Memory Management and Paging

CSCI 3753 Operating Systems

Spring 2005

Prof. Rick Han

Announcements

- PA #2 due Friday March 18 11:55 pm - note extension of a day
- Read chapters 11 and 12

From last time...

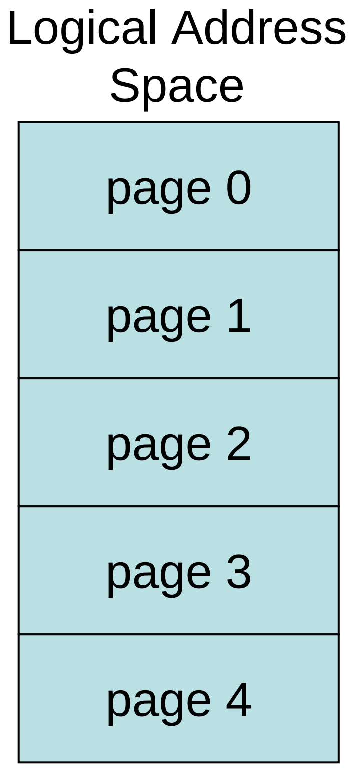
- Memory hierarchy
- Memory management unit (MMU)
  - relocates logical addresses to physical addresses using base register
  - checks for out of bounds memory references using limit register
- Address binding at run time
- Swapping to/from backing store / disk
  - fragmentation of main memory
    - first fit, best fit, worst fit

Paging

- One of the problems with fragmentation is finding a sufficiently large contiguous piece of unallocated memory to fit a process into
  - heavyweight solution is to compact memory
- Another solution to external fragmentation is to divide the logical address space into fixed-size ***pages***
  - each 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, so each process can now be scattered throughout memory and can be viewed as a collection of frames that are not necessarily contiguous
    - 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, i.e. to keep track of the mapping of each logical page to a physical memory frame

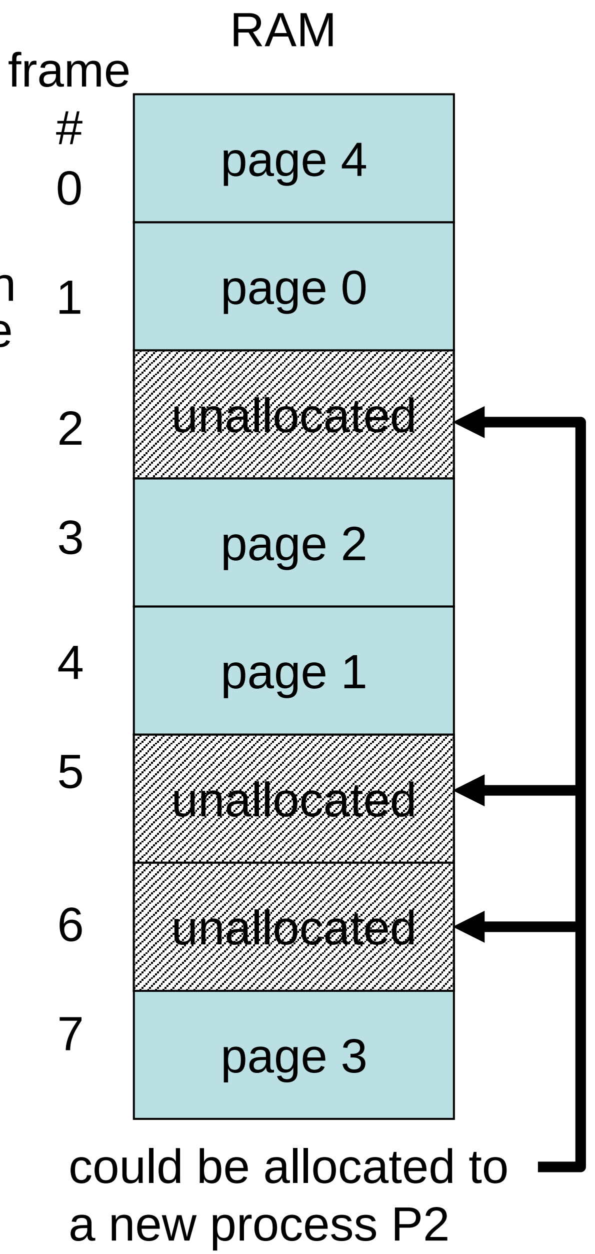
Paging

- A page table for each process is maintained by the OS
- Given a logical address, MMU will determine to which logical page it belongs, and then will consult the page table to find the physical frame in RAM to access



Page Table

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

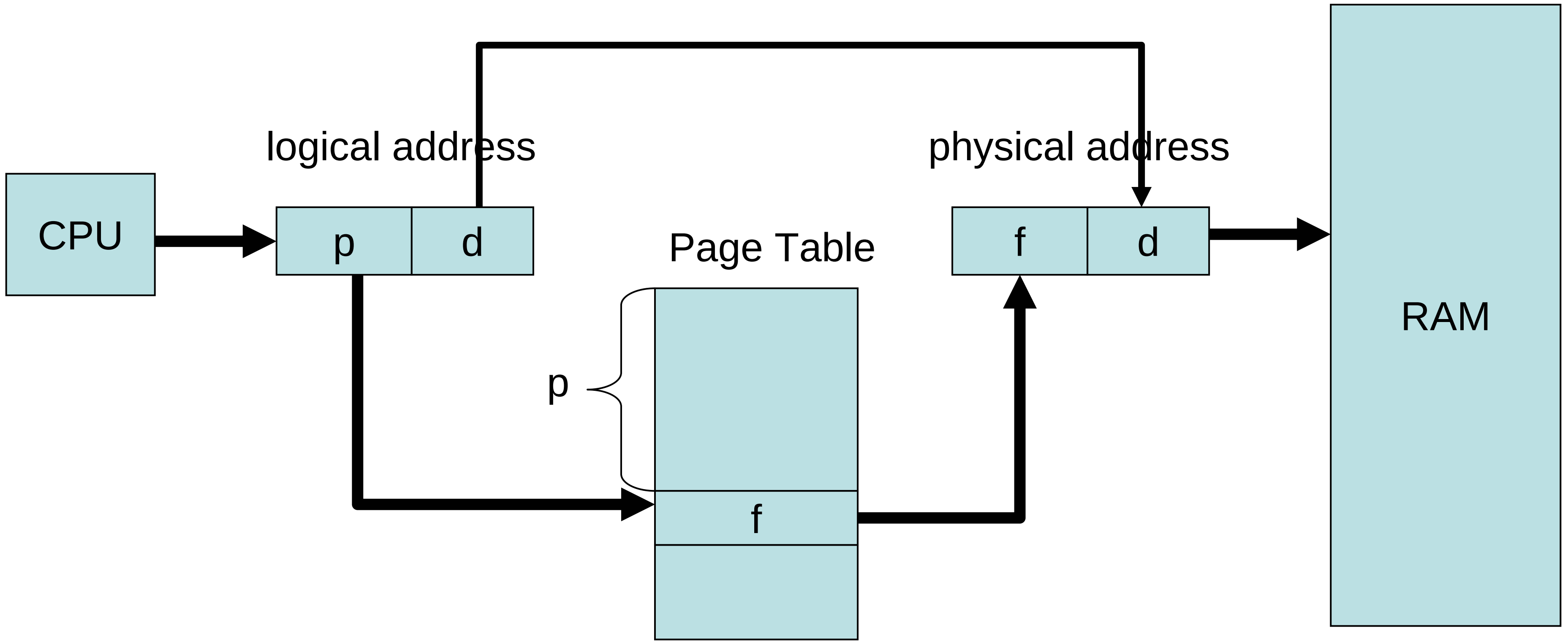


Paging

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

Paging

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



Paging

- Implementing a page table
  - option #1: use dedicated bank of hardware registers to store the page table
    - fast per-instruction translation
    - slow per context switch - entire page table has to be reloaded
    - limited by being too small - some page tables can be large, e.g. 1 million entries
  - option #2: store the page table in main memory and just 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 PTBR needs to be reloaded
    - slow per-instruction translation, because each instruction fetch requires two steps:
      - finding the page table in memory and indexing to the appropriate spot to retrieve the physical frame # f
      - retrieving the instruction from physical memory frame f

Paging

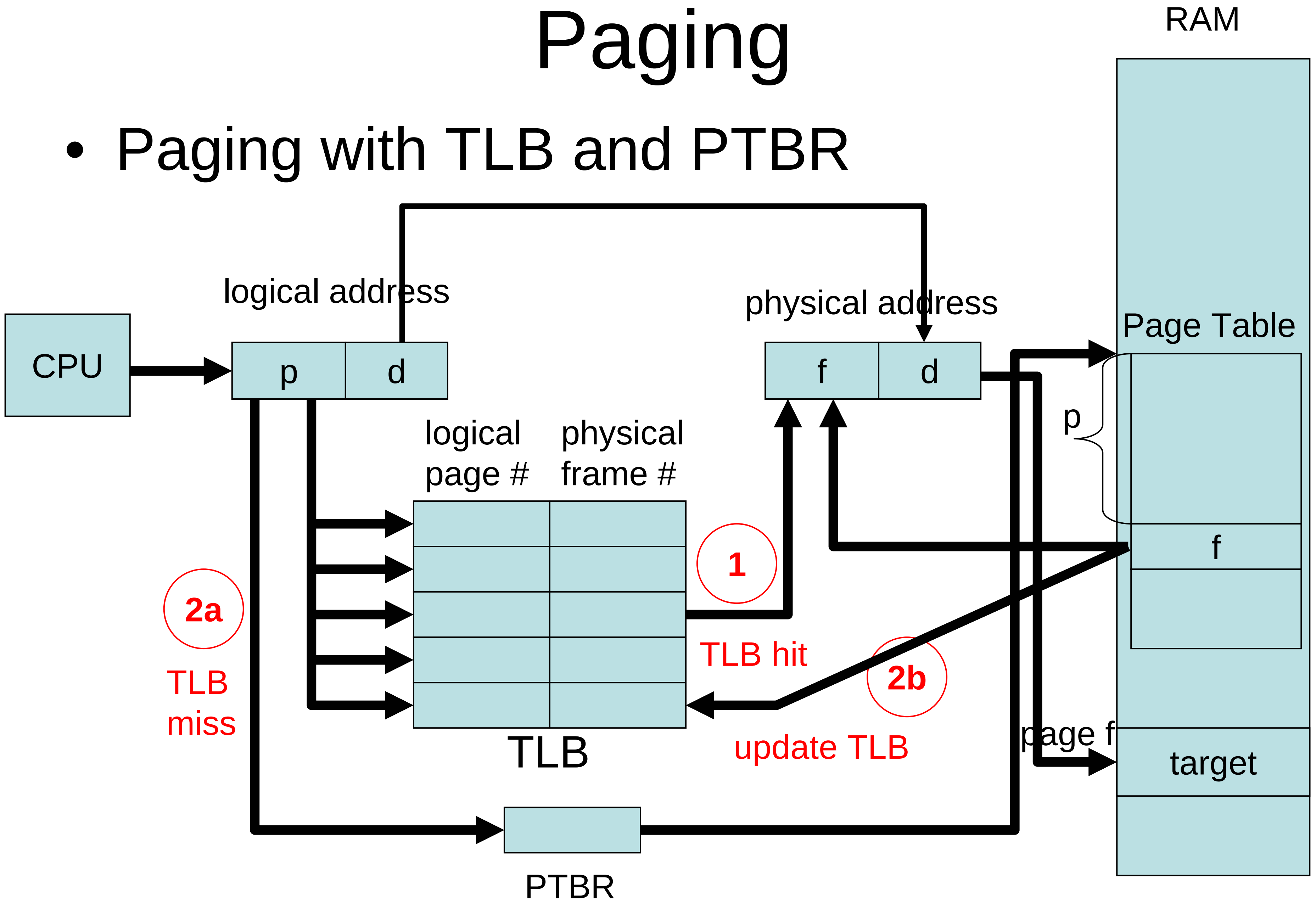
- Solution to option #2’s slow two-step memory access:
  - cache a subset of page table mappings/entries in a small set of CPU buffers called Translation-Look-aside Buffers (TLBs)
  - Several TLB caching policies:
    - cache the most popular or frequently referenced pages in TLB
    - cache the most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

Paging

- MMU in CPU first looks in TLB's to find a match for a given logical address
  - 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
  - if no match found, then
    1. 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
    2. 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
    - This is called a *TLB miss*
- Goal is to maximize TLB hits and minimize TLB misses

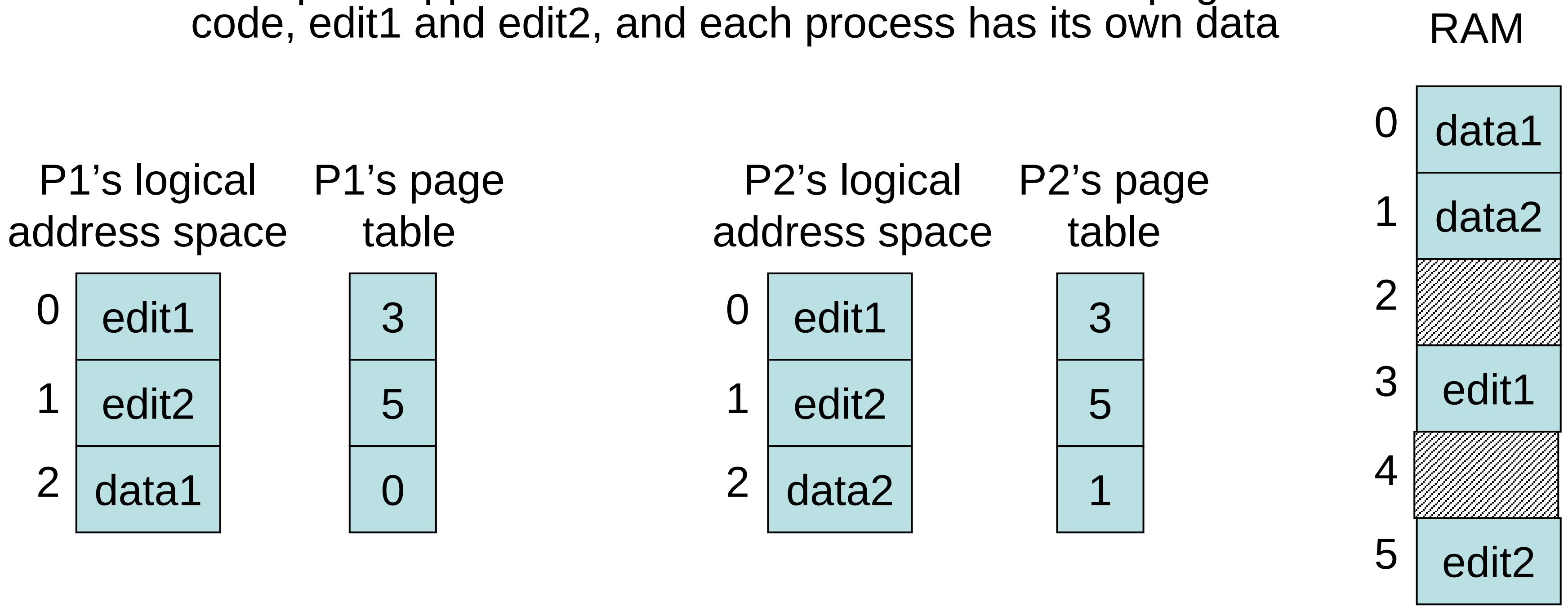
Paging

- Paging with TLB and PTBR



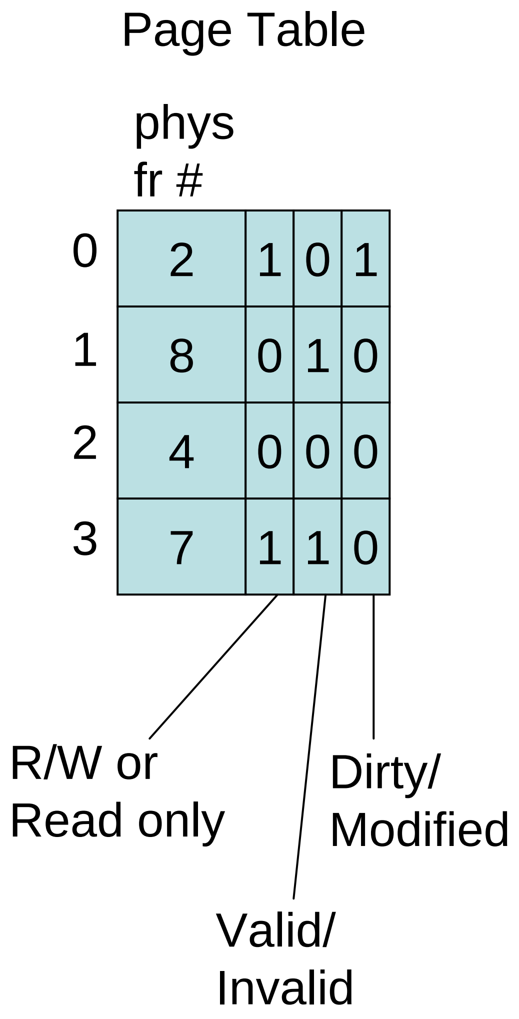
Paging

- Shared pages
  - if code is thread-safe or reentrant, then multiple processes can share and execute the same code
    - example: multiple users each using the same editor (vi/emacs) on the same computer
  - in this case, page tables can point to the same memory frames
  - example: suppose a shared editor consists of two pages of code, edit1 and edit2, and each process has its own data



Paging

- Each entry in the page table can actually store several extra bits of information besides the physical frame # f
  - *R/W or Read-only bits* - for memory protection, writing to a read-only page causes a fault and a trap to the OS
  - *Valid/invalid bits* - for memory protection, accessing an invalid page causes a page fault
    - Is the logical page in the logical address space?
    - If there is virtual memory (we'll see this later), is the page in memory or not?
  - *dirty bits* - has the page been modified for page replacement? (we'll see this later in virtual memory discussion)

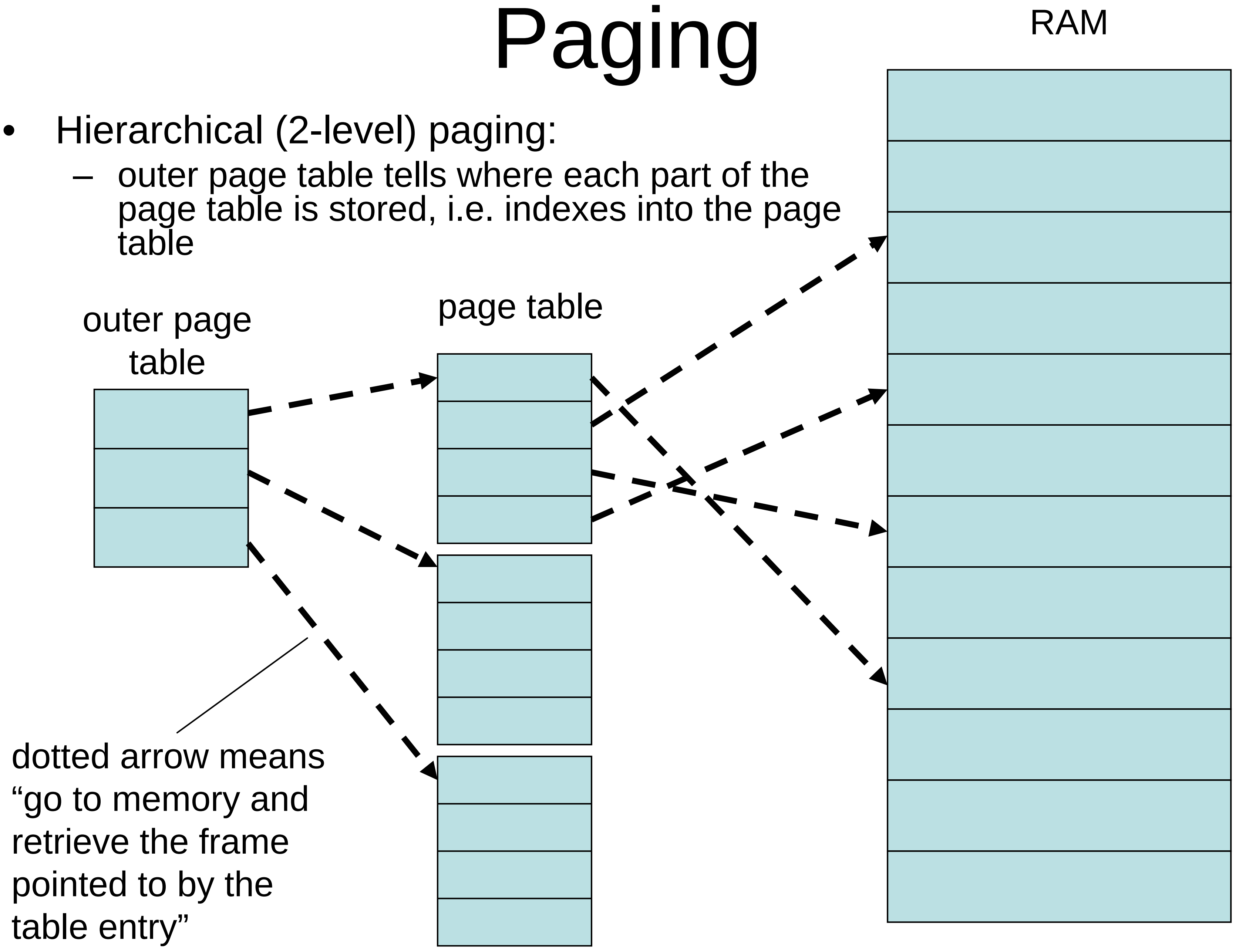


Paging

- 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*
  - 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*.
    - subdividing a process into pages helped to fit a process into memory by allowing it to be scattered in non-contiguous pieces of memory, thereby solving the external fragmentation problem
    - so reapply that principle here to fit the page table into memory, allowing the page table to be scattered non-contiguously in memory
    - This is an example of 2-level paging. In general, we can apply N-level paging.

Paging

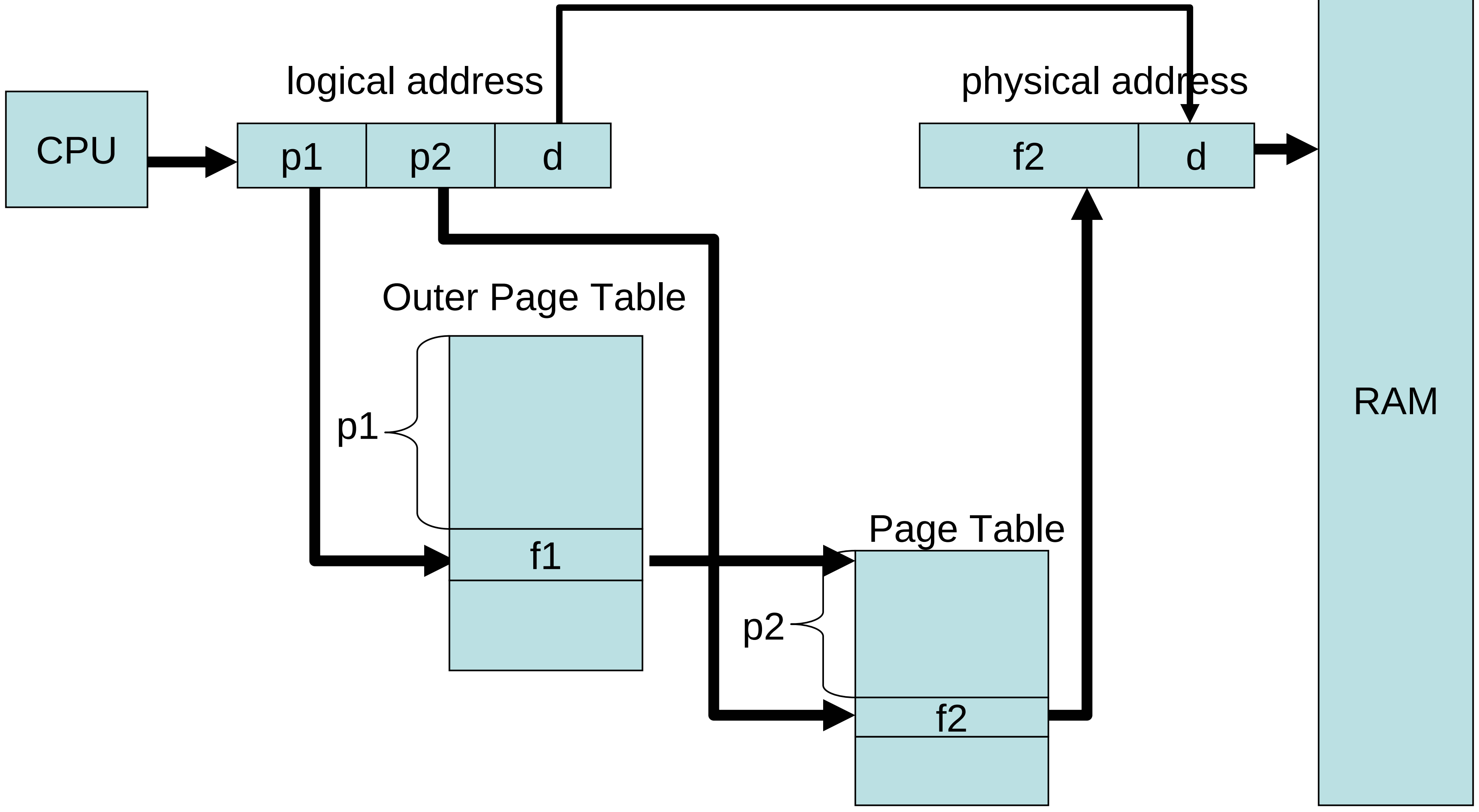
- Hierarchical (2-level) paging:
  - outer page table tells where each part of the page table is stored, i.e. indexes into the page table



dotted arrow means "go to memory and retrieve the frame pointed to by the table entry"

Paging

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


logical address
CPU
p1
p2
d
physical address
f2
d
RAM
Outer Page Table
p1
f1
Page Table
p2
f2