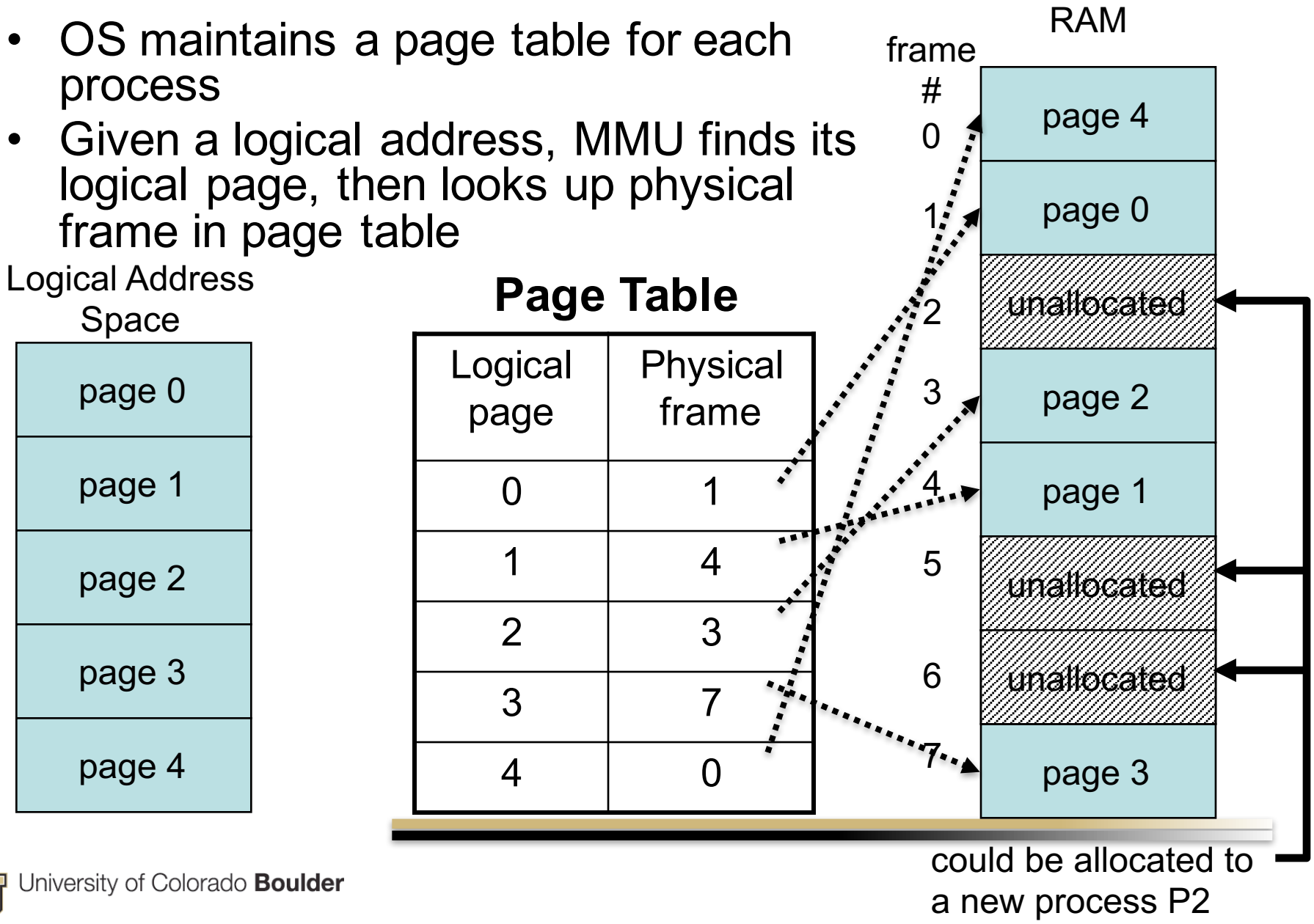
Chapter 8: Paging and Segmentation

CSCI 3753 Operating Systems

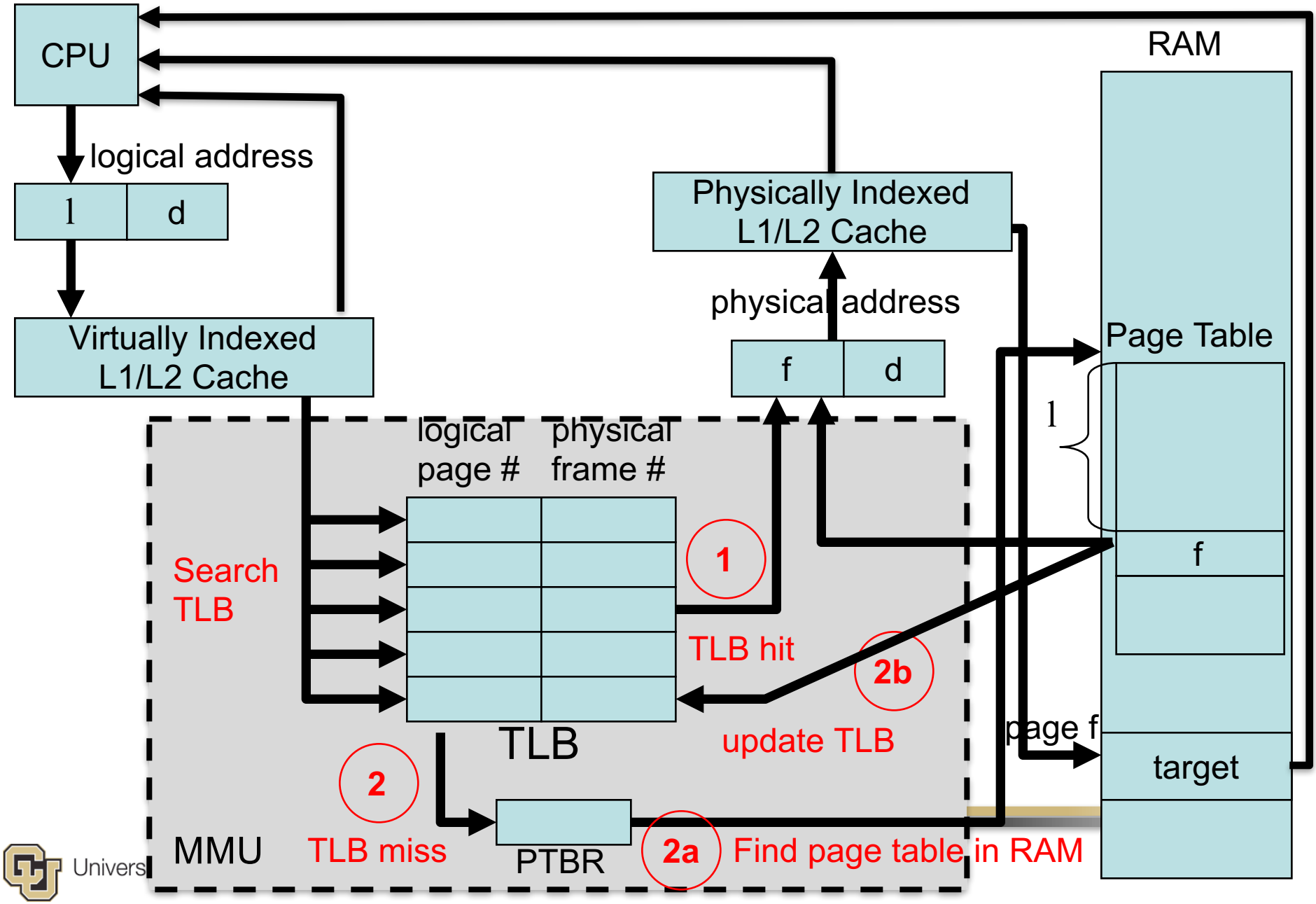
Prof. Rick Han

Recap: 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

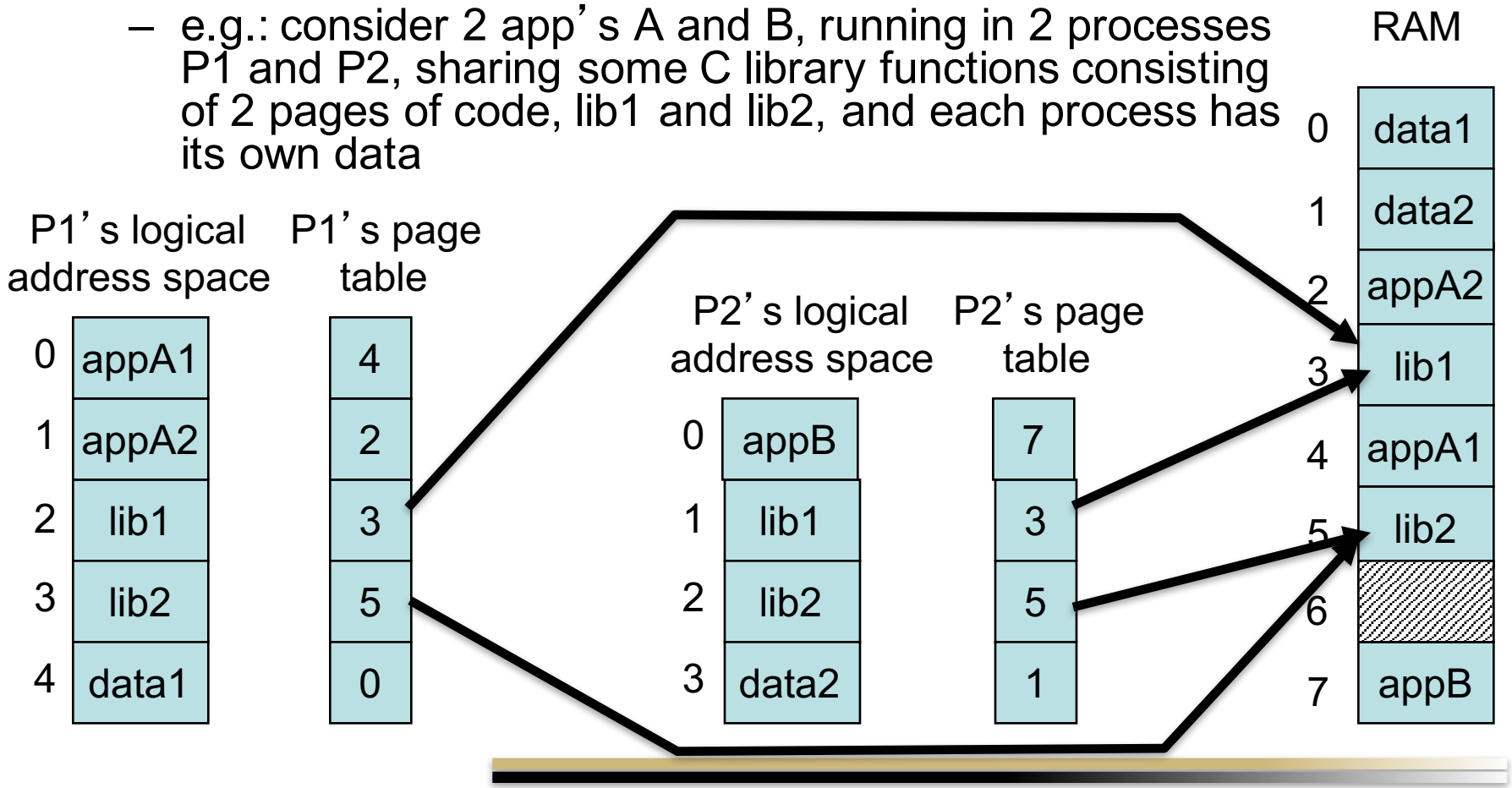


Recap: Paging with TLB and L1/L2 Caching



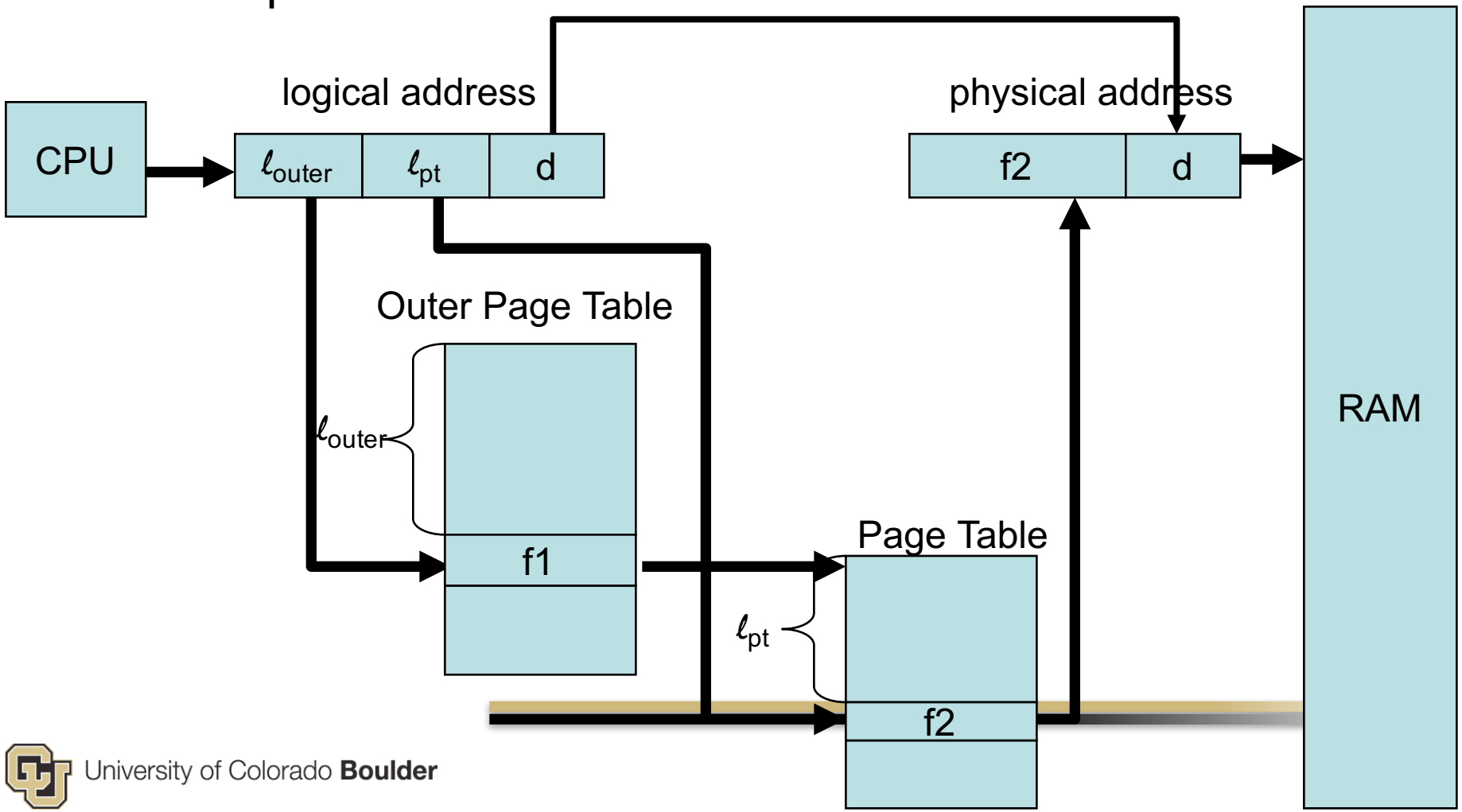
Recap: Shared Pages

- 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



Recap: Hierarchical Paging

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



Sparse Page Tables

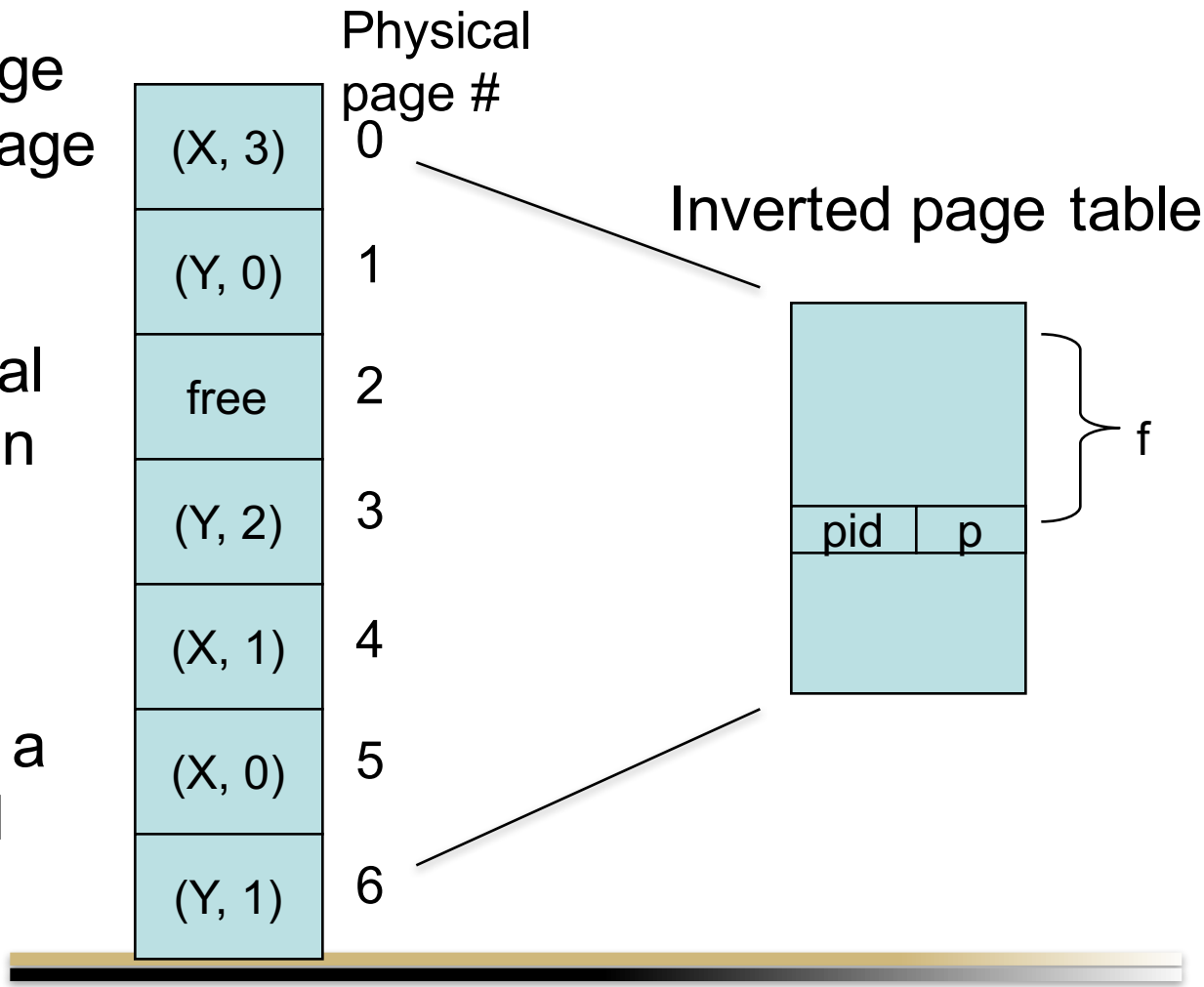
- Size of page tables may be large – millions of entries
- Few entries in each page table may be filled in
  - e.g. stack may never grow large enough to occupy a large fraction of virtual memory
  - e.g. heap may never grow large enough to occupy a large fraction of virtual memory
- Hence, there can be much “wasted” space in page tables
  - Still have to allocate space in memory for these unused page of the page table

Inverted Page Table

- Solution: use inverted page table (IPT)
  - Have only one page table for all of memory, rather than one for each process
    - this saves on memory
  - Each entry in the inverted page table lists which process owns a physical frame f, and what logical page # p is stored in that physical frame
    - each entry in the IPT lists a process id pid and a logical page p, namely IPT[f] = <pid, p>
    - thus the index into an inverted page table is the physical frame # f
    - compare to a page table, whose index is the logical page # p

Inverted Page Table

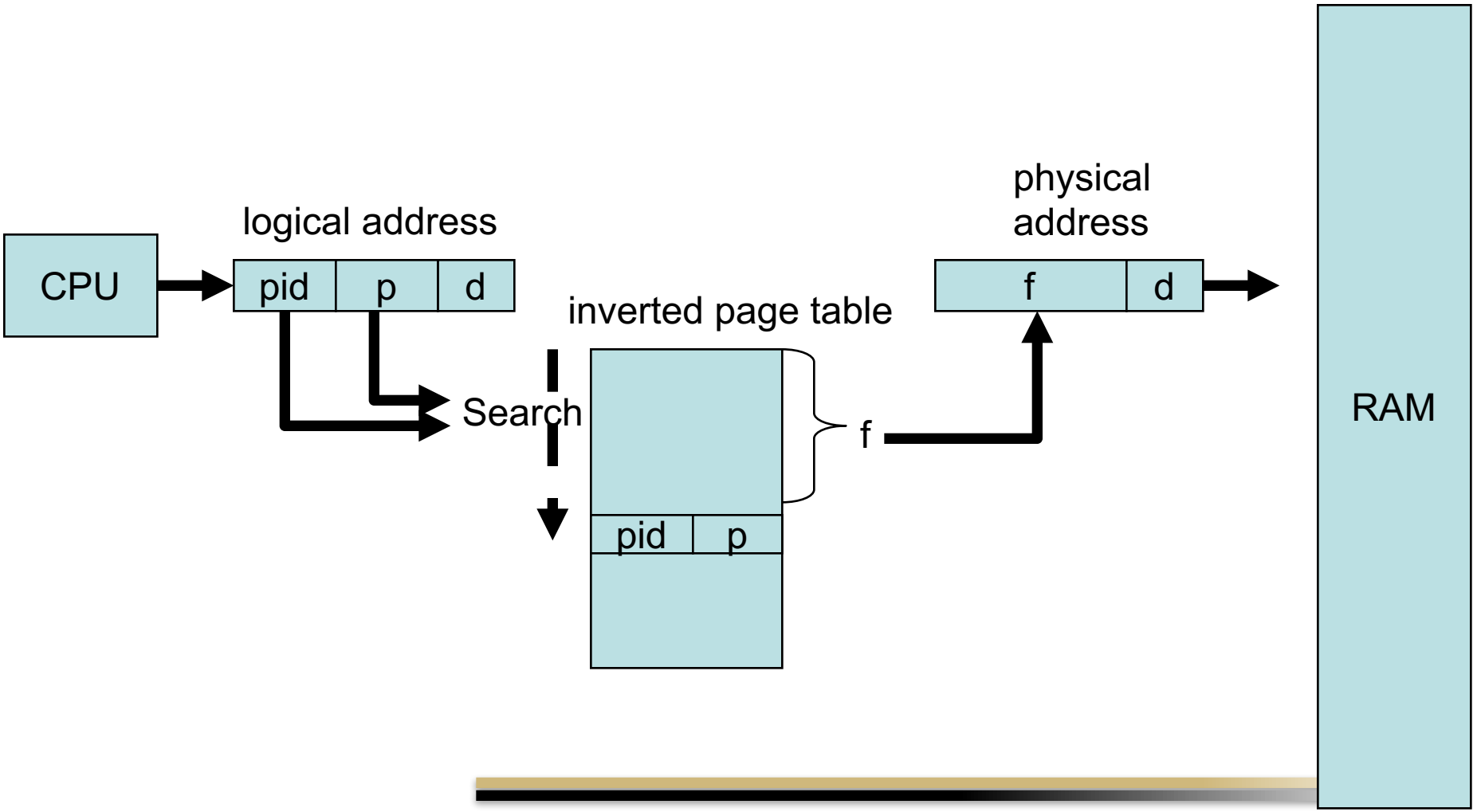
- Each physical page stores a logical page owned by some process
- Process X's logical page 3 is stored in physical page 0:
  - e.g. (X,3) in 0
- Collect all such assignments into a separate inverted page table array



Inverted Page Table

- Still want to use an inverted page table to map a logical page to a physical frame
  - the IPT stores an array of <pid, p> pairs,
  - given a pid and logical page p, *search* through IPT to find an entry that matches
    - this can be slow - a hash table can speed this up (later slides)
- another drawback: it's hard to implement shared memory pages with IPTs
  - since only one process owns each physical frame,
  - whereas in shared memory multiple processes can use the same code in a physical frame

Inverted Page Table



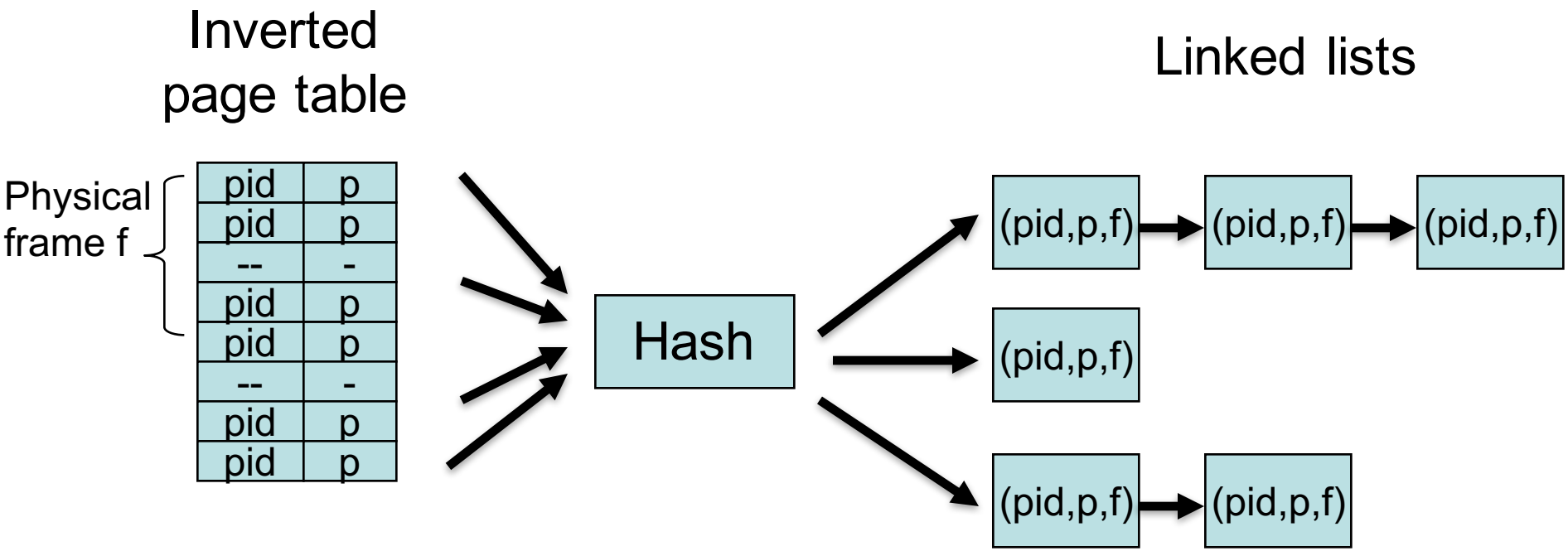
Hashed Page Table

- A hash table allows you to search fewer entries in either an inverted page table, or a regular page table
- A hash function h is a many-to-one function
  - example: take the most significant byte of a 32-bit address, i.e. h(32-bit addr) = MSB
  - there may be more than one input value that maps onto the same output value, because it is many-to-one
    - if h takes the MS bit, then h(10) = 1 = h(11), so both 10 and 11 map to the same hashed value

Hashed Page Table

- for an IPT:
  - input values <pid, p> into h and return a hashed value h(pid,p) = H.
  - Then index into the hash table with value H.
  - Each entry of the hash table contains a linked list, in case there are other input values that map to the same hashed value H.
  - Search through this linked list.
    - This is faster than searching the whole IPT linearly from the beginning

Hashed Inverted Page Table

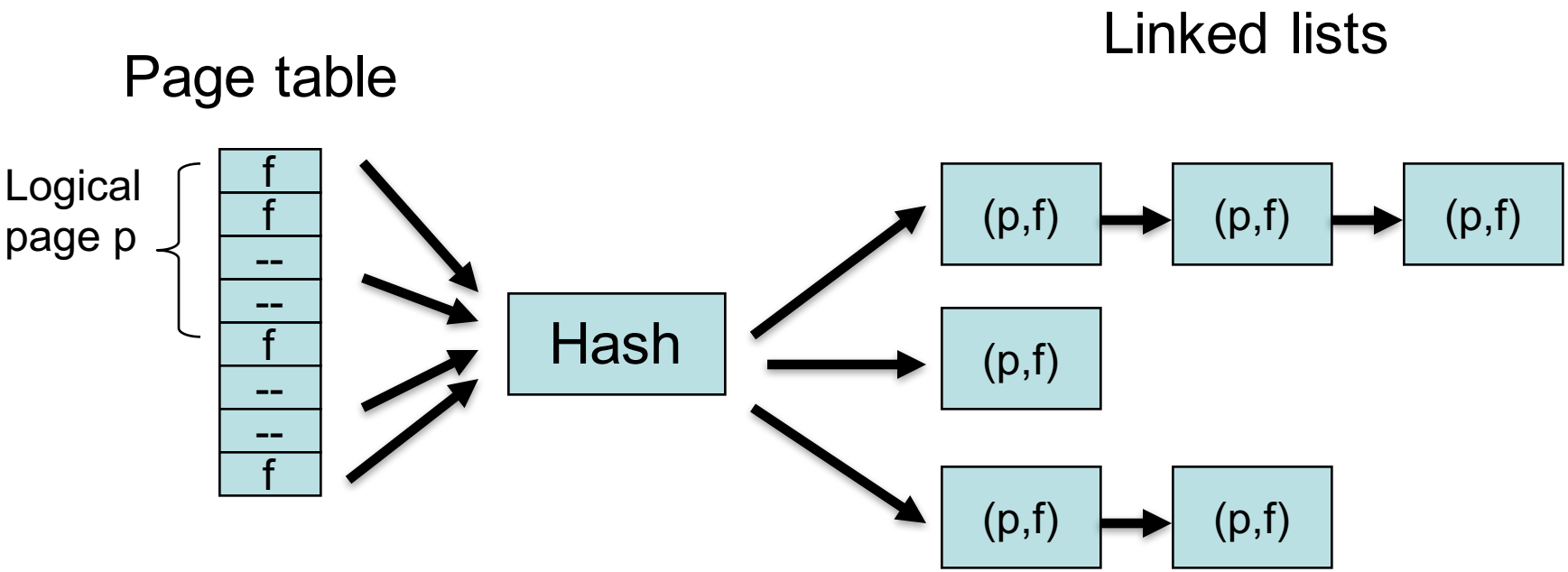


- Hash takes as input <pid,p>
- Note that empty entries of the inverted page table never need to be hashed

Hashed Page Table

- For a regular page table,
  - Hashing is useful for efficiently tracking assigned parts of very large *sparse* logical address spaces
  - example: typical heap and stack may be quite sparse with many empty logical addresses
  - Apply hashing with linked lists to track all logical addresses
  - Only assigned addresses will invoke the hash and will tracked as indexes in a hash table
    - unassigned pages never invoke hash, so no entries to track!
    - hash table, even with a linked list per entry to search, is seen as smaller and faster

Hashed Page Table



- Hash takes as input the logical page # p
- Note that empty entries of the page table never need to be hashed

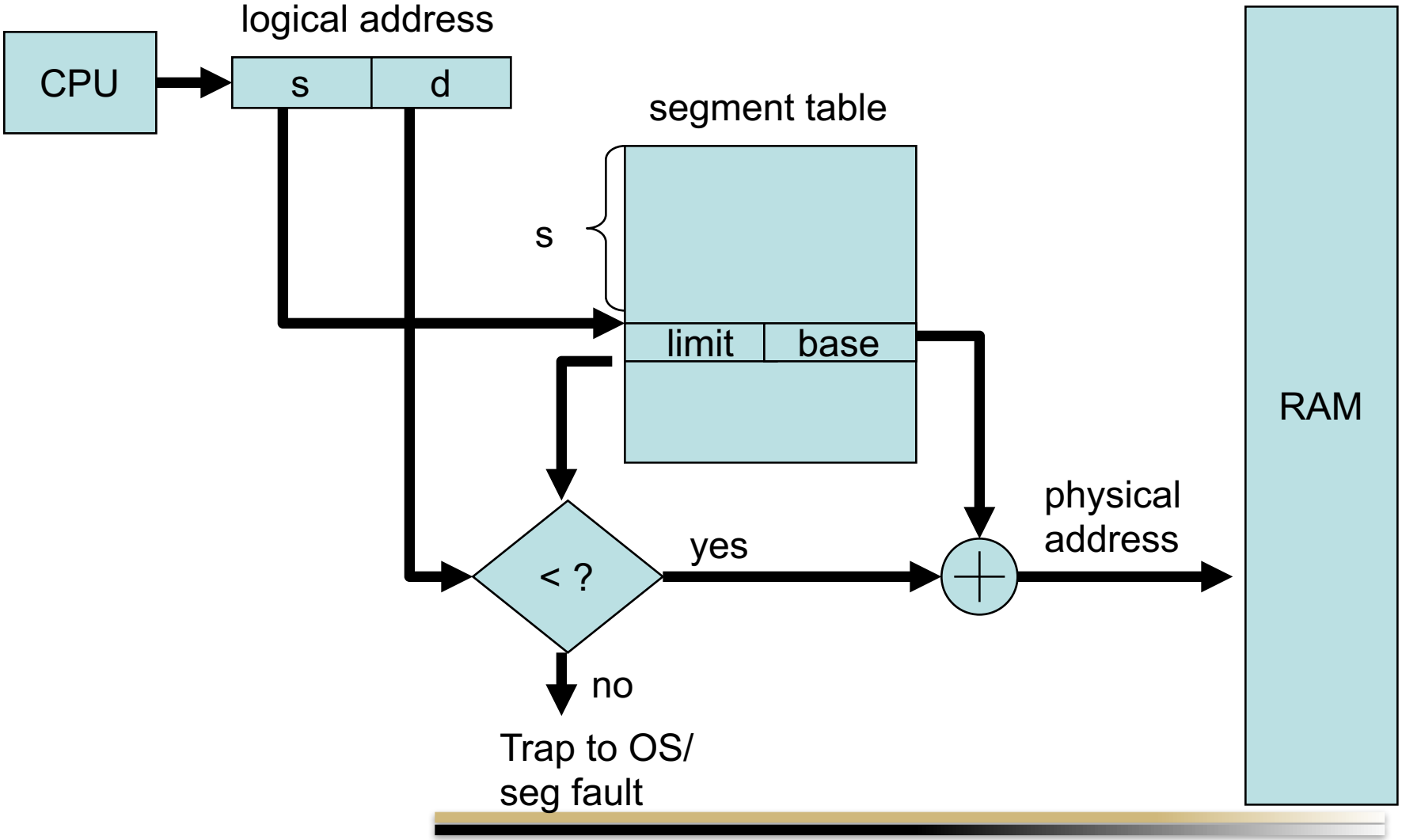
Segmentation

- An alternative solution for external fragmentation is to employ variably sized *segments*
  - Contrast to fixed-sized pages
  - Divide a process into variably sized segments that are organized according to some logical criteria
    - a process has code, data, stack, and heap - each of these could be a separate segment
    - a process could subdivide its code into functional segments
      - e.g. a Web server could divide its functional components into a networking segment, a database interface segment, and a dynamic page composition segment, etc.

Segmentation

- Each instruction in a segment has a logical address = <segment #, offset>
- Need a segment table to keep track of where each segment is mapped in main memory
- Segments are variably sized
  - Each entry of the segment table needs a base and limit field to place in the base and limit registers

Segmentation



Segmentation

- x86 Pentium supports pure segmentation and segmentation with paging
  - Segmentation with paging means that that each segment is divided into pages
- Linux on x86 Pentium uses segmentation sparingly
- Drawback of segmentation:
  - fragmentation of free space in RAM becomes even more complex and hard to manage