CSCI 3753
Operating Systems

Memory Management
Paging and Segmentation

Chapters 8 and 9

Lecture Notes By
Shivakant Mishra
Computer Science, CU-Boulder
Last Update: 10/17/17

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 paging

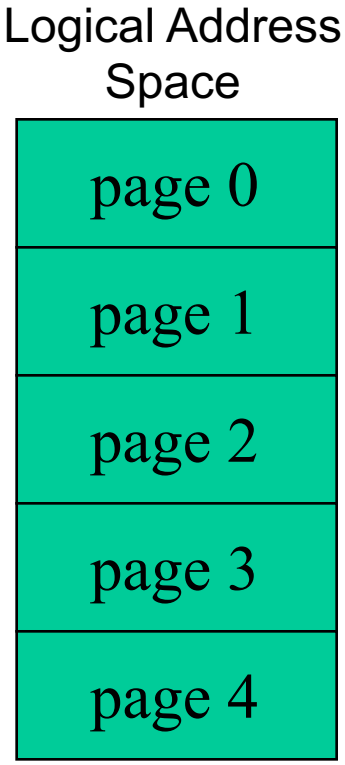
Paging

Divide the logical address space into fixed-size ***pages***

Main memory is divided into similarly-sized frames

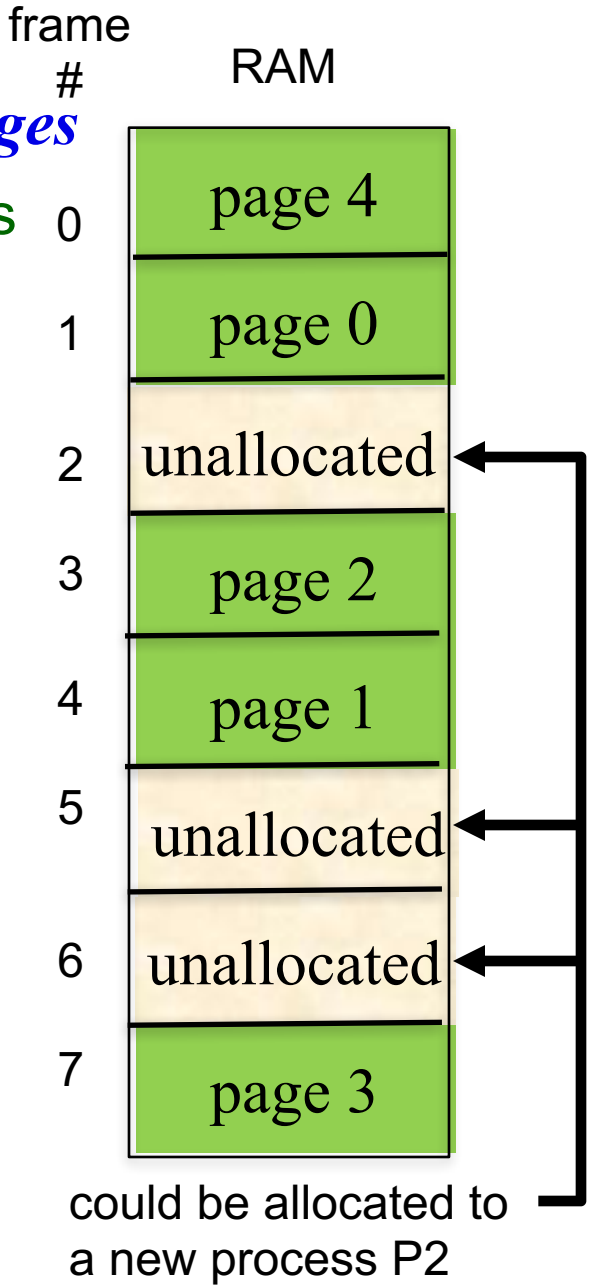
Each page is mapped to a page frame

Maintain a page table to store this mapping



Page Table

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



Paging

- Typical page size is 4-8 KB
  - Linux system call: *getpagesize*( ); command line: getconf PAGESIZE
  - Example: if a page table allows 32-bit entries, and page size is 4 KB, then can address 2^{44} bytes = 16 TB of memory
  - Example: a 4 GB 32-bit address space with 4 KB/page (2^{12}) implies that there can be $2^{32}/2^{12}$ = 1 million entries in a process's page table. Your page table would need to be >= 20 bits/entry.
- No external fragmentation, but internal fragmentation
  - Example: if my process is (2^{12} + 1) B, and each page size is 4 KB, then I have to allocate two pages = 8 KB, so that 4095 B 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: Address Mapping

- Given a logical address *A*, what is the corresponding physical address?
- Page size: *s* bytes

  Page number, $l = A/s$

  Page frame number = f → page table lookup for page # *l*

  Offset within the page, $d = A \bmod s$

  Physical address, $P = f * s + d$
- Problem: Need to do this mapping fast?

Why is fast addressing crucial?

- Need to perform address mapping for every memory access
- Example: Consider a high level statement:

```
x = y + z;
```

The corresponding assembly language translation can be:

```
mov eax   [y]
mov ebx   [z]
add eax ebx
mov [x]   eax
```

- Three memory access
- In general, a high level instruction can result in 3-5 memory accesses

Announcements

- Midterm exam
  - Thursday, October 26 in class
  - Closed book, closed notes
  - Lecture sets 1 -11, chapters 1 – 6 and 13
- Quiz on Friday
  - Process management: lecture sets 5 – 11
  - Twice the weightage
- Exam review
  - During recitation on Friday
  - CAs will hold a special review session

Recap…

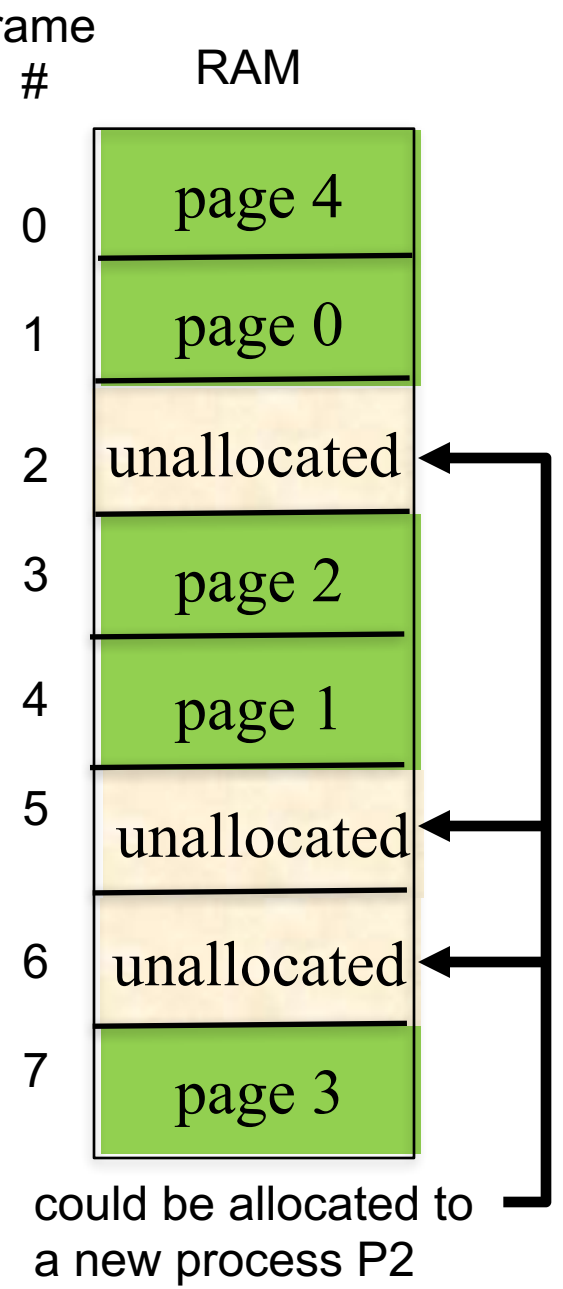
- Paging
  - Logical address space is divided into pages
  - Memory is divided into frames
  - A page table stores mapping between pages and frames

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



Recap…

- Address mapping
  - Logical address A, page size s

  Page number, $l = A/s$

  Offset, $d = A\%s$

  Frame number, f → page lookup from page table

  Physical address $= f * s + d$
- Problem: How to do this mapping fast?

How to compute fast?

- Use sound programming techniques
  - Avoid byte/memory copy as much as possible
  - Use threads instead of processes
  - Incorporate parallelism as much as possible
- Use caching
- Get help from hardware

Paging: Efficient Address Mapping

- Need to perform three operations on logical address A
  - Division to get page #
  - Page table lookup to get frame number
  - Mod to get offset

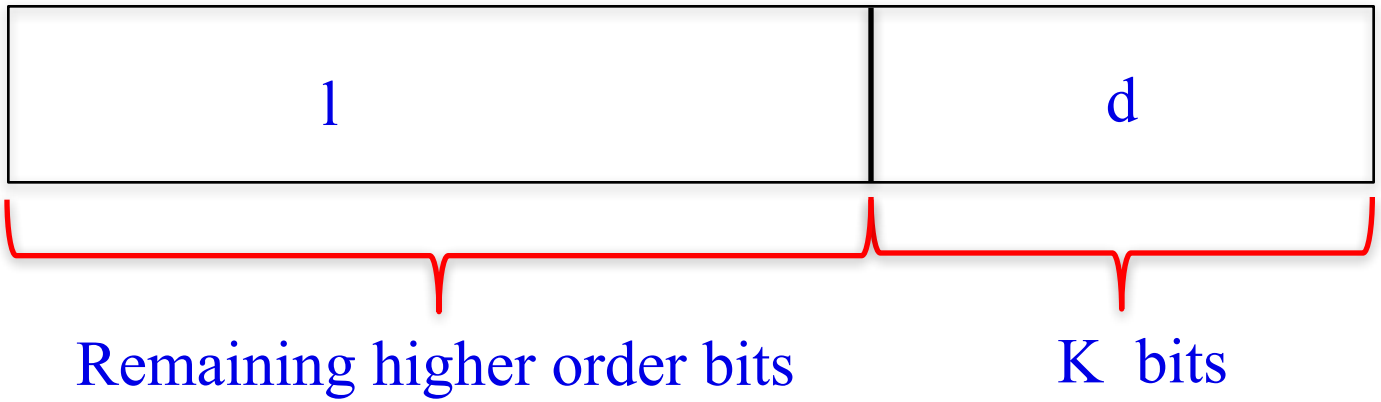
Choose a page size that is a power of 2

Page size = 2^K

Page number, $l = A / 2^K$ Right shift A by K times

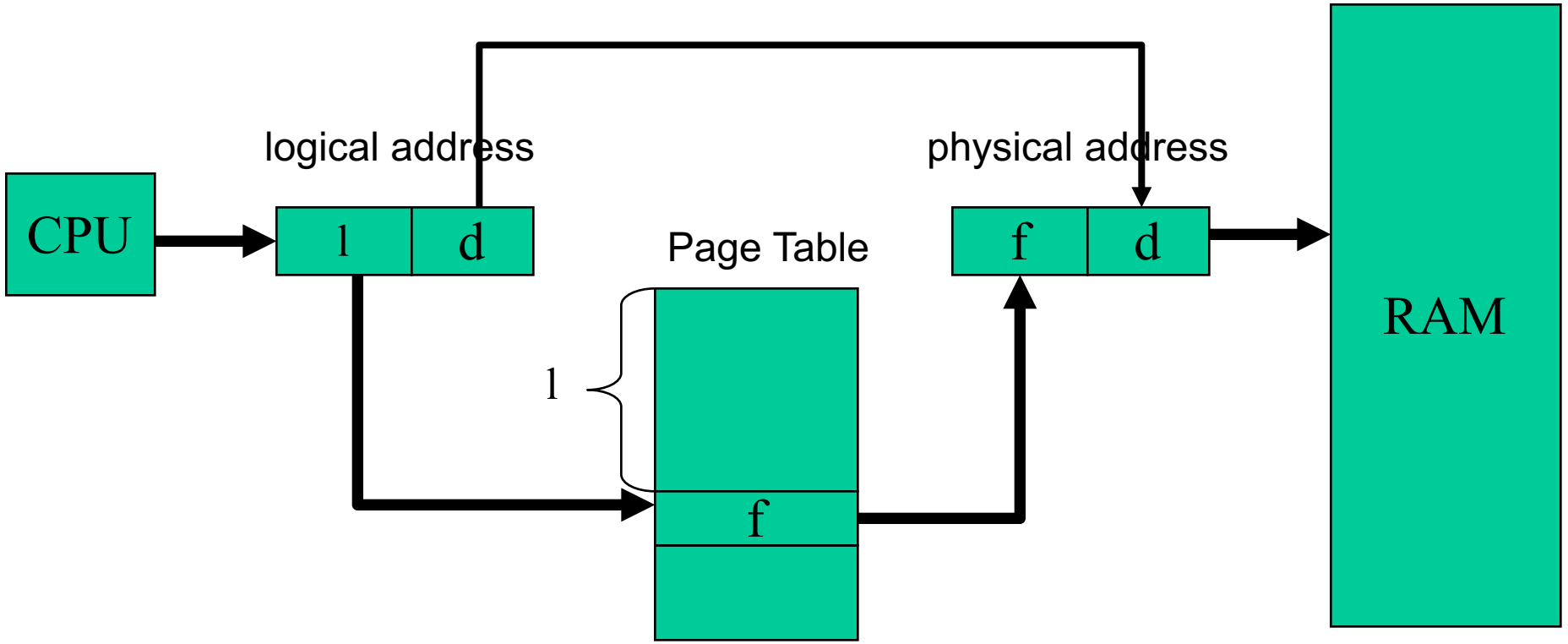
Offset, $d = A \bmod 2^K$ Lower order K bits of A

Address A



Paging

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



Implementing Page Tables

- How to do address mapping fast?
- Problem: page table lookup: page # →frame #
- 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

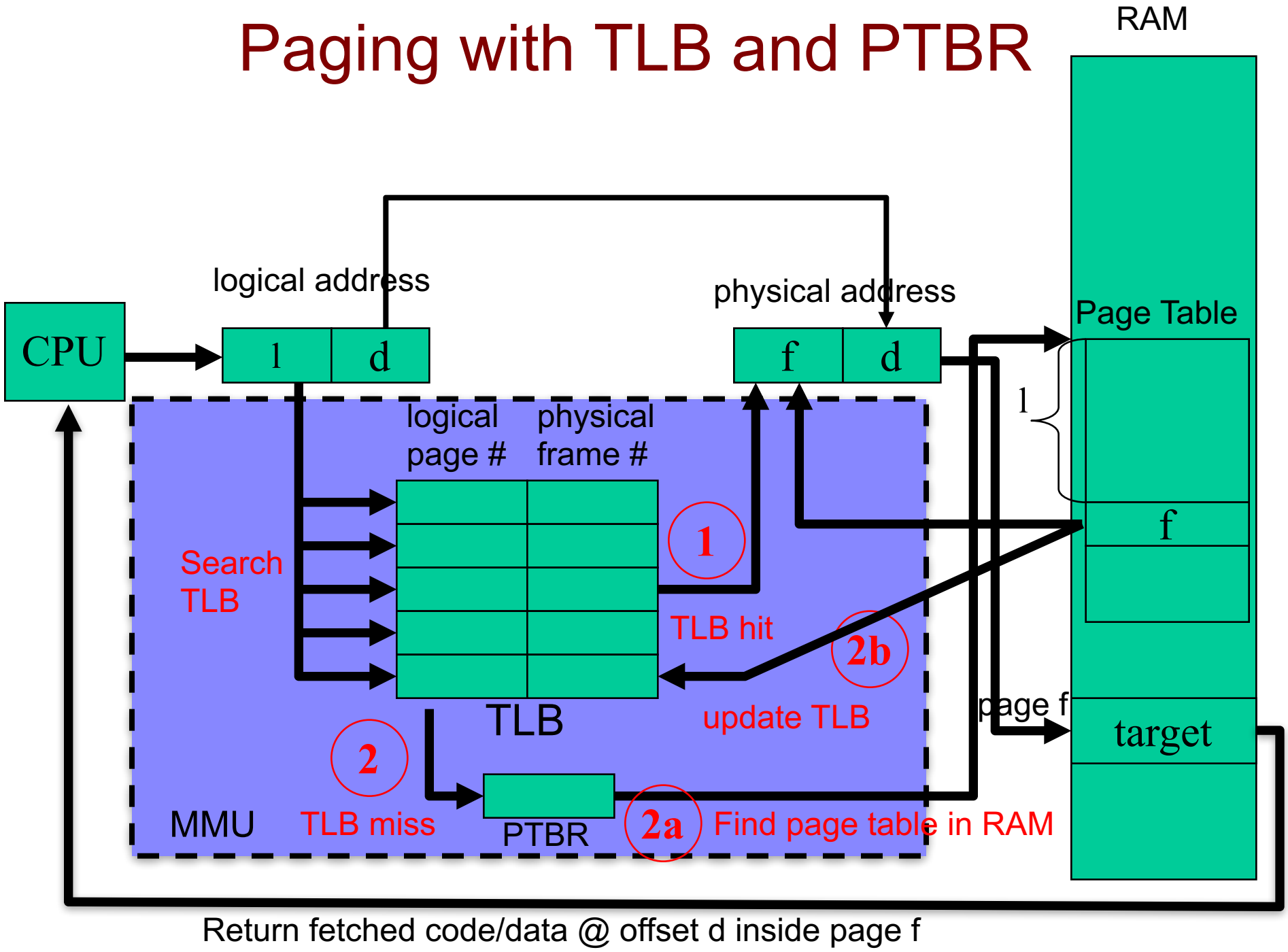
Implementing Page Tables

- 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:
    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 and Caching

- Option #3:
  - Cache a subset of page table mappings/entries in a small set of CPU buffers called *Translation-Look-aside Buffers* (TLBs)
    - 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
    - <key, value> pair
  - Several TLB caching policies
    - Cache the most popular or frequently referenced pages in TLB
    - Cache the most recently used pages
  - Goal is to maximize TLB hits and minimize TLB misses

Paging with TLB and PTBR



Paging and Caching

- On a context switch, since different processes have different page tables, the on-chip TLB entries would typically have to be entirely invalidated/completely flushed (x86 behavior)
  - 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 process ID matching the current task are considered valid. (See DEC RISC Alpha CPU)
  - In Intel Pentium Pro, the page global enable (PGE) flag in the register CR4 and the global (G) flag of a page-directory or page-table entry can be used to prevent frequently used pages from being automatically invalidated in the TLBs on a task switch
  - ARM allows flushing of individual entries from the TLB indexed by virtual address

Shared Pages

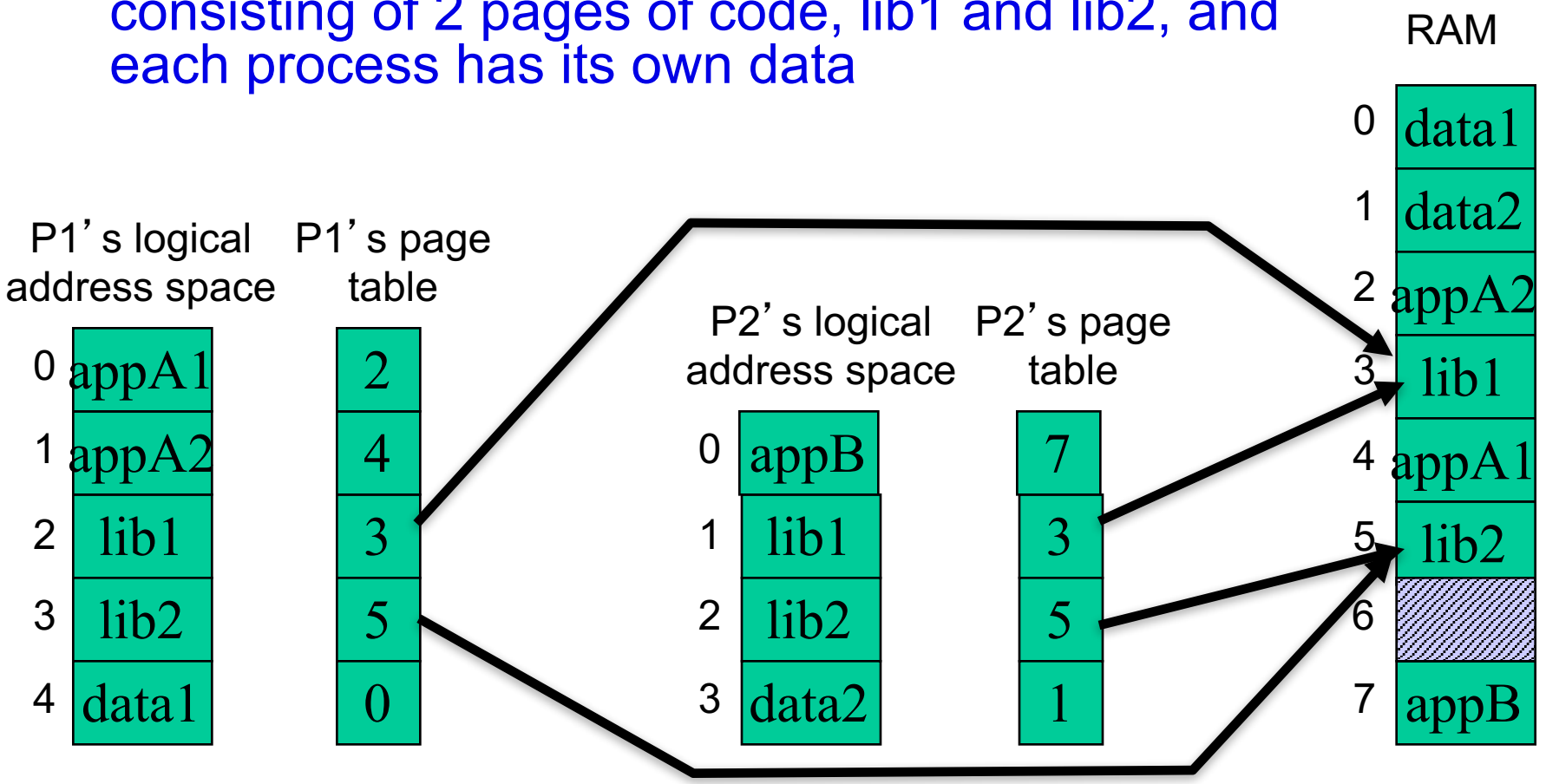
- Paging allows sharing of code and data between different processes
- Simply have entries in different process' page tables point to the same physical page/frame(s)
  - Sharing code:
    - Forking a child process causes the child to have a copy of the entire address space of the parent, including code. 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.
    - Share libraries (image, C, …) between multiple processes, so map each process' page table to point to the same shared library pages in memory
    - Shared code should be thread-safe and reentrant

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

- Consider 2 apps 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

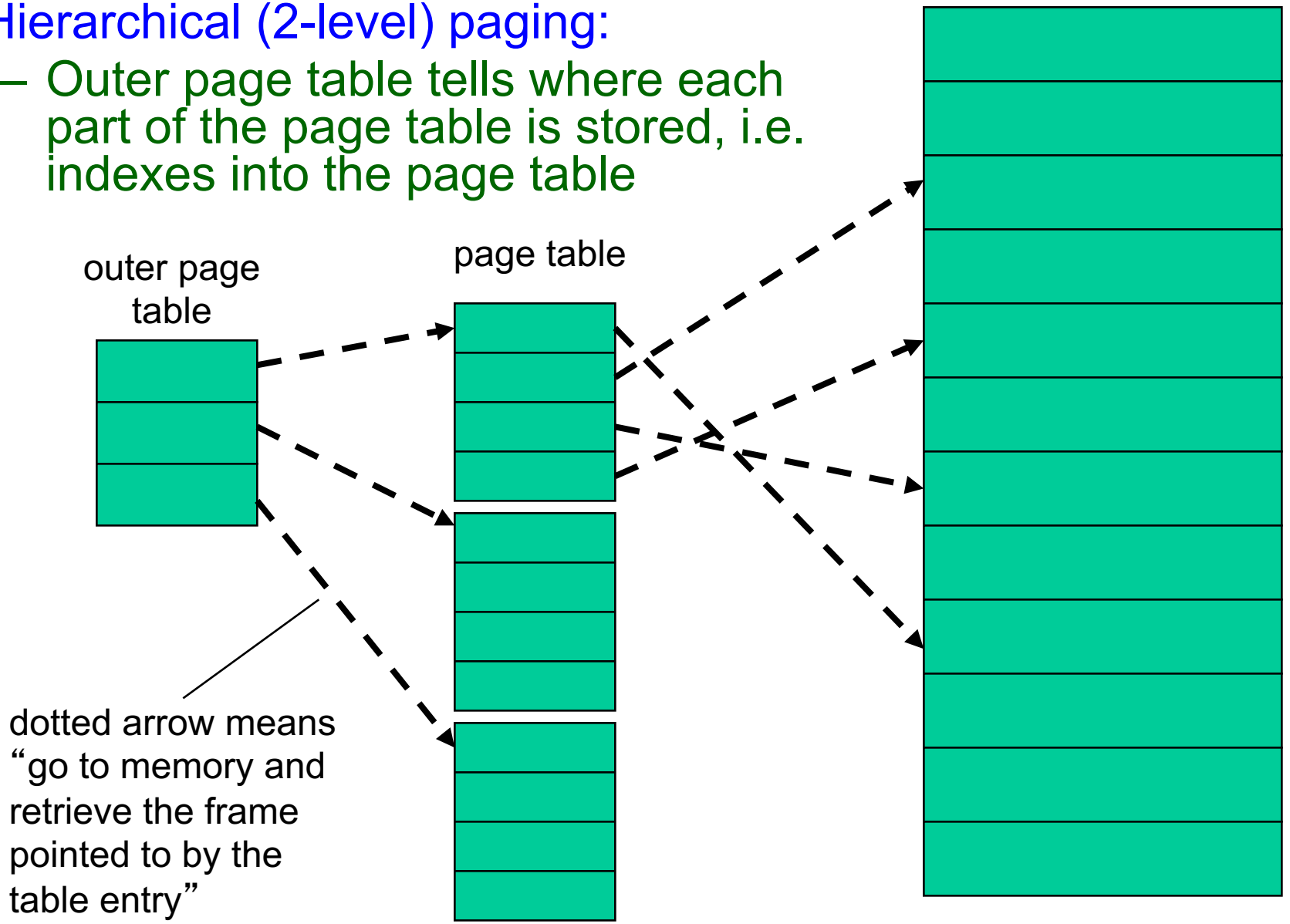


Page Table Size

- Problem with page tables: they can get very large
  - It's hard to find contiguous allocation to store each page table, each of which can be as large as 1 MB.
  - Solution: page the page table →*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

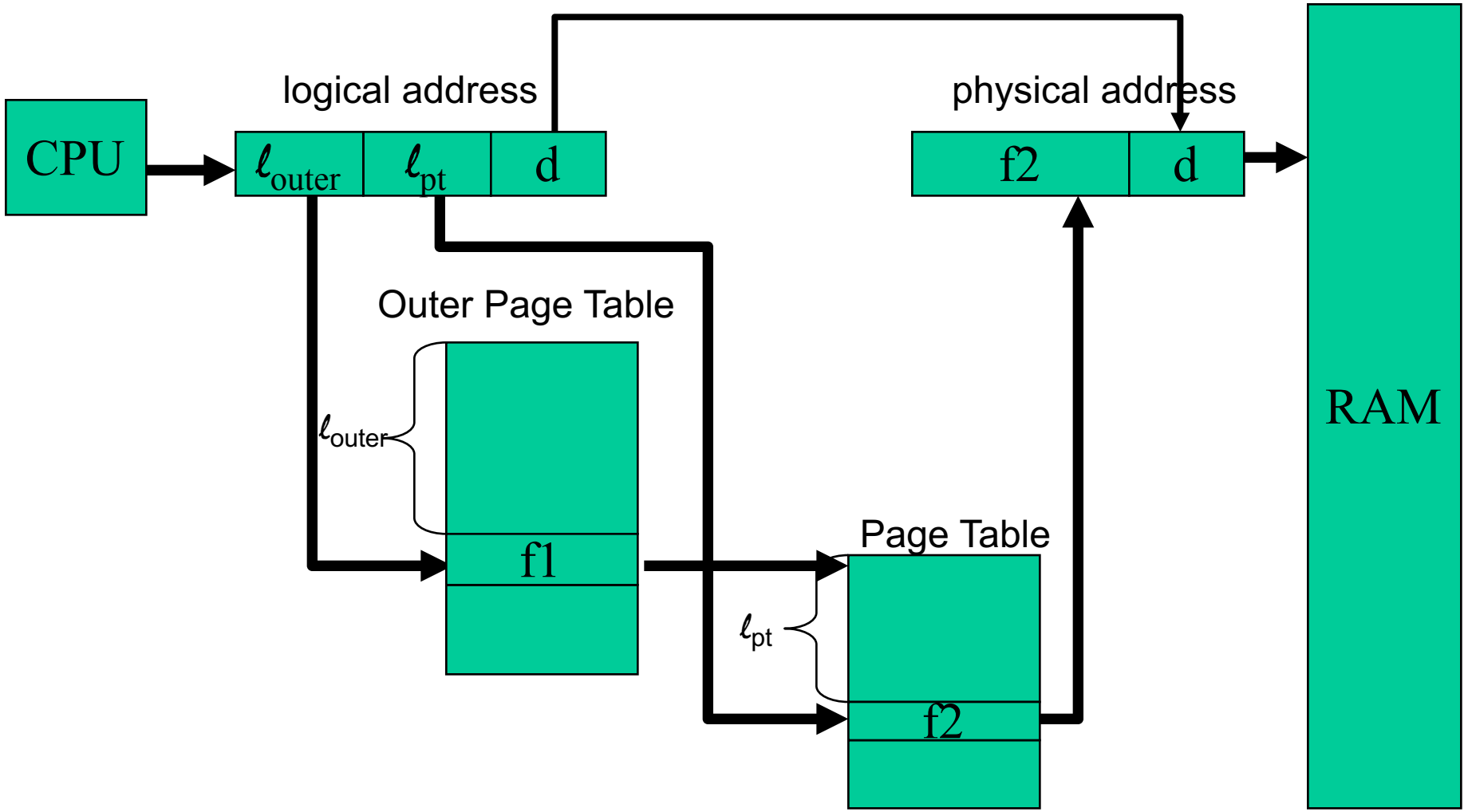
Hierarchical Paging

- 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

- Hierarchical page tables address the problem of contiguous memory need to store page tables
- But, they do not solve the problem of the large page table size
  - if several processes have a page table that contains a million entries, then a not-insignificant fraction of RAM becomes devoted to storing/managing 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

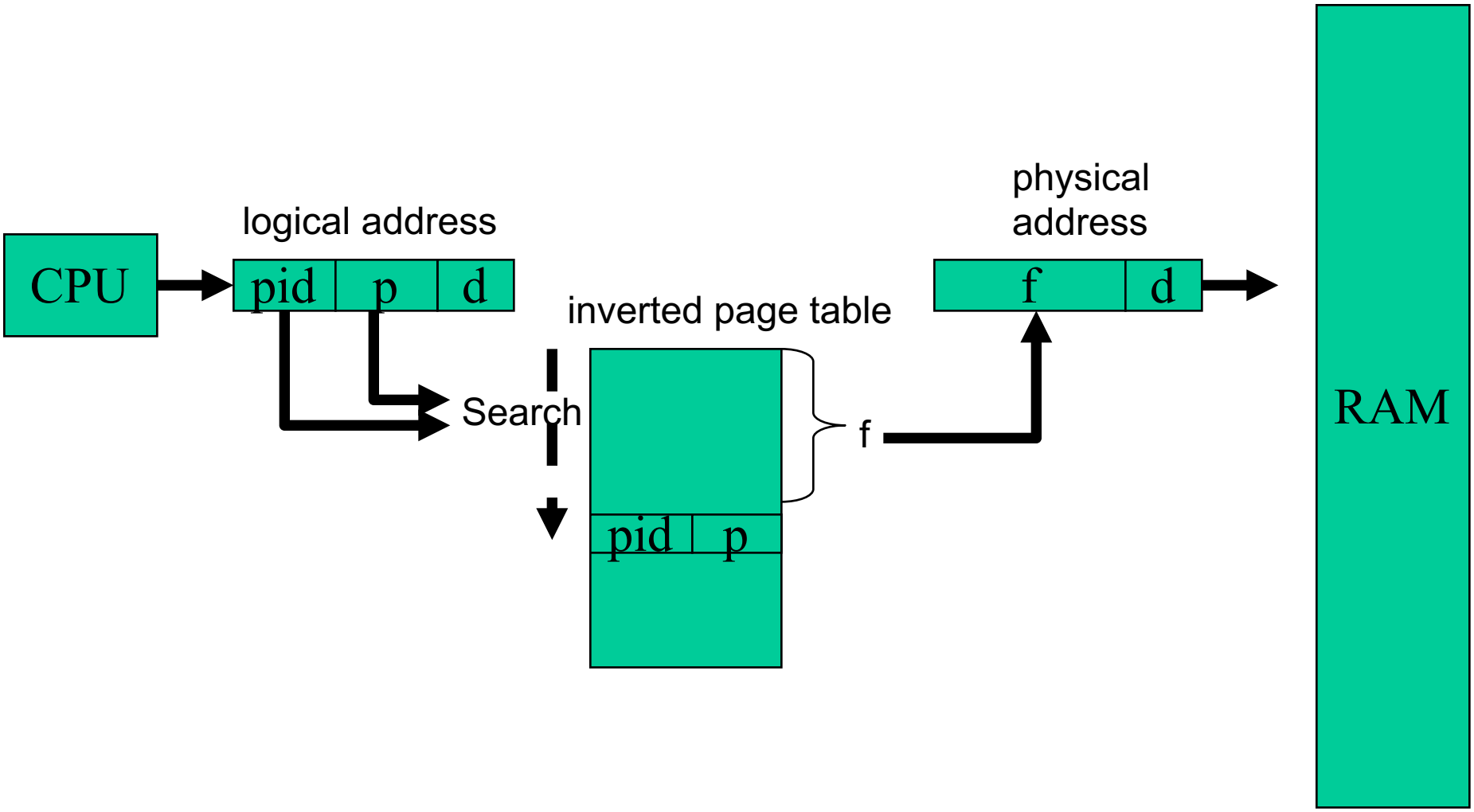
Inverted Page Table

- 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

- Still want to use an inverted page table to map a logical page to a physical frame
  - Since the IPT stores an array of <pid, p> pairs, then given a pid and logical page p, you have to *search* through the IPT to find an entry that matches
    - this can be slow – use hash table
  - 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



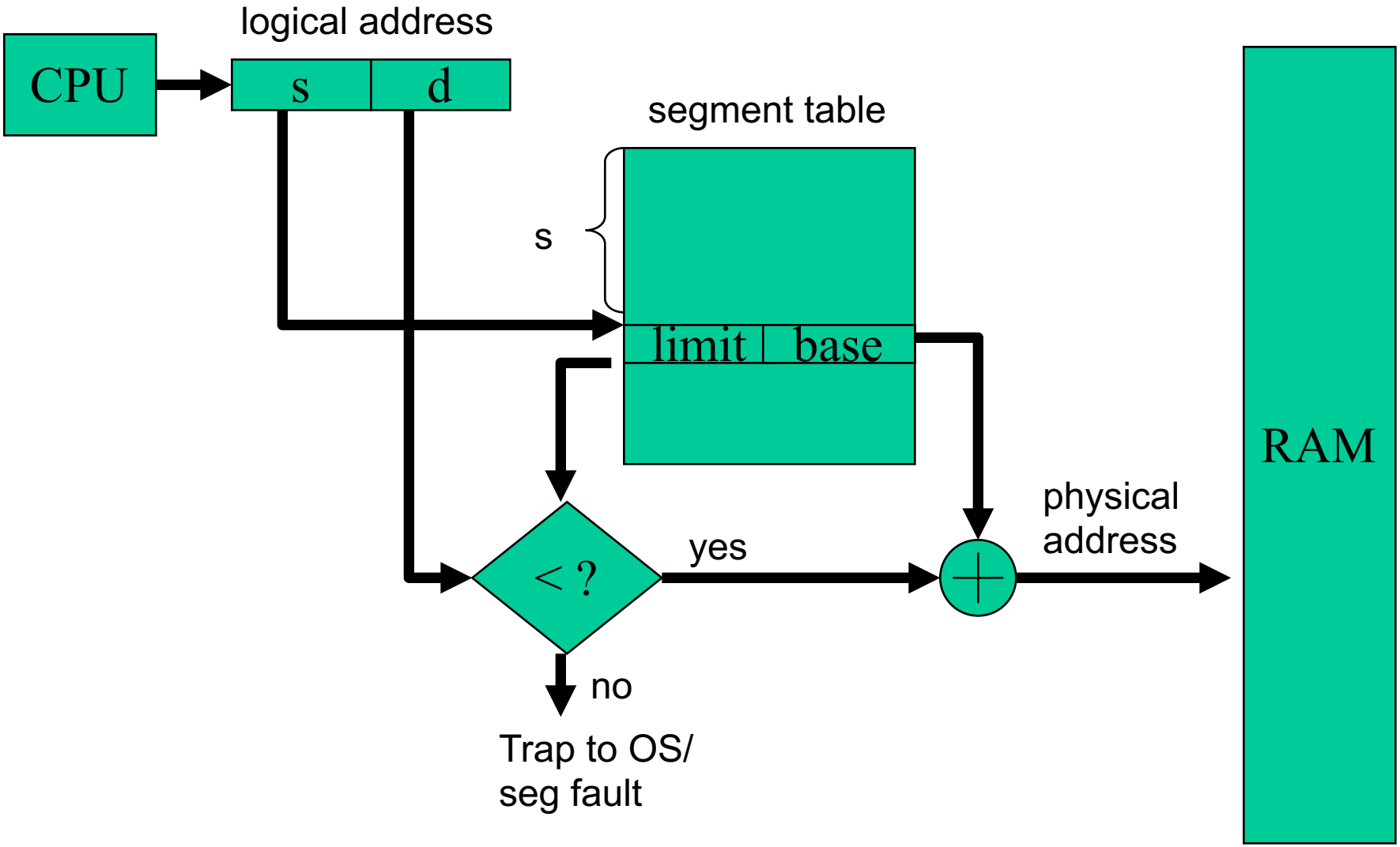
Segmentation

- An alternative to dealing with external fragmentation using fixed size pages is instead to employ variable sized *segments*
  - Subdivide 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 subdivide via 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
- Each entry of the segment table needs a base and limit field to place in the base and limit registers, because segments are variably sized

Segmentation



Segmentation

- Pentium supports pure segmentation and segmentation with paging
- Linux on the Pentium uses segmentation sparingly
  - my reading of the literature suggests that paging is more popular than segmentation
  - Linux supports 3-level paging
- Drawback of segmentation: fragmentation of free space in RAM becomes even more complex and hard to manage