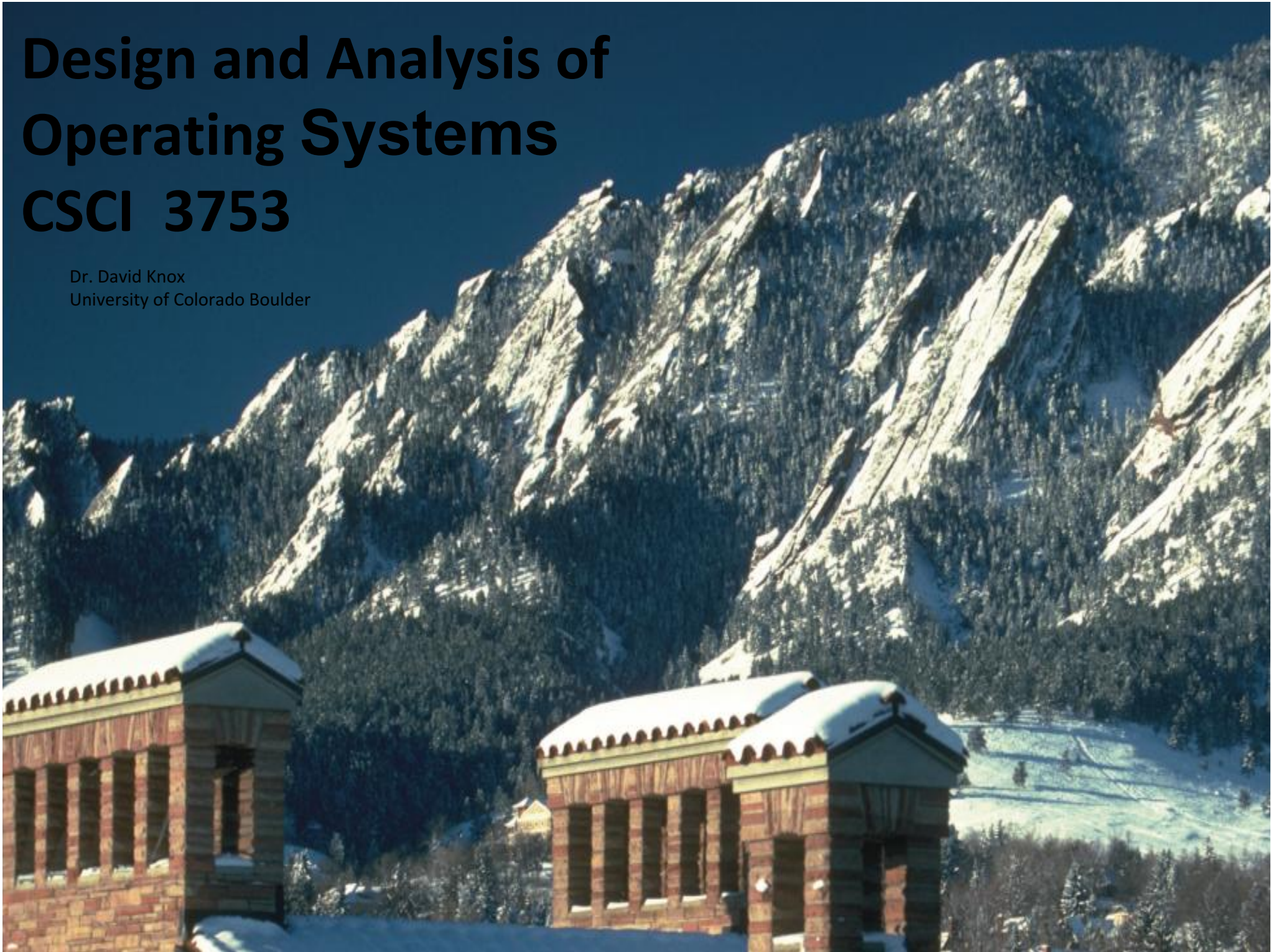


Design and Analysis of Operating Systems CSCI 3753

Dr. David Knox
University of Colorado Boulder





Department of Computer Science
UNIVERSITY OF COLORADO **BOULDER**



Design and Analysis of Operating Systems CSCI 3753

Memory Management

Dr. David Knox
University of
Colorado Boulder

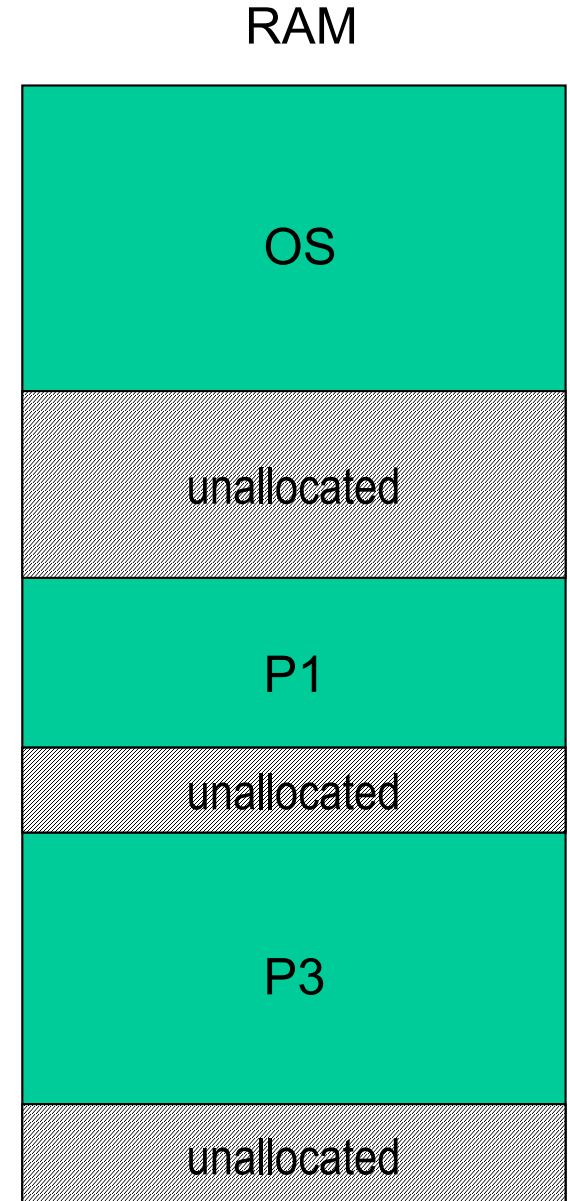
Material adapted from: Operating Systems: A Modern Perspective : Copyright © 2004 Pearson Education, Inc.

Memory Management

Paging

Fragmentation

- over time, repeated allocation and deallocation will cause many small chunks of non-contiguous unallocated memory to form between allocated processes in memory
- resulting in *external fragmentation*
- OS must swap out current processes to create a large enough contiguous unallocated memory
- could use de-fragmentation, but it is costly to move memory



Paging

- A better solution to external fragmentation is to divide the logical address space into fixed-size *pages*

Logical Address Space

page 0
page 1
page 2
page 3
page 4

- We can also break main memory into fixed-sized *frames*
- Each *page* can be located in any *frame*
- *The frames used by a process don't have to be contiguous in memory*
- Need a method to map from logical address to physical address

RAM	
frame #	
0	page 4
1	page 0
2	unallocated
3	page 2
4	page 1
5	unallocated
6	unallocated
7	page 3

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

logical memory

0	5
1	6
2	1
3	2

page table

0	
4	i j k l
8	m n o p
12	
16	
20	a b c d
24	e f g h
28	

physical memory

Paging

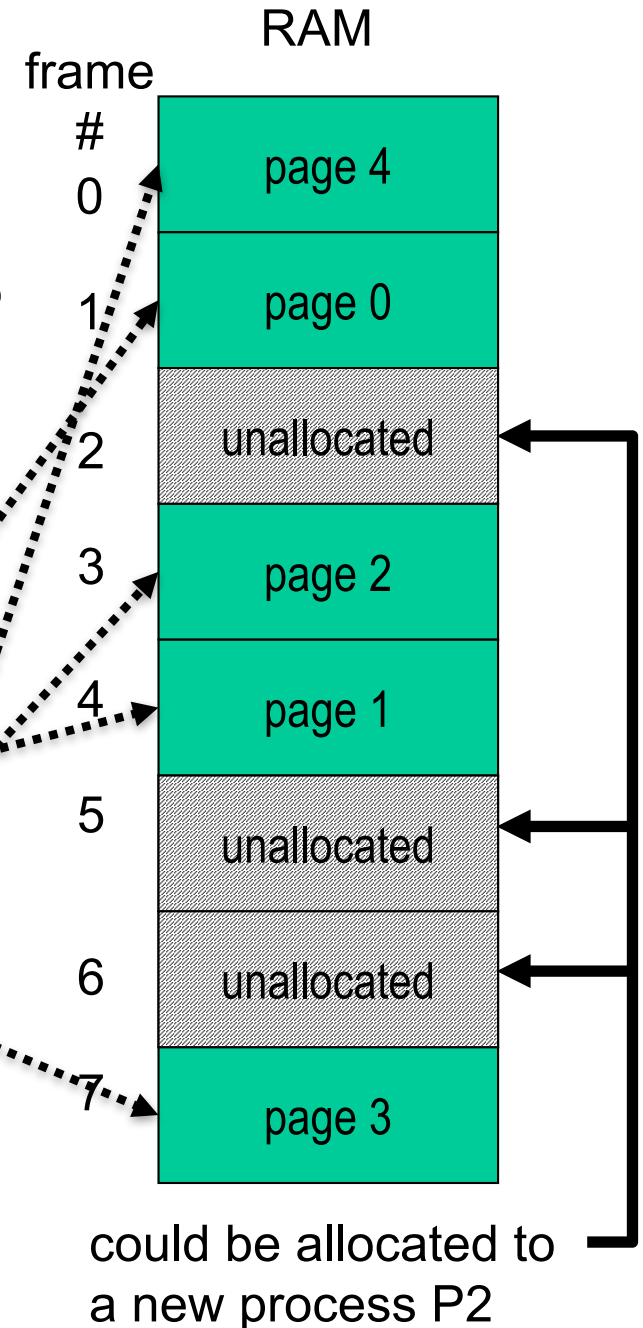
- OS maintains a page table for each process
- Given a logical address, MMU finds its logical page, then looks up physical frame in *page table*

Logical Address Space

page 0
page 1
page 2
page 3
page 4

Page Table

Logical page	Physical frame
0	1
1	4
2	3
3	7
4	0



Paging

Logical Address
Space

page 0
page 1
page 2
page 3
page 4

- **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**
 - Page Table Size: a 4 GB 32-bit address space with 4 KB/page (2^{12}) $\Rightarrow 2^{32}/2^{12} = 1$ million entries in page table
 - Your page table would need at least 20 bits per table entry (~ 3 MB per process)
 - Address Limitations:
 - 2^{32} bytes = 4 GB of memory
 - 2^{44} bytes = 16 TB (32 bit table entry * 12 bits per page)
 - 2^{64} bytes = 16 Exabytes
- There is support for huge/large pages ~ 4 MB each

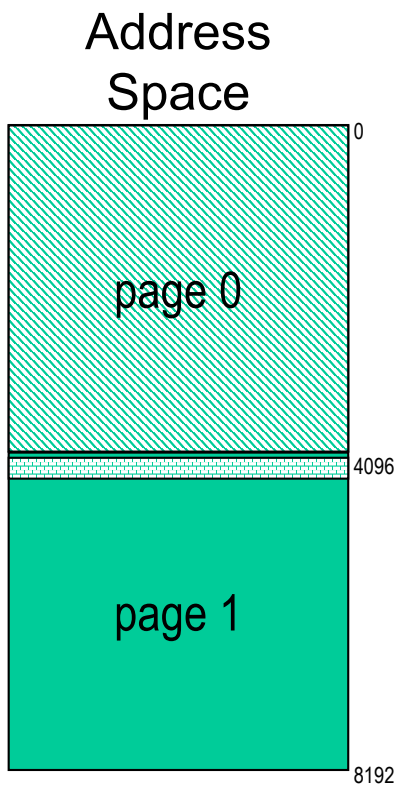
Paging

- No external fragmentation

- But we do get some *internal fragmentation*

- example: suppose my process is size 4100 B, and each page size is 4 KB (4096 B)
 - then I have to allocate two pages = 8 KB,
 - 4092 B of 2nd page is wasted due to fragmentation that is internal to a page

- OS has to maintain a frame table/pool that keeps track of what physical memory frames are free

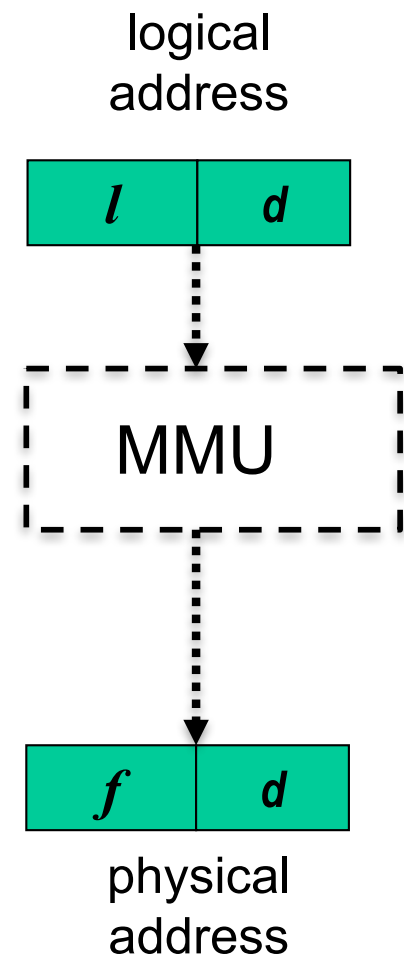


Paging

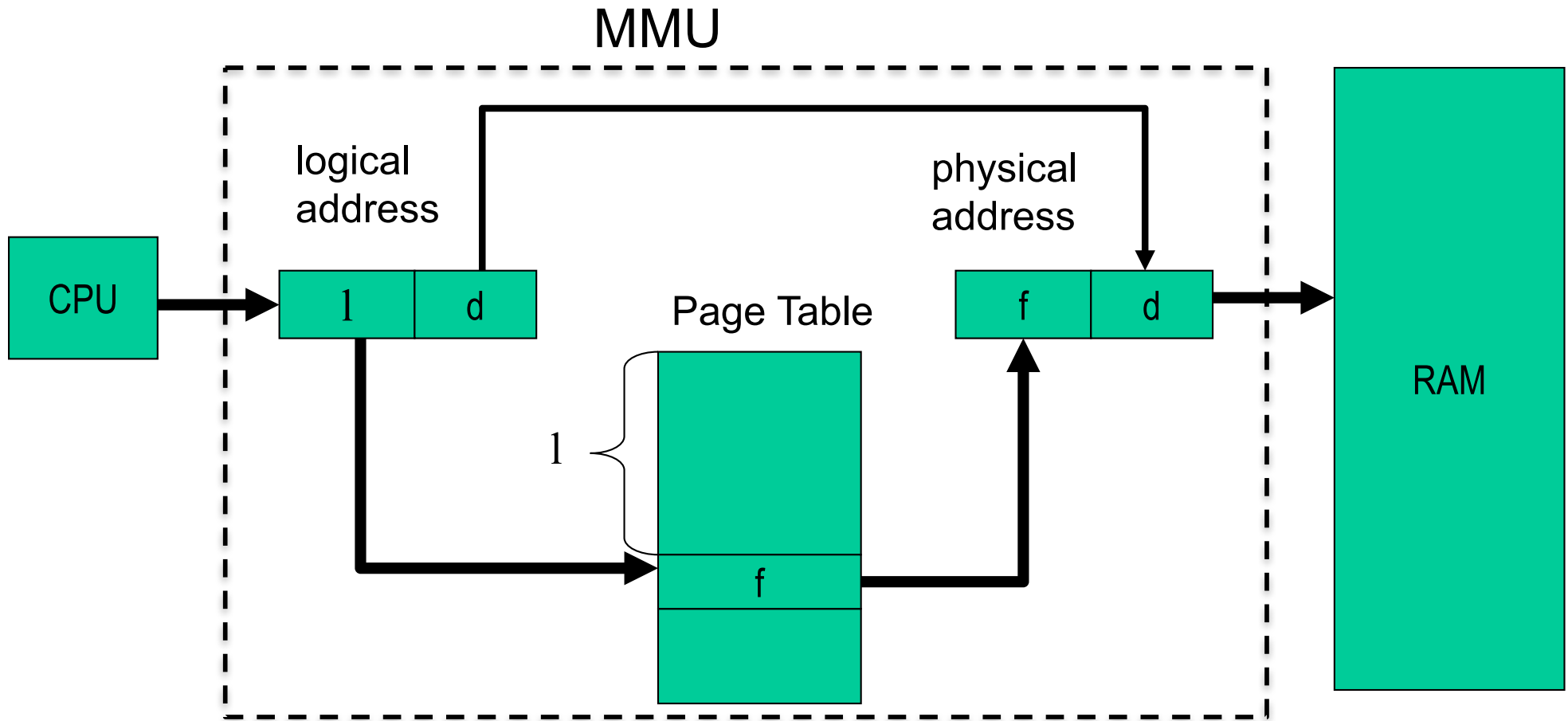
- Conceptually, every logical/virtual address can now be divided into two parts:

logical page and page offset

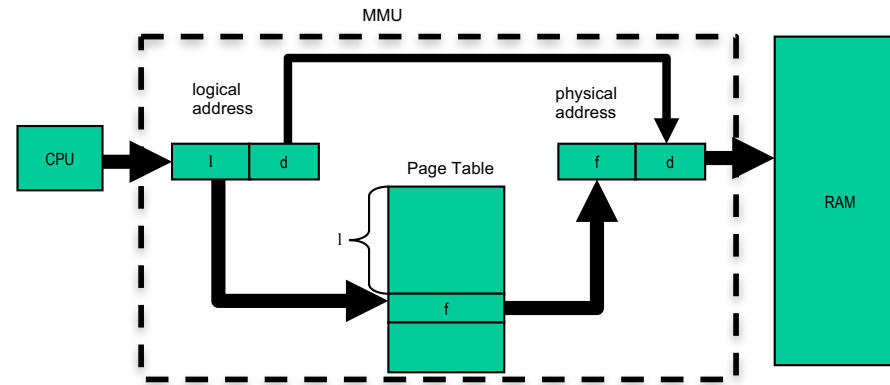
- most significant bits = logical page # l ,
 - Same as the virtual address / page size
 - Logical page is used to index into *page table* to retrieve the corresponding physical frame f
- least significant bits of the address represent the page offset d



Paging

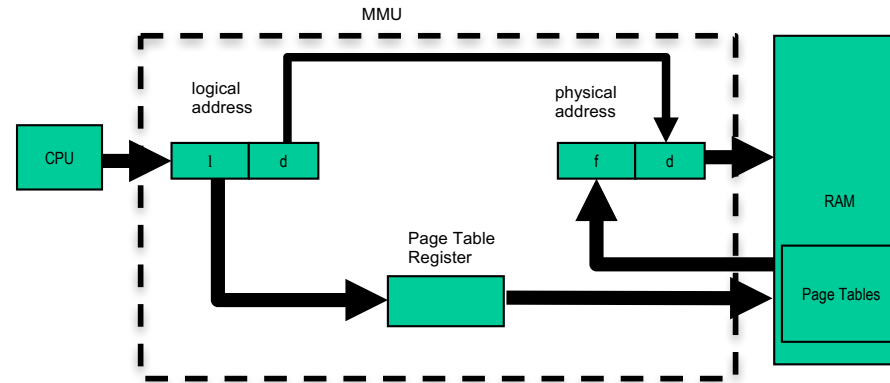


Implementing a page table



- 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

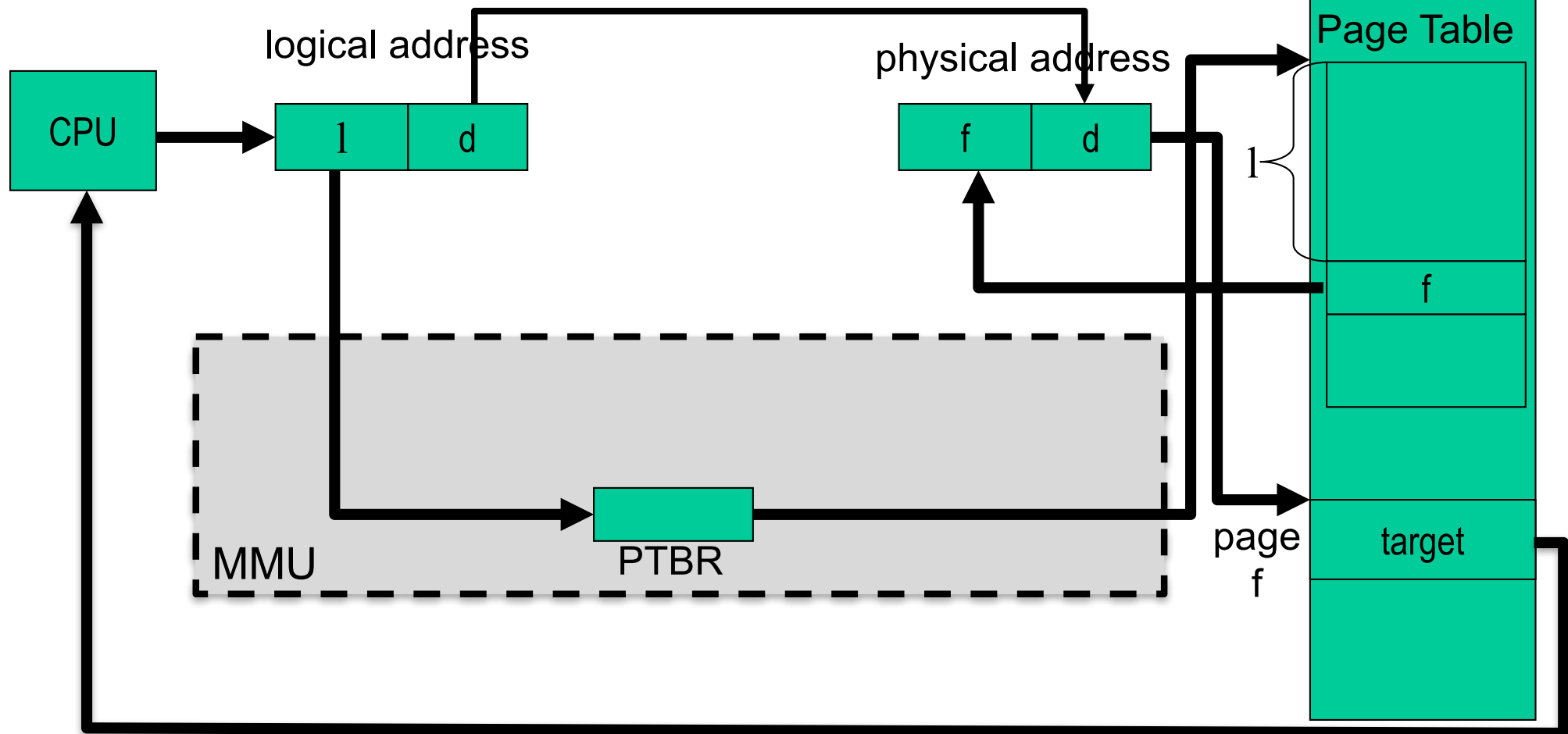
Implementing a page table



- 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 with 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

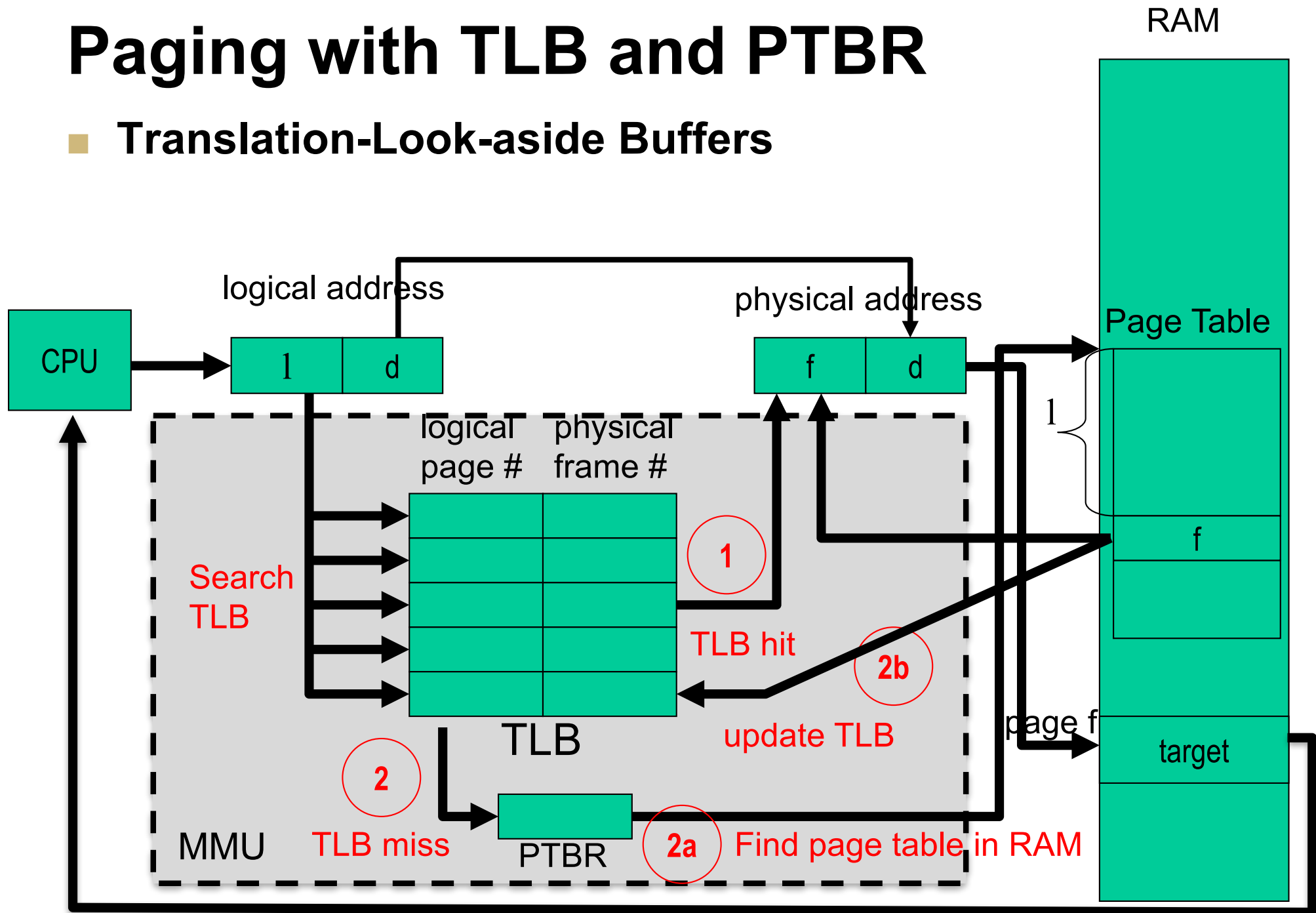


Implementing a page table

- 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 with TLB and PTBR

■ Translation-Look-aside Buffers



Return fetched code/data @ offset **d** inside page **f**

Paging and TLB Caching

- **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

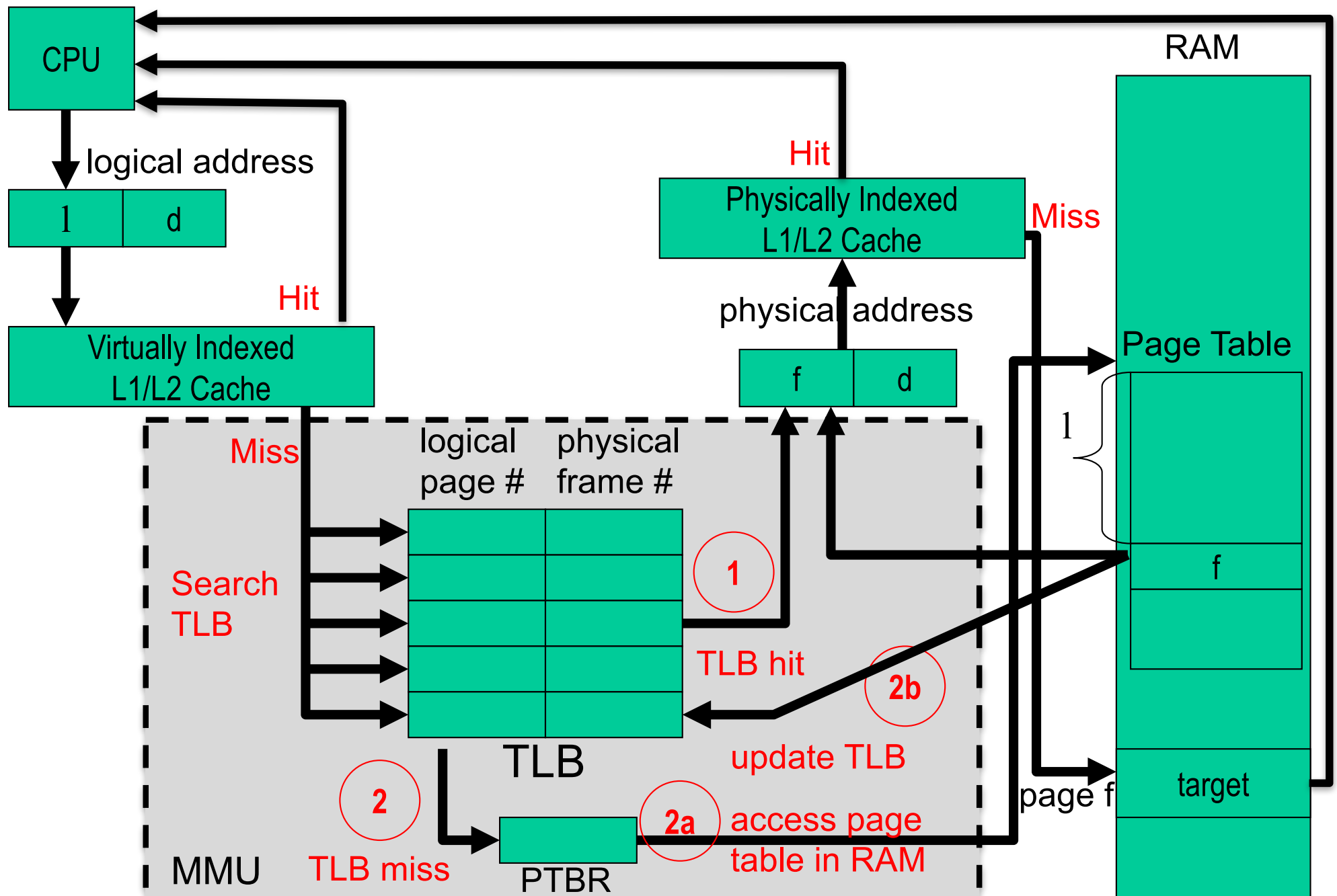
Paging and TLB Caching

- 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).

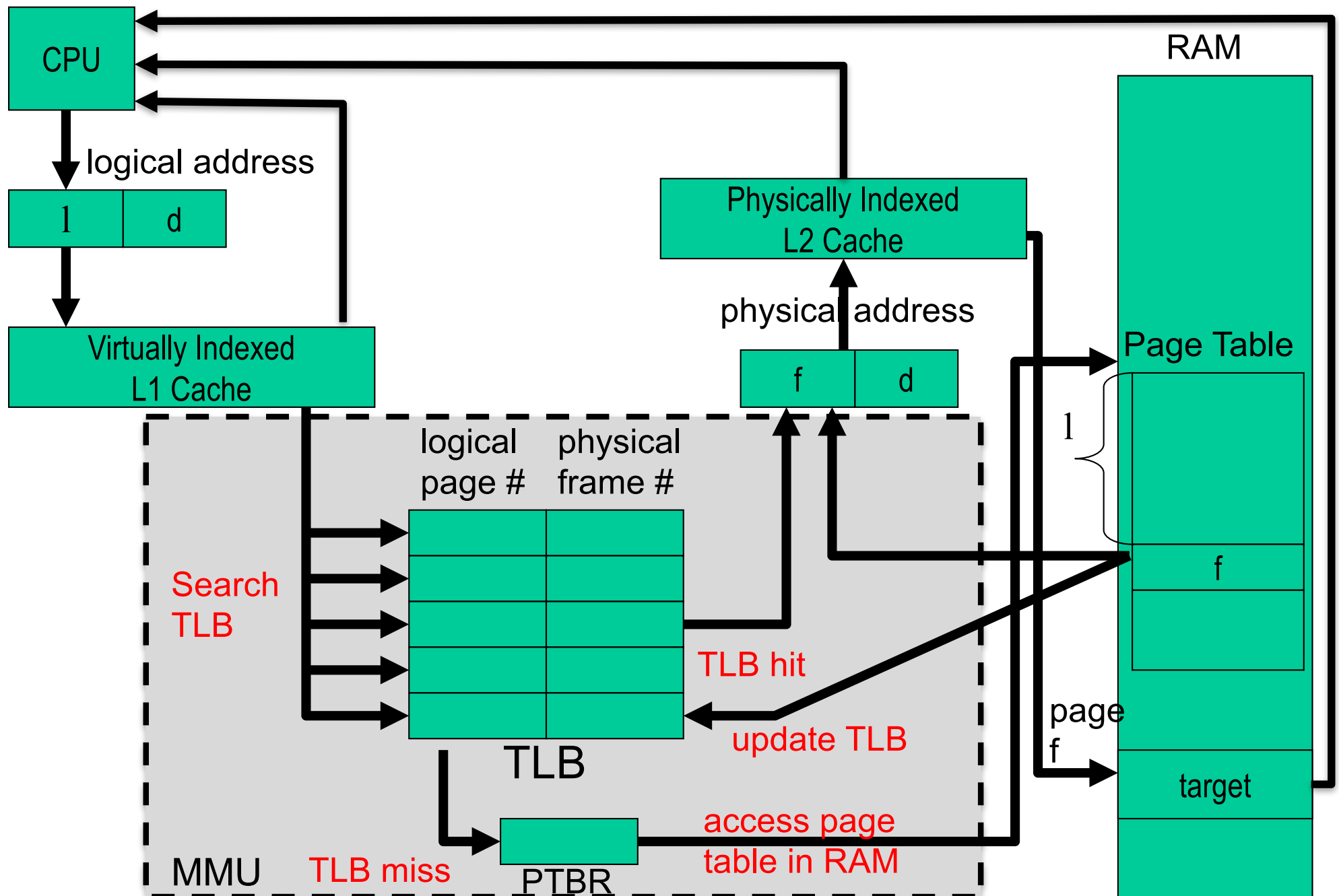
Paging and L1/L2 Caching

- **Some solutions to the homonym problem:**
 - Flush the cache on each context switch
 - process gets entire cache to itself, but have to rebuild cache
 - 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
 - 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 $[0, \text{MAX}]$

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.

■ Paging with TLB and L1/L2 Caching



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

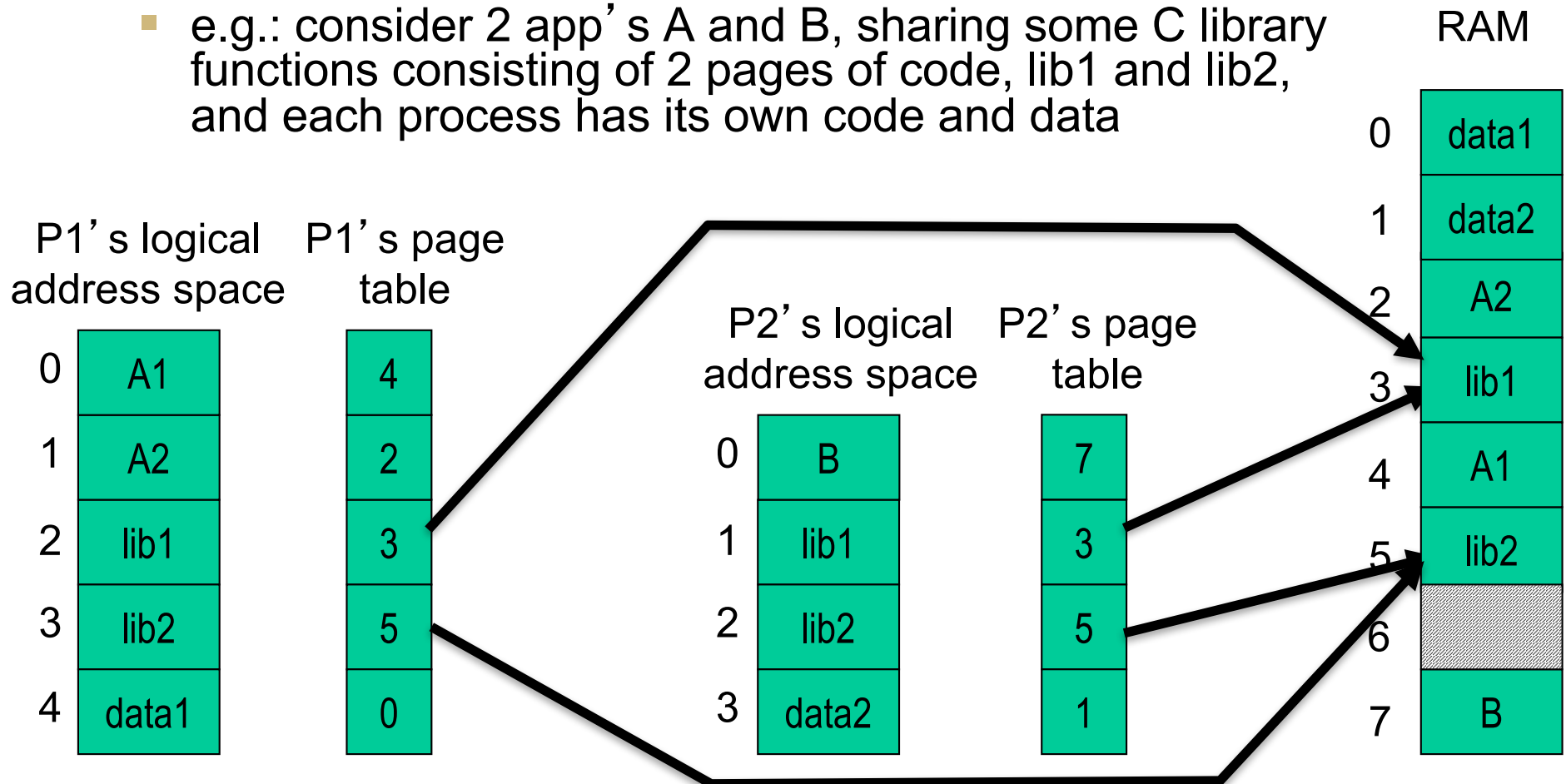
- **Sharing code:**

- 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

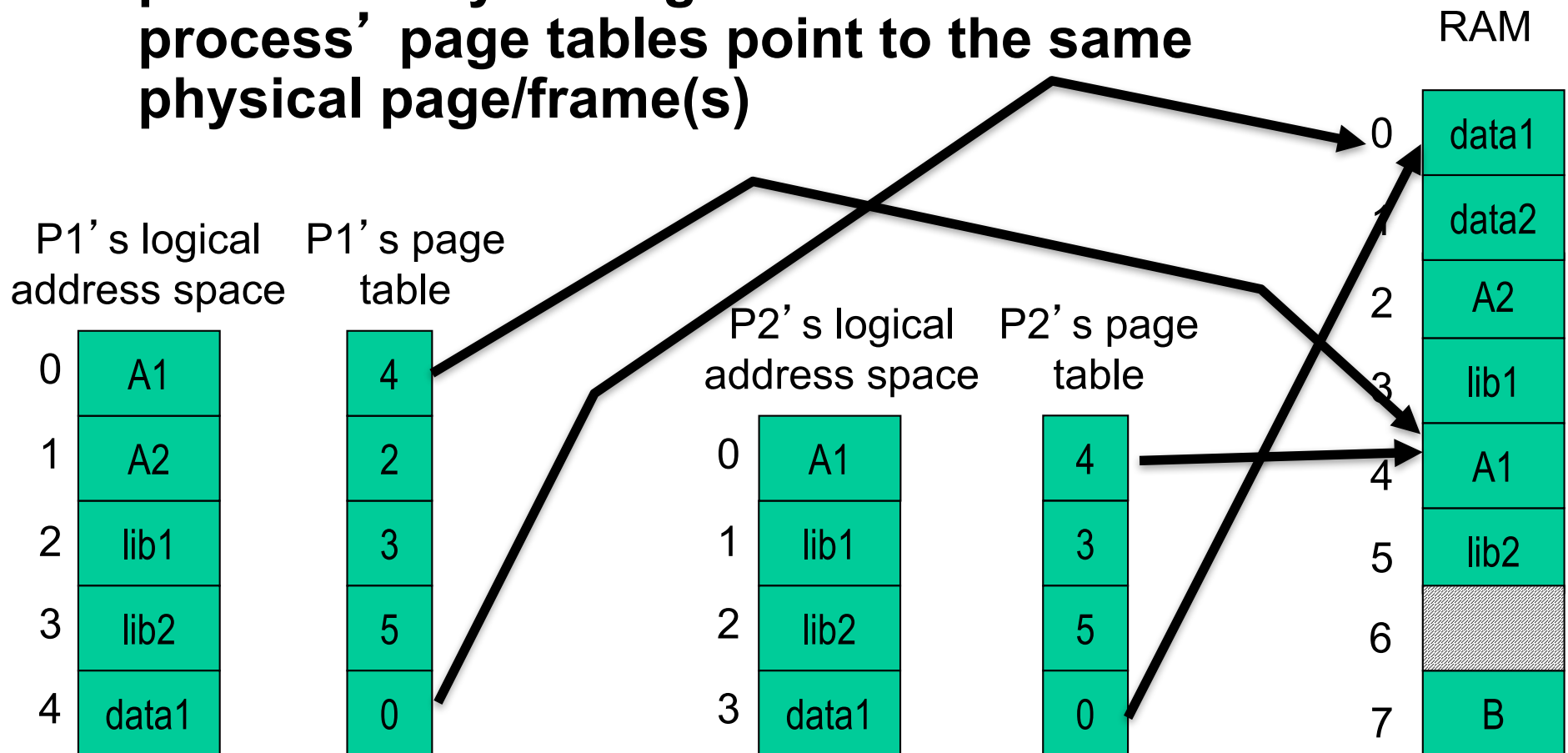
- Page tables can point to the *same* memory frames

- e.g.: consider 2 app's A and B, sharing some C library functions consisting of 2 pages of code, lib1 and lib2, and each process has its own code and data



Shared Pages

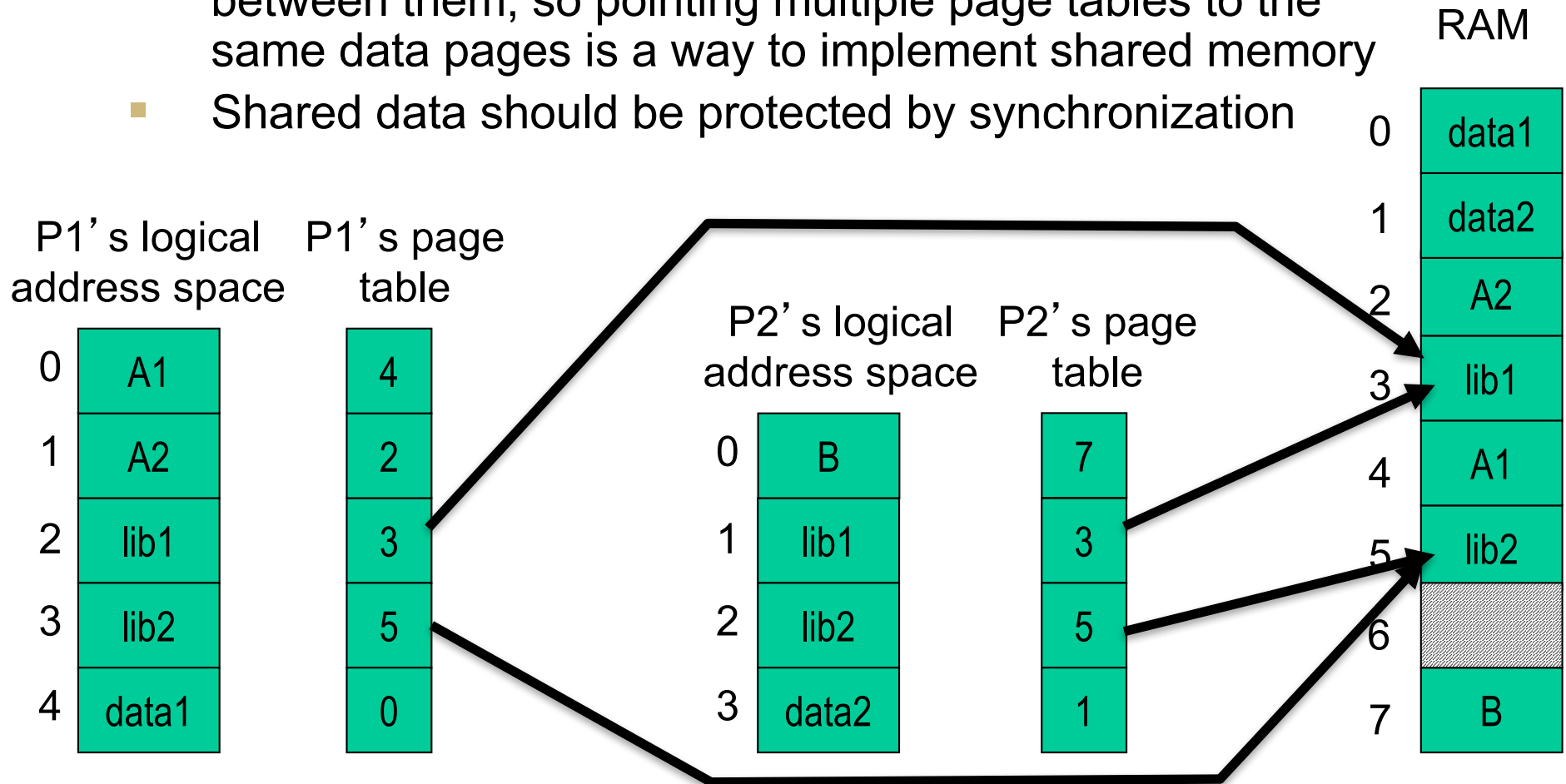
- Can share code & data pages between processes by having entries in different process' page tables point to the same physical page/frame(s)



Shared Pages

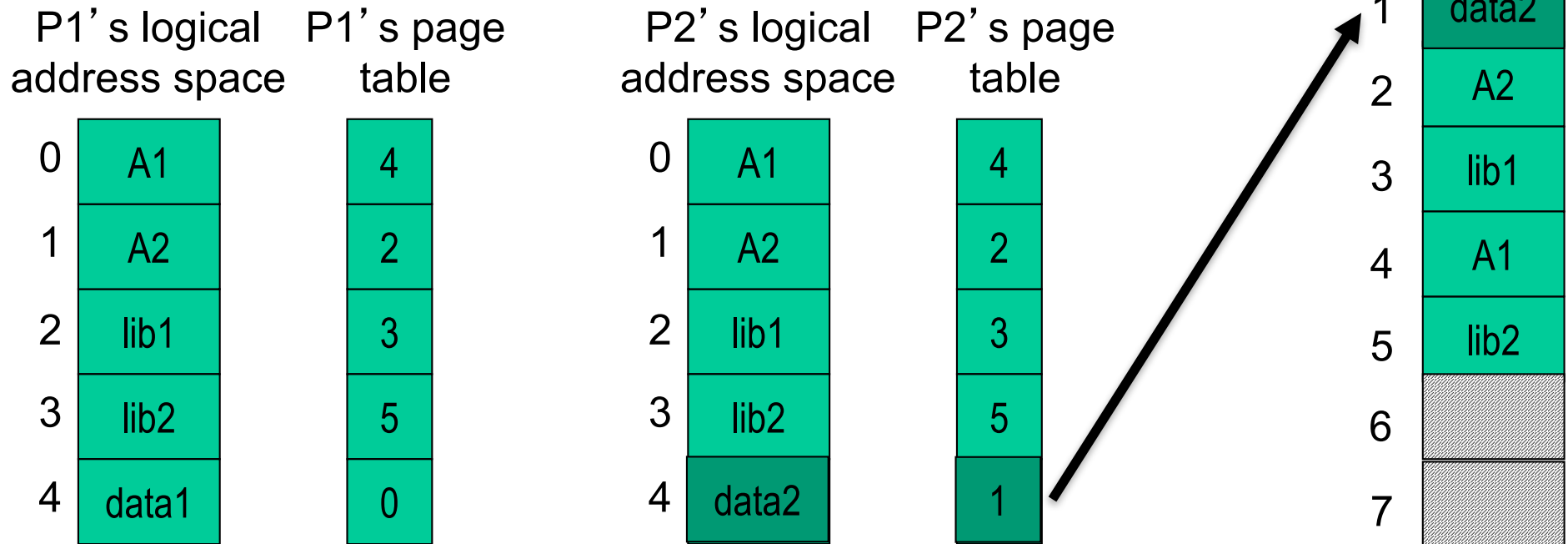
■ Sharing data:

- 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

- 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



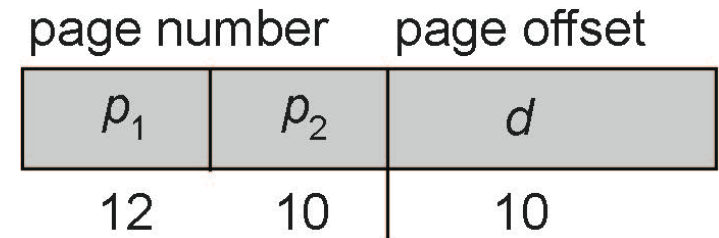
Hierarchical Paging

- **Problem with page tables: they can get very large**
 - example: 32-bit address space with 4 KB/page (2^{12}) implies that there are $2^{32}/2^{12} = 1$ million entries in page table per process. if each entry is 4 bytes that is 4 MB.
 - it's hard to find contiguous allocation of at least 4 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

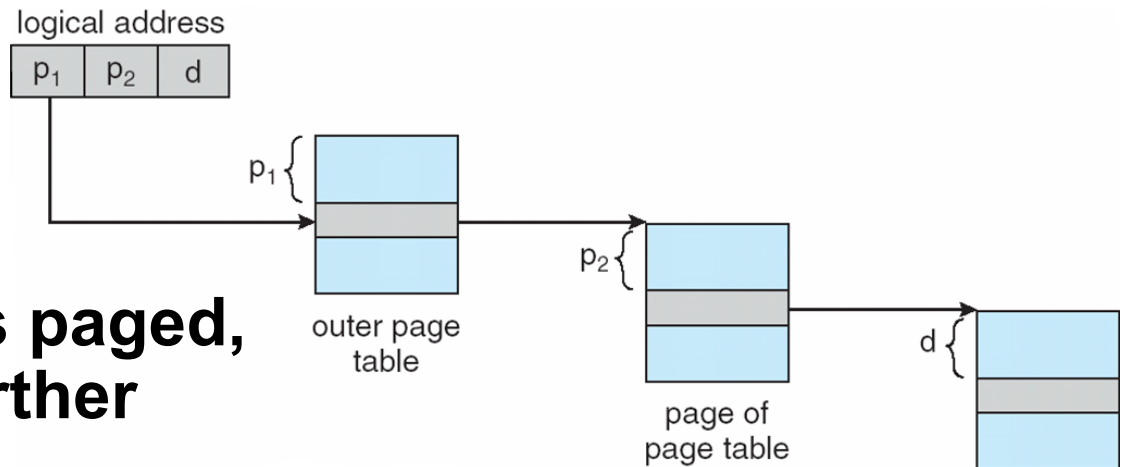
- A logical address (on 32-bit machine with 1K page size) is divided into:

- a page number consisting of 22 bits
- a page offset consisting of 10 bits



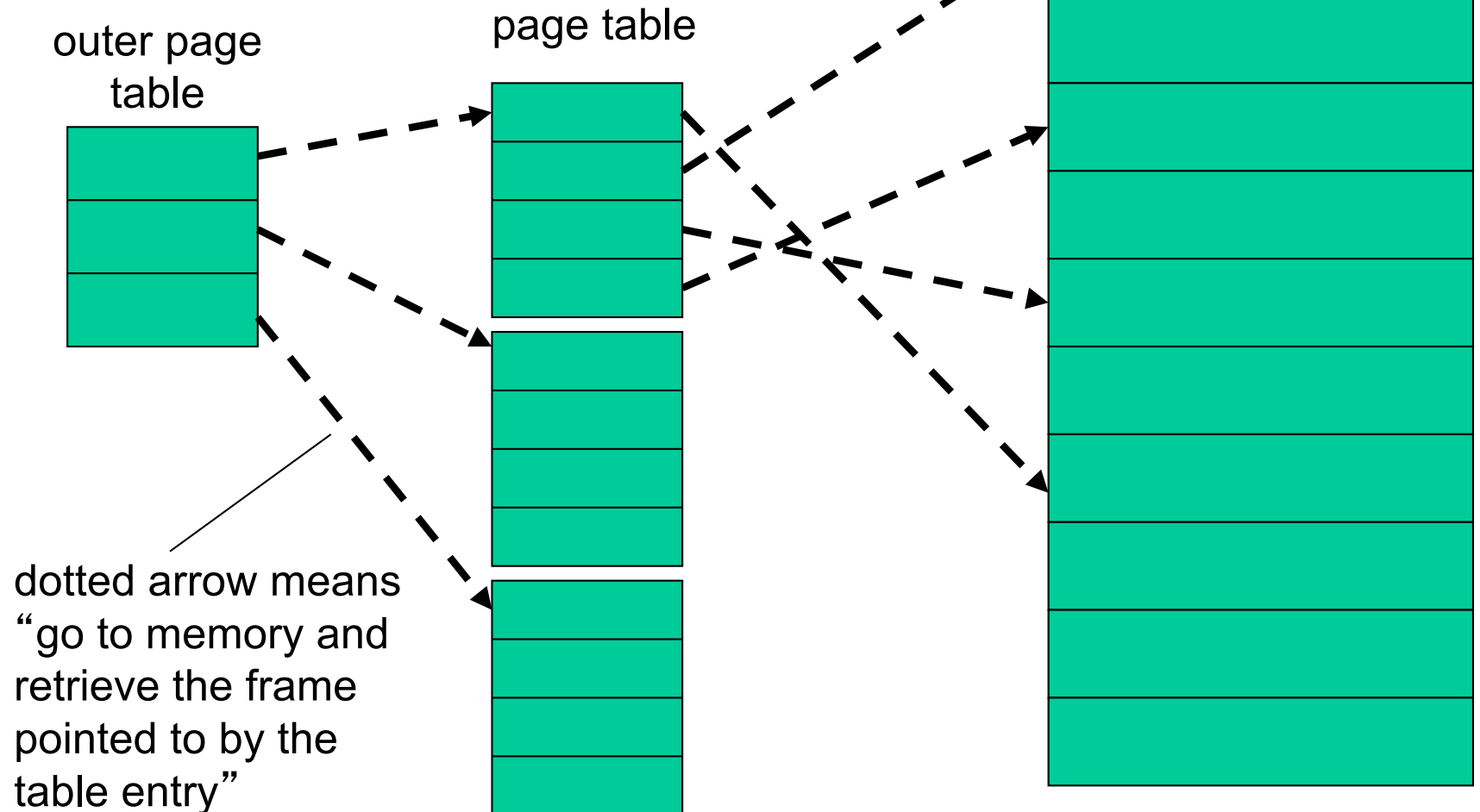
- Since the page table is paged, the page number is further divided into:

- a 12-bit page table number (outer page)
- a 10-bit page table offset



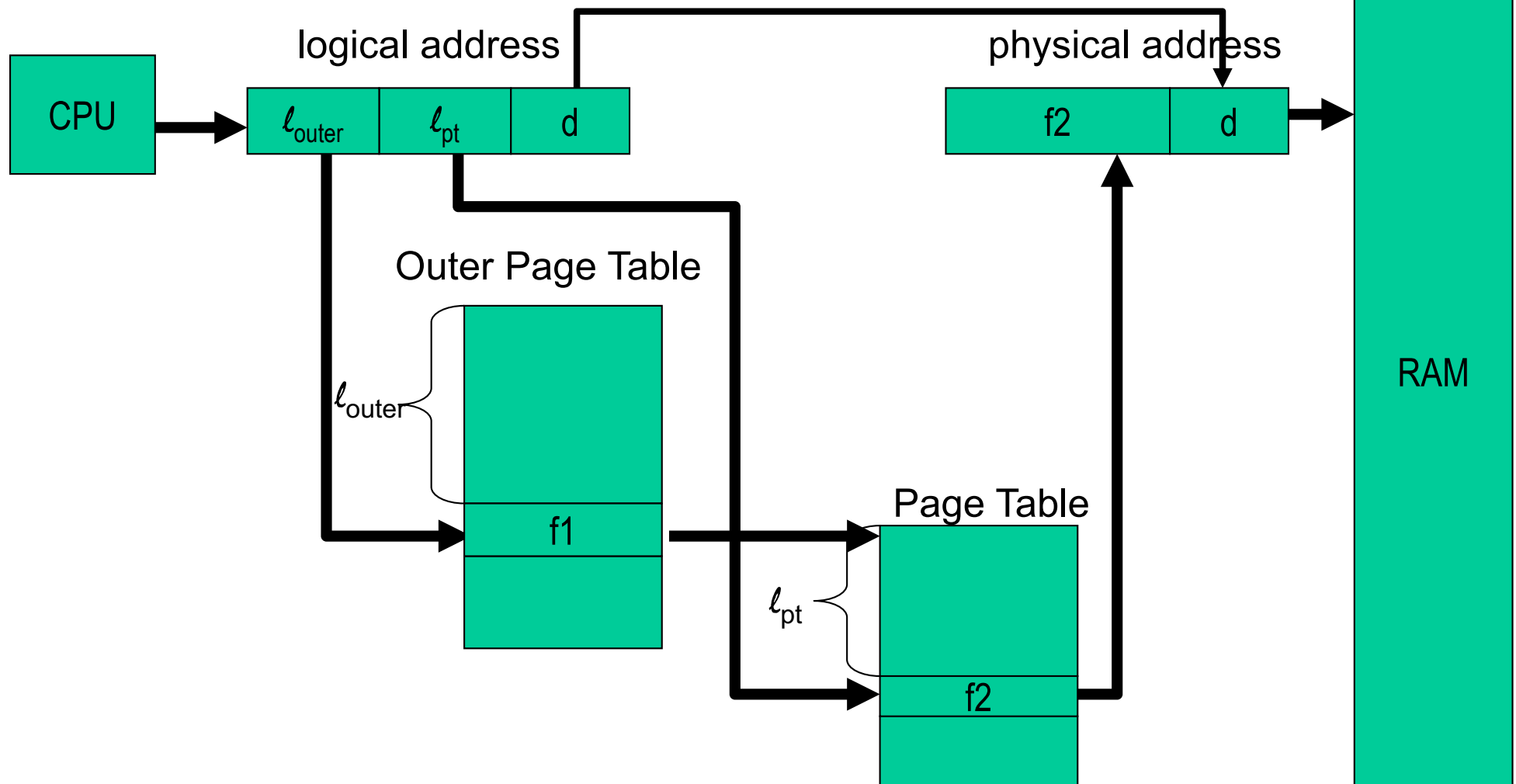
Hierarchical Paging

- **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



Department of Computer Science
UNIVERSITY OF COLORADO **BOULDER**



Design and Analysis of Operating Systems CSCI 3753



Dr. David Knox
University of
Colorado Boulder

Material adapted from: Operating Systems: A Modern Perspective : Copyright © 2004 Pearson Education, Inc.