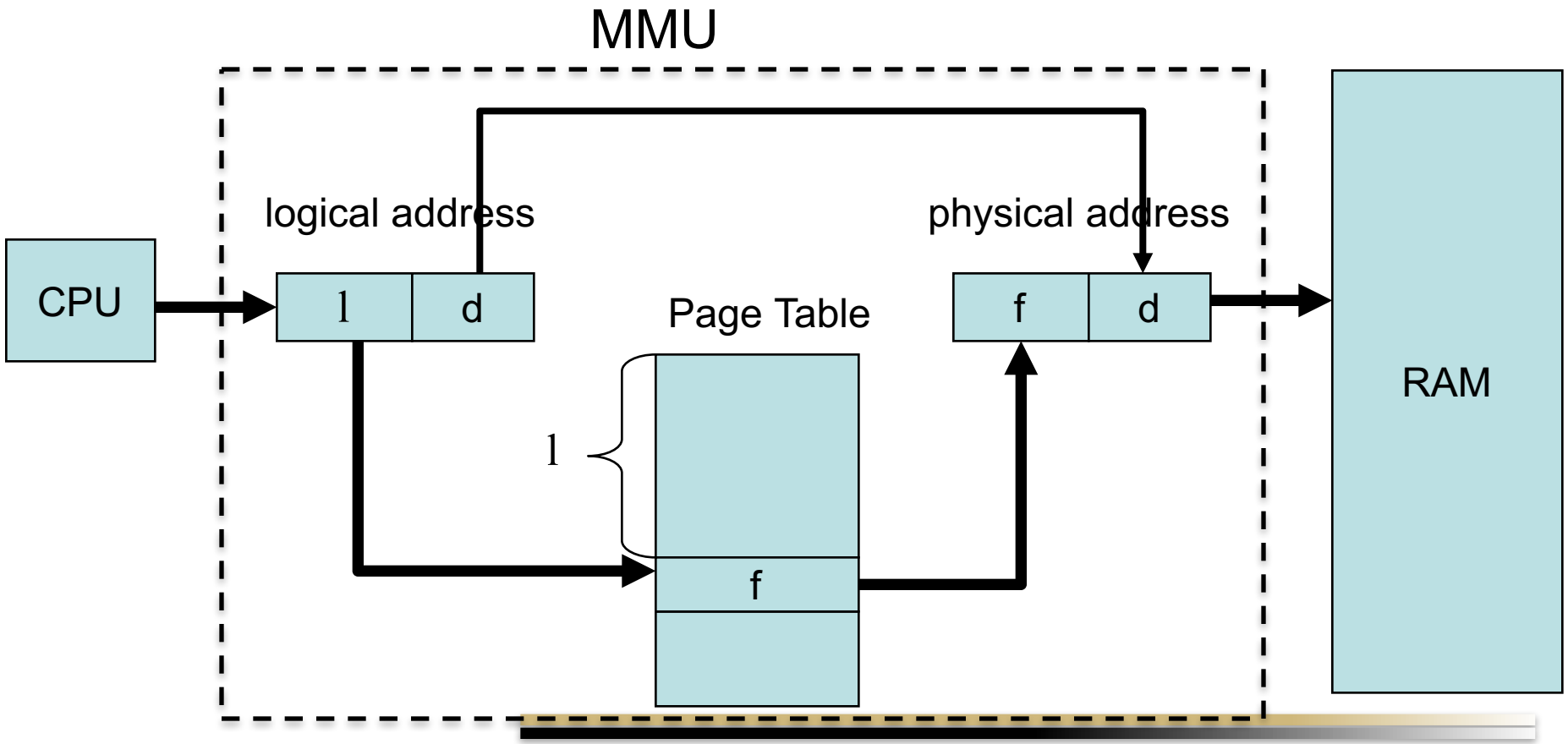
Paging

- No external fragmentation
- But we do get some *internal fragmentation*
  - example: suppose my process is size 4001 B, and each page size is 4 KB
    - then I have to allocate two pages = 8 KB,
    - 3999 B of 2nd page is wasted due to fragmentation that is internal to a page
- OS also has to maintain a frame table/pool that keeps track of what physical memory frames are free

Paging

- Conceptually, every logical/virtual address can be divided into two parts:
  - most significant bits = logical page # l,
    - Equals the virtual address / page size
    - used to index into page table to retrieve the corresponding physical frame *f*
  - least significant bits = page offset *d*

Paging


MMU
logical address
physical address
CPU
l
d
Page Table
f
d
RAM
l
f

Paging

- Implementing a page table:
  - option #1: use dedicated bank of hardware registers or memory to store the page table
    - fast per-instruction translation
    - slow per context switch - entire page table has to be reloaded
    - limited by cost (expensive hardware) to being too small - some page tables can be large, e.g. 1 million entries – too expensive

Paging

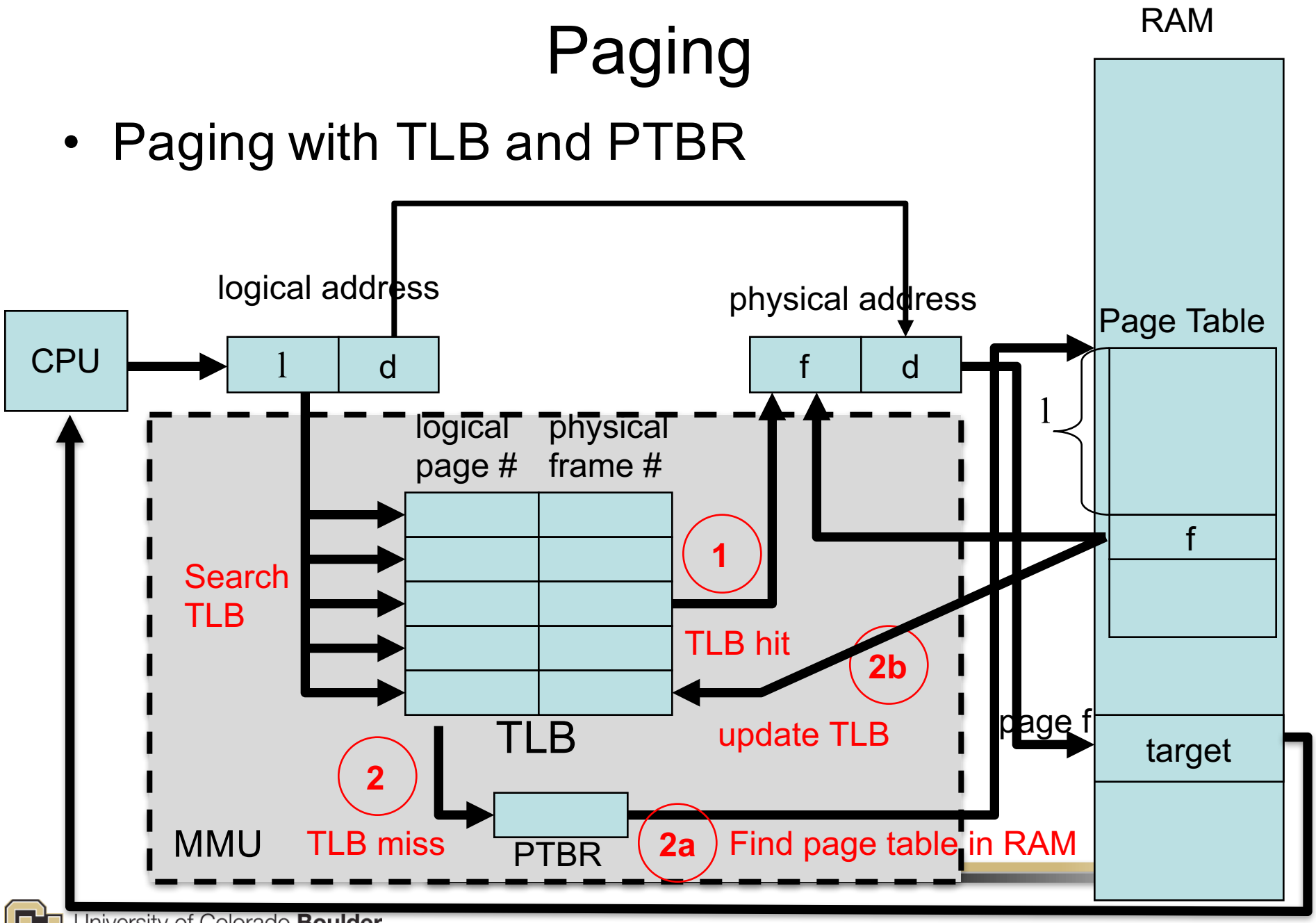
- Implementing a page table:
  - option #2: store the page table in main memory
    - keep a pointer to the page table in a special CPU register called the Page Table Base Register (PTBR)
    - can accommodate fairly large page tables
    - fast context switch - only reload the PTBR!
    - slow per-instruction translation, because each instruction fetch requires *two steps*:
      1. finding the page table in memory and indexing to the appropriate spot to retrieve the physical frame # f
      2. retrieving the instruction from physical memory frame f

Paging

- Implementing a page table:
  - Option #3: cache a subset of page table mappings/entries in a small set of CPU buffers called Translation-Look-aside Buffers (TLBs)
    - Fast solution to option #2's slow two-step memory access
    - Several TLB caching policies:
      - cache the most popular or frequently referenced pages in TLB
      - cache the most recently used pages

Paging

- Paging with TLB and PTBR



Paging and TLB Caching

- Summarize steps depicted in the graph on the last slide
- MMU in CPU first looks in TLB's to find a match for a given logical address
  1. if match found, then quickly call main memory with physical address frame f (plus offset d)
     - this is called a *TLB hit*
     - TLB as implemented in hardware does a fast parallel match of the input page to all stored values in the cache - about 10% overhead in speed

     (continued next slide)

Paging and TLB Caching

(continued from previous slide)

2. if no match found, then this is a *TLB miss*
   a) go through regular two-step lookup procedure: go to main memory to find page table and index into it to retrieve frame #f, then retrieve what's stored at address <f,d> in physical memory
   b) Update TLB cache with the new entry from the page table
      – if cache full, then implement a cache replacement strategy, e.g. Least Recently Used (LRU) - we'll see this later

•Goal is to maximize TLB hits and minimize TLB misses

Paging and TLB Caching

- On a context switch, the TLB entries would typically have to all be invalidated/completely flushed
  - since different processes have different page tables
  - x86 behavior behaves likes this
- An alternative is to include process IDs in TLB
  - at the additional cost of hardware and an additional comparison per lookup
  - Only TLB entries with a process ID matching the current task are considered valid
  - See DEC RISC Alpha CPU

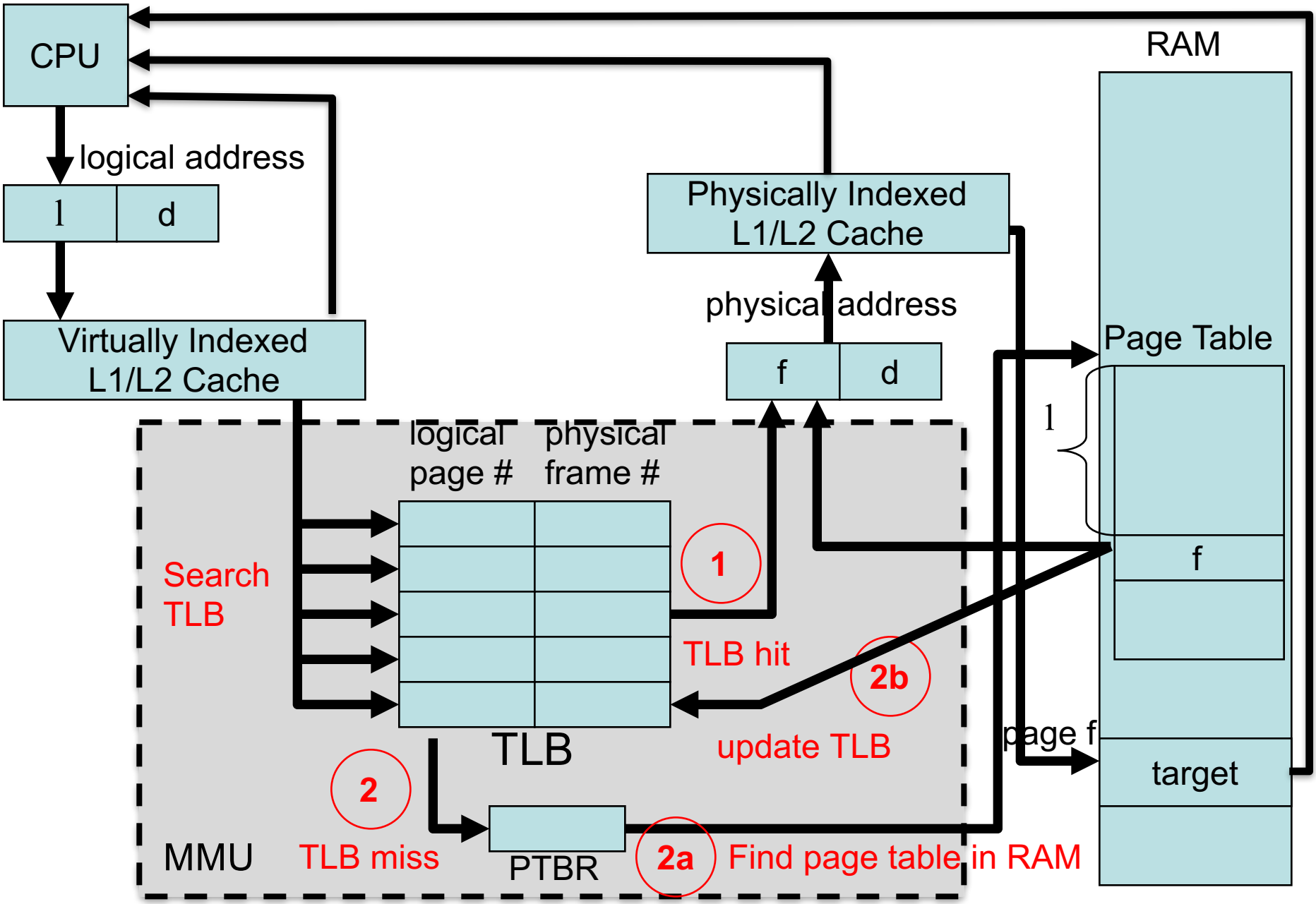
Paging and TLB Caching

- Another option: prevent frequently used pages from being automatically invalidated in the TLBs on a task switch
  - In Intel Pentium Pro, use the page global enable (PGE) flag in the register CR4 and the global (G) flag of a page-directory or page-table entry
- ARM allows flushing of individual entries from the TLB indexed by virtual address

Paging and L1/L2 Caching

- How does MMU interact with L1 or L2 data or instruction caches?
  - It depends on whether the items in a cache are viewed as virtual or physical
- L1/L2 data/instruction caches can store their information and be indexed by either virtual or physical addresses
  - If physical, then MMU must first convert virtual to physical, before the cache can be consulted – this is slow, but each entry is uniquely identifiable by its physical address
  - If virtual, then cache can be consulted quickly to see if there's a hit without invoking the MMU (if miss, then MMU must still be invoked…)

- Paging with TLB and L1/L2 Caching



Paging and L1/L2 Caching

- From prior figure:
  - The desired logical address is first searched for in a virtually indexed L1 or L2 cache if such a cache is provided by the hardware designer.
  - If a match is found, then CPU uses this cached item immediately.
  - Otherwise, the MMU must be invoked to convert the logical to physical address as described earlier.
    - TLB caching improves this speed of translation

Paging and L1/L2 Caching

- From prior figure:
  - After the MMU has produced the desired physical address, then a physically indexed L1 or L2 cache is searched for the desired physical address, if such a cache is provided by the hardware designer.
  - If a match is found, then CPU uses this cached item immediately.
  - Otherwise, must fetch item from main memory.

Paging and L1/L2 Caching

- A virtually indexed L1/L2 data/instruction caches introduces some problems:
  - Homonym problem: when a new process is switched in, it may use the same virtual address V as the previous process.
    - The cache that indexes just by virtual address V will return the wrong information (cached information from the prior process).

Paging and L1/L2 Caching

- Some solutions to the homonym problem:
  - Add an address space id (process id) to each entry of the cache
    - so only data/instructions for the right process are returned for a given virtual address V
    - requires hardware support and an extra comparison
    - reduces available cache space for each process, since it has to be shared
  - Flush the cache on each context switch
    - process gets entire cache to itself, but have to rebuild cache
  - Each process uses non-overlapping virtual addresses in its address space
    - unlikely, violates model that each process is compiled & executes independently in its own address space [0,MAX]

Paging and L1/L2 Caching

- In practice,
  - Most L1 caches are virtually indexed – fast
  - Most L2 caches are physically indexed
    - Each entry is unique
    - No collisions
    - This is good for code/data from shared library pages, i.e. if multiple processes share the same code/data, then it just has to be stored once in cache
  - Thus in the figure, the virtually indexed cache is essentially a small L1 cache, and the physically indexed cache is a much larger L2 cache.

Shared Pages

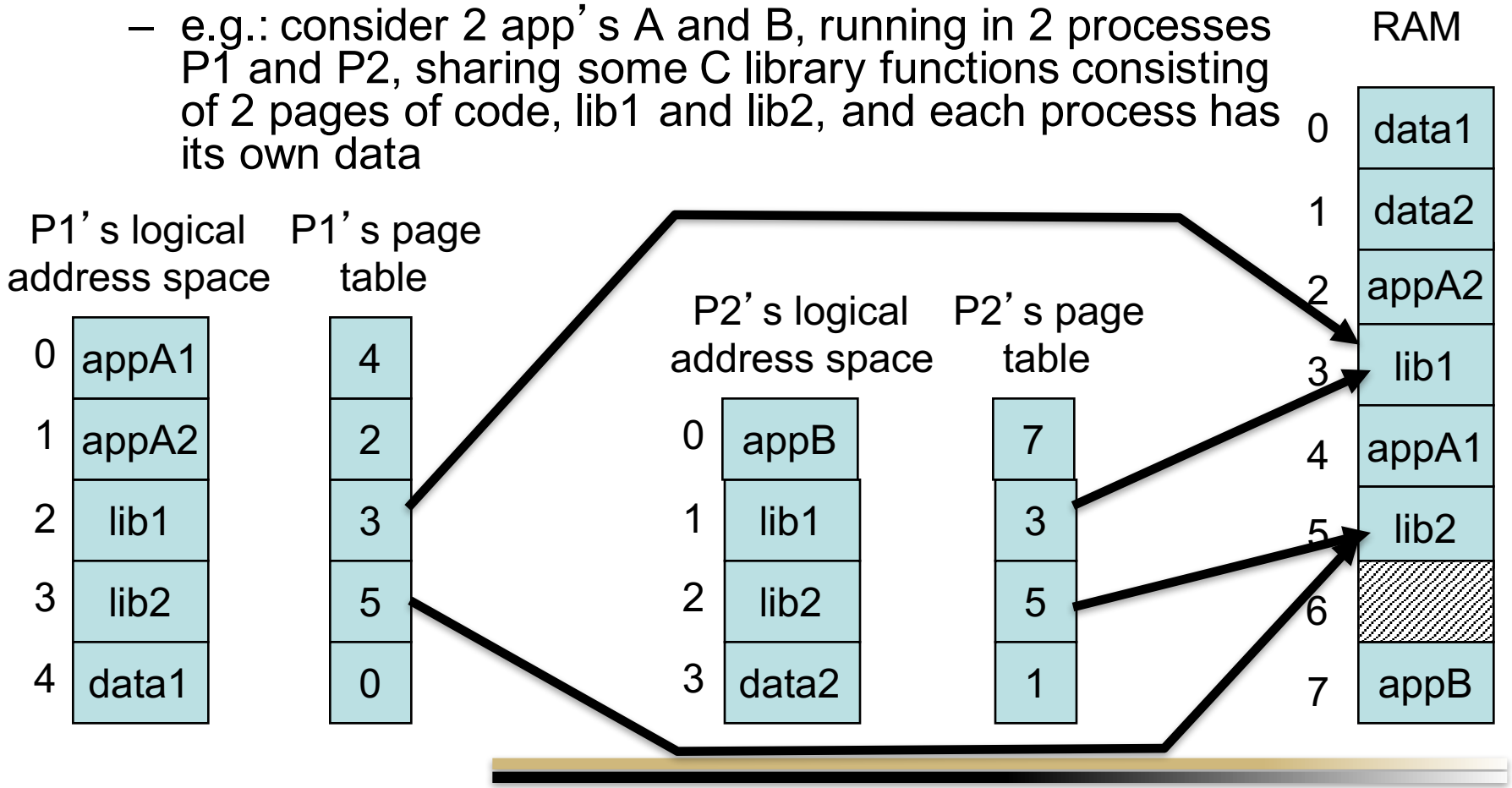
- Can share code & data pages between processes by having entries in different process' page tables point to the same physical page/frame(s)
- Sharing data:
  - Two or more processes may want to share memory between them, so pointing multiple page tables to the same data pages is a way to implement shared memory
  - Shared data should be protected by synchronization

Shared Pages

- Sharing code:
  - Fork()'ing a child process causes the child to have a copy of the entire address space of the parent, including code
    - Rather than duplicating all such code pages, can simply map the child's page table to point to the same set of code pages as the parent.
    - This is a way to implement copy-on-write
  - May want to share dynamically linked libraries (image, C, …) between multiple processes
    - so map each process' page table to point to the same dll pages in memory
  - Shared code should be thread-safe and reentrant

Shared Pages

- To achieve sharing of code and data, page tables can point to the *same* memory frames
  - e.g.: consider 2 app's A and B, running in 2 processes P1 and P2, sharing some C library functions consisting of 2 pages of code, lib1 and lib2, and each process has its own data

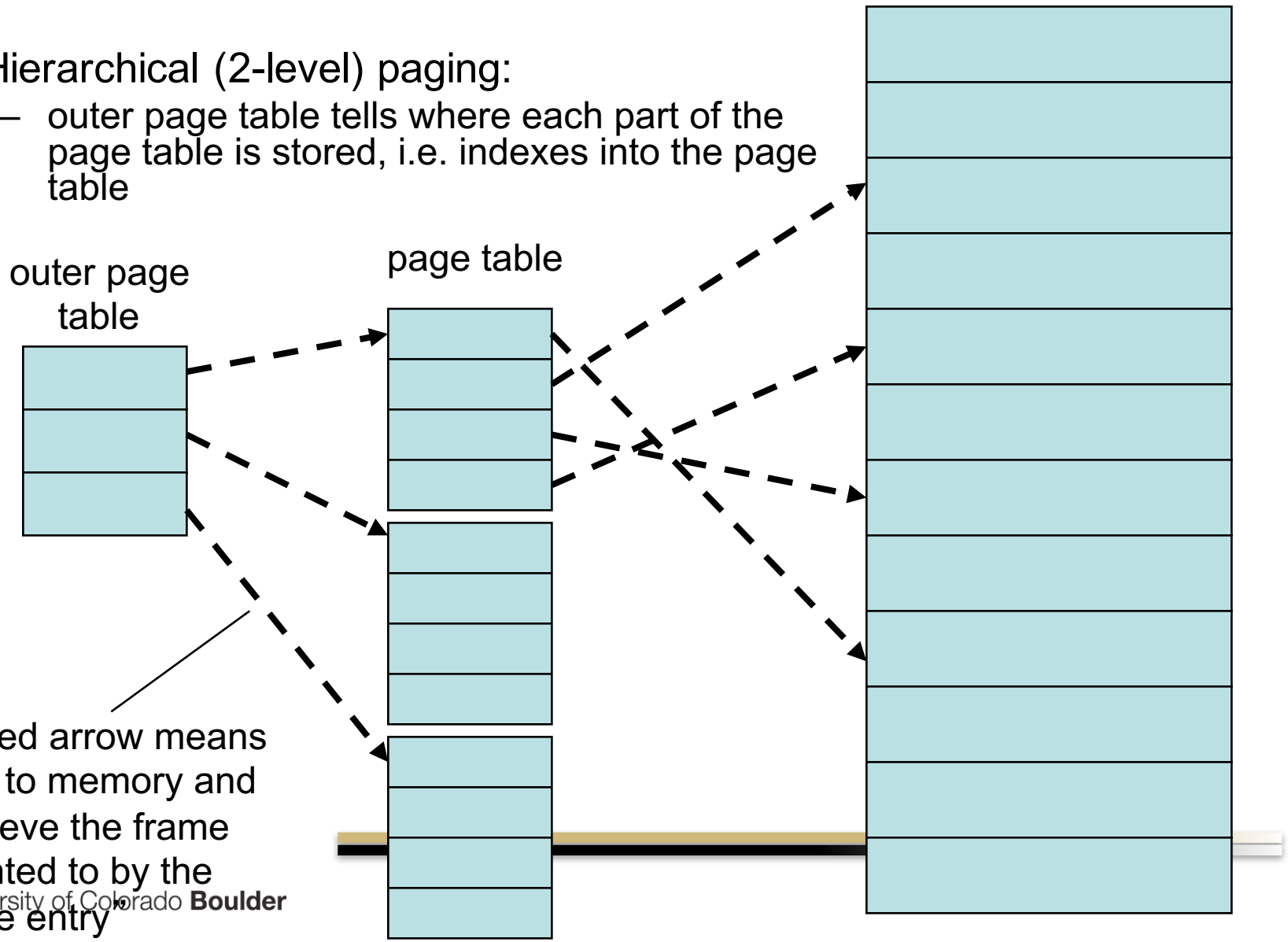


Hierarchical Paging

- Problem with page tables: they can get very large
  - example: 32-bit address space with 4 KB/page (212) implies that there are 232/212 = 1 million entries in page table per process
  - it's hard to find contiguous allocation of at least 1 MB for each page table
- solution: page the page table! this is an example of hierarchical paging.
  - Just as processes are split into pages to fit into a memory, split page table into pages to fit into memory
  - This is an example of 2-level paging. In general, we can apply N-level paging.

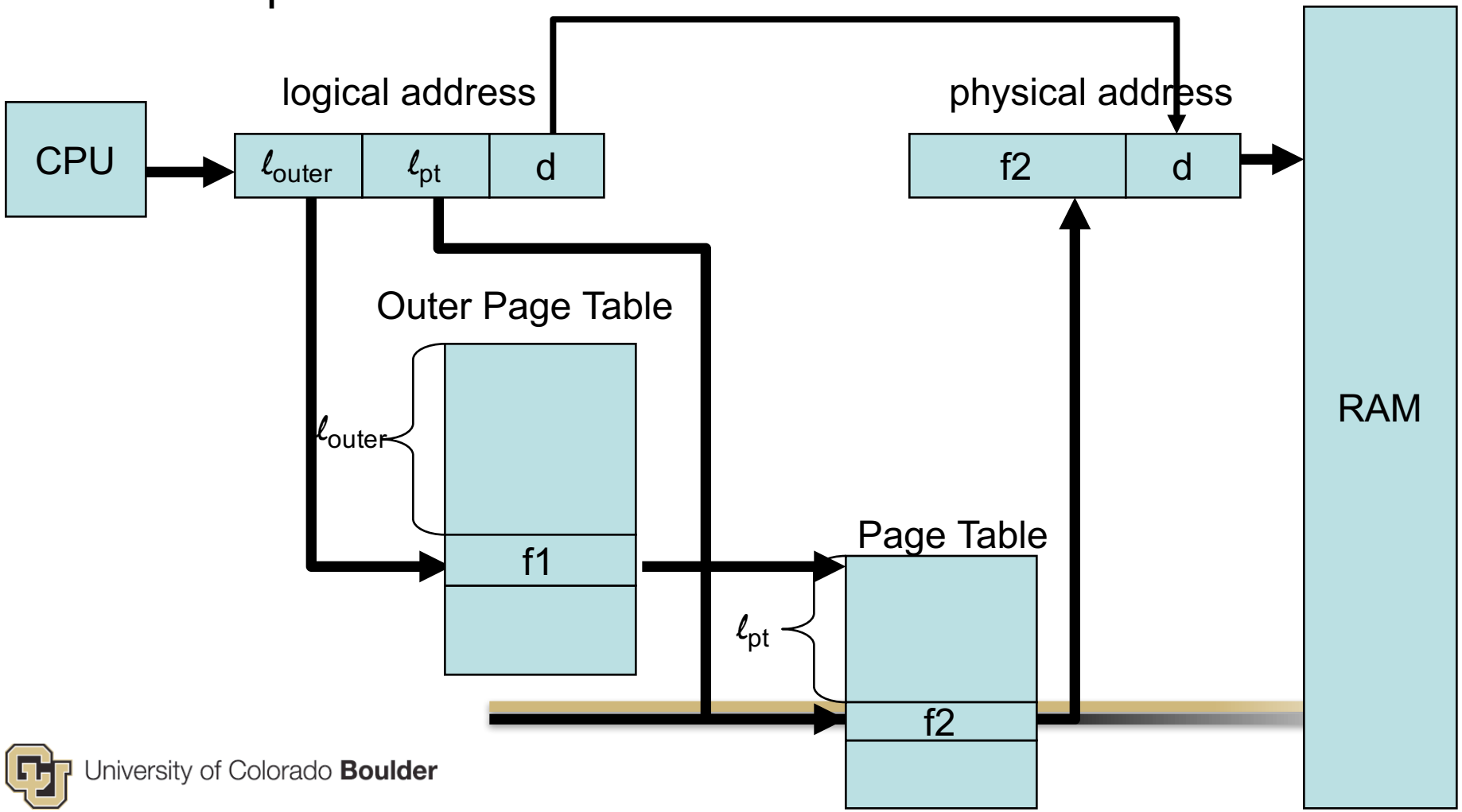
Hierarchical Paging

- Hierarchical (2-level) paging:
  - outer page table tells where each part of the page table is stored, i.e. indexes into the page table



Hierarchical Paging

- Hierarchical (2-level) paging divides the logical address into 3 parts:



Hierarchical Paging

- While hierarchical page tables enable a large page table to be split to fit into memory, they do not solve the problem of the large size of the page table
  - if several processes have a page table that contains a million entries, then much of RAM may become devoted to storing/managing multiple page tables
  - The page tables may be *sparse* - many of the entries in the page table are just empty placeholders for logical pages that are not in memory, and may never be in memory
    - e.g. stack and heap may never grow large enough to use all of allocated memory