Virtual Memory (I)

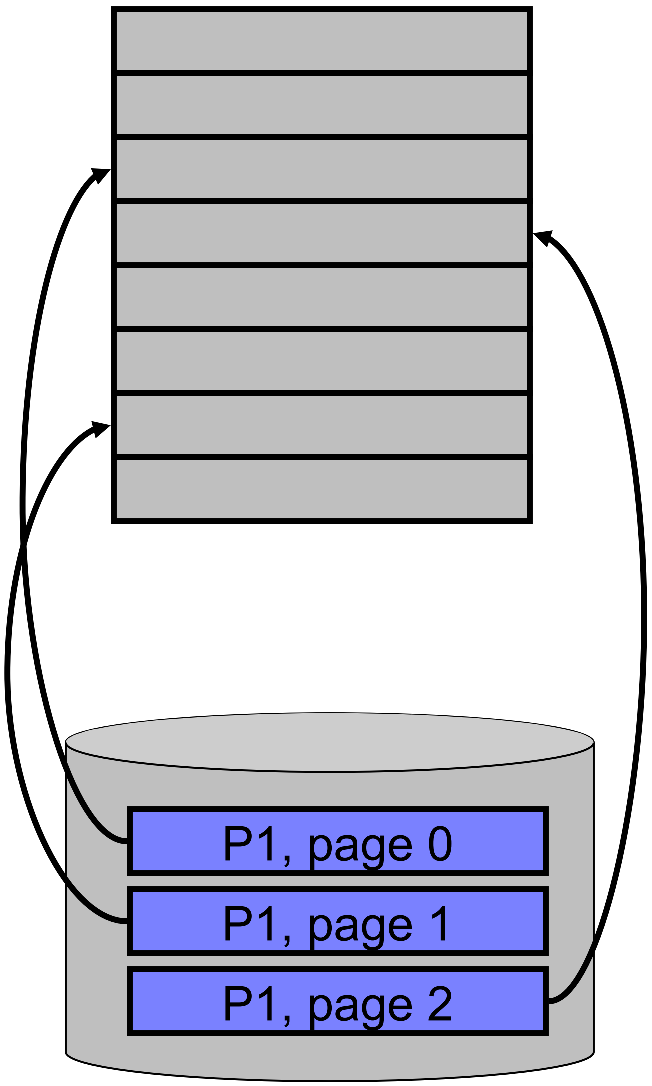
ICS332
Operating Systems

We are in a bind

- With contiguous memory allocation we have a big problem: we may have a bunch of small holes in memory when a big process arrives
- A radical solution: break up process address spaces into tiny bits!
- Typical analogy: If I give you a bunch of cardboard boxes to fit in a bunch of bins of various sizes, things get really simple if I give you a box cutter
- In fact, things are very easy if I cut all the boxes in slices all of the same size (then just put the slices into the bins in whatever way until all the bins are filled)
- We call each "same-size piece" of a process' address space a page, and we talk of "paging"

Paging

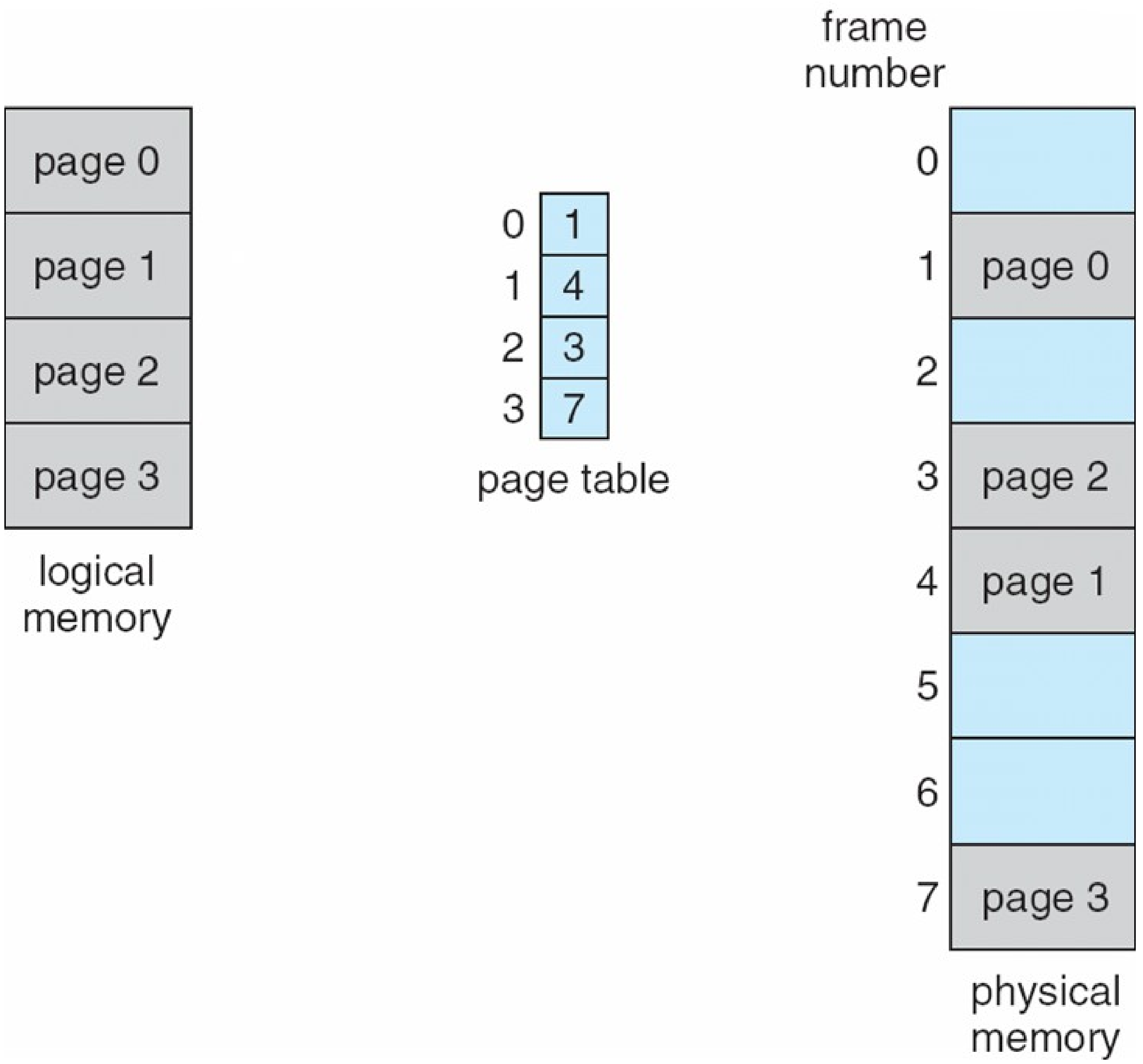
- Most systems today structure a process' address space as a set of fixed-size pages
  - Requires the OS and the hardware to work together
  - Same structure in memory and on disk
- Physical memory is structured as fixed-size frames
- A page can fit in any available frame
- This allows non-contiguous allocations



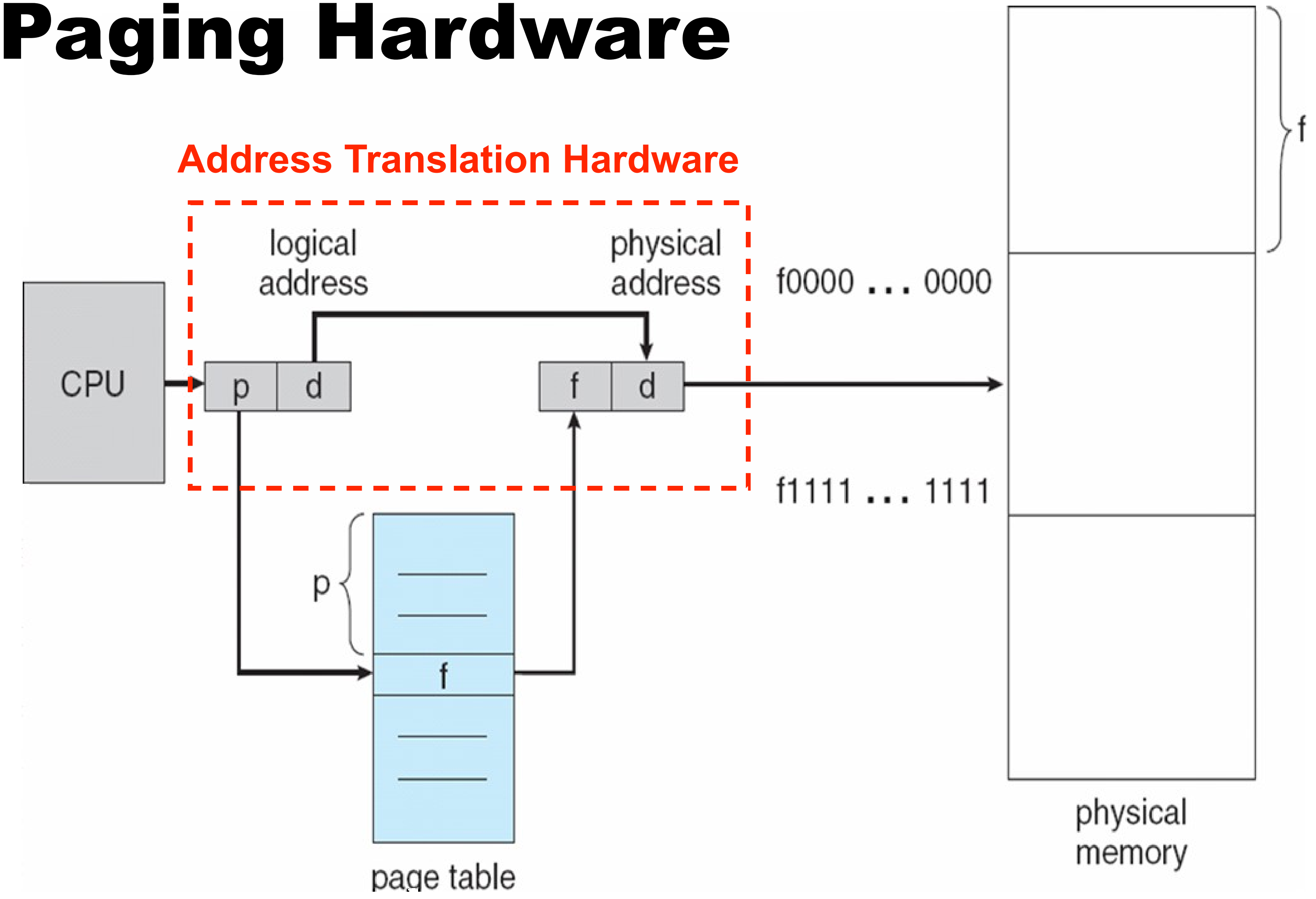
Page Number

- When the CPU issues a logical address, this address is split in two:
  - The logical page number (p)
  - The offset within the page (d)
  - Essentially, given an address "the byte at address x", we need to transform it into "the x-th byte in the y-th page"
- The process has the illusion of contiguous logical pages starting with page 0
- But in fact, in physical memory, each page is in a frame
  - Therefore, the offset in the page is the same as the offset in the frame
  - If the y-th page is stored in the z-th frame, then the x-th byte in the y-th page is also the x-th byte in the z-th frame
- So we need to keep track of where each page is
- To do so, we use a page table
  - Important: Each process has its own page table

Page Table


frame number
page 0
page 1
page 2
page 3
logical memory
0 1
1 4
2 3
3 7
page table
0
1 page 0
2
3 page 2
4 page 1
5
6
7 page 3
physical memory

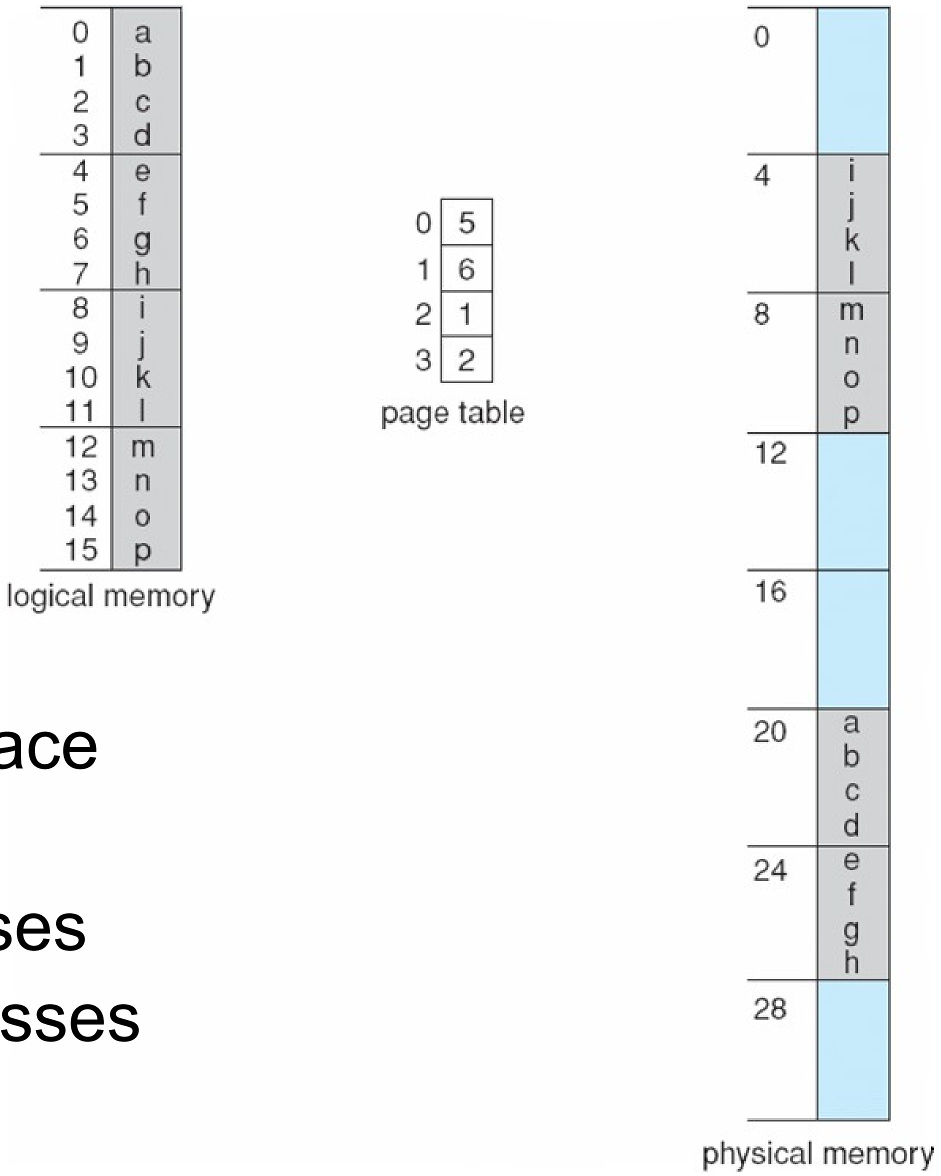
Paging Hardware



Page Size

- The page size is defined by the hardware and is a power of 2
- If the size of the logical address space is 2^m words, and a page is 2^n words then:
  - The m-n high-order bits of the address are the logical page number
  - The n low-order bits of the address are the offset into the page (between 0 and 2^{n-1})
- In everything from now own, we’re going to assume that “a word” is “a byte”

Small Example



- 32-byte memory
- 16-byte address space
- 4-byte pages
- 4-bit logical addresses
- 5-bit physical addresses

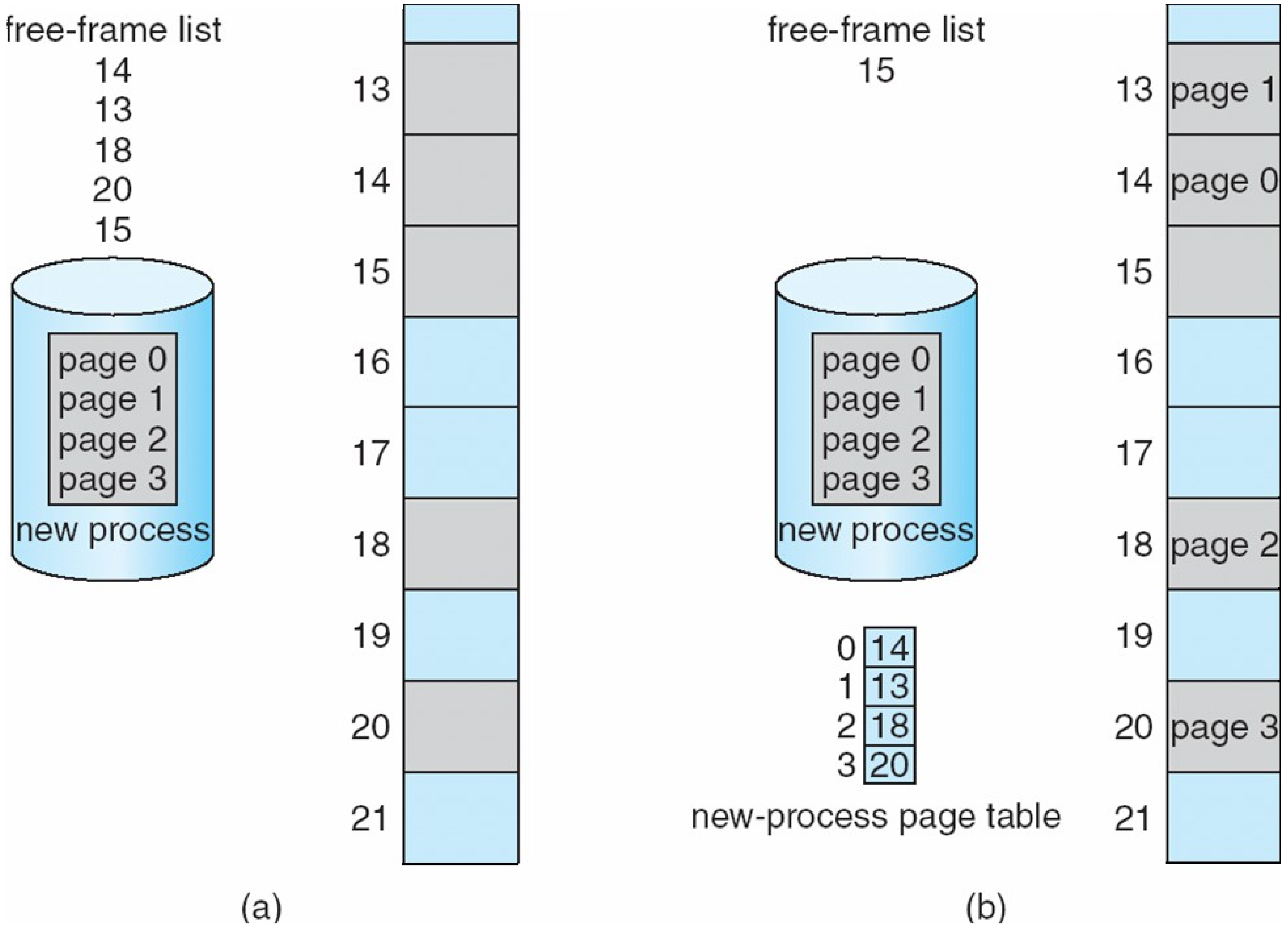
Fragmentation

- We can only have internal fragmentation (no external)
  - Worst case: a process needs n pages + 1 byte
  - On average we expect that each process wastes half a page
- Therefore small pages are good
- But larger pages have advantages
  - Smaller page tables, hence less lookup overhead
  - Loading many small pages from a hard drive take more time than loading few large pages
- Typical sizes: 4KiB or 8KiB
  - The “getconf PAGESIZE” Linux command will let you know
- Modern OSes support multiple page sizes (Lin: Huge pages; FreeBSD: superpages; Win: Large pages) thru CPU support.
- The OS keeps track of free frames and of what process is allocated to which frame

Free Frames

- The OS keeps track of free frames
  - Much simpler than keeping track of a list of holes that all have different sizes as would be needed for contiguous allocation
- The data structure is called the free frame list
- When a process needs a new frame (e.g., upon creation) then the OS takes frames from the free frame list and allocated them to the process

Giving out Frames



Paging and Hardware

- The address translation hardware had better be very fast
  - Each address coming out of the CPU is translated!
- Modern OSes keep the page table of each process in main memory
  - And those can be very large, with millions of entries, i.e., several MiBs
- When a new process is given the CPU, the dispatcher loads a special register with the address of the beginning of the process' page table: the page-table base register (PTBR)
- This makes it fast to switch page tables
- But it doesn't do anything to speed up translation
  - If anything it adds one level of indirection
- Each memory access will be doubled
  - One access to the page table
  - One access to the memory location of interest
- Memory is what makes computers slow, so doubling the number of memory accesses is not a viable option!

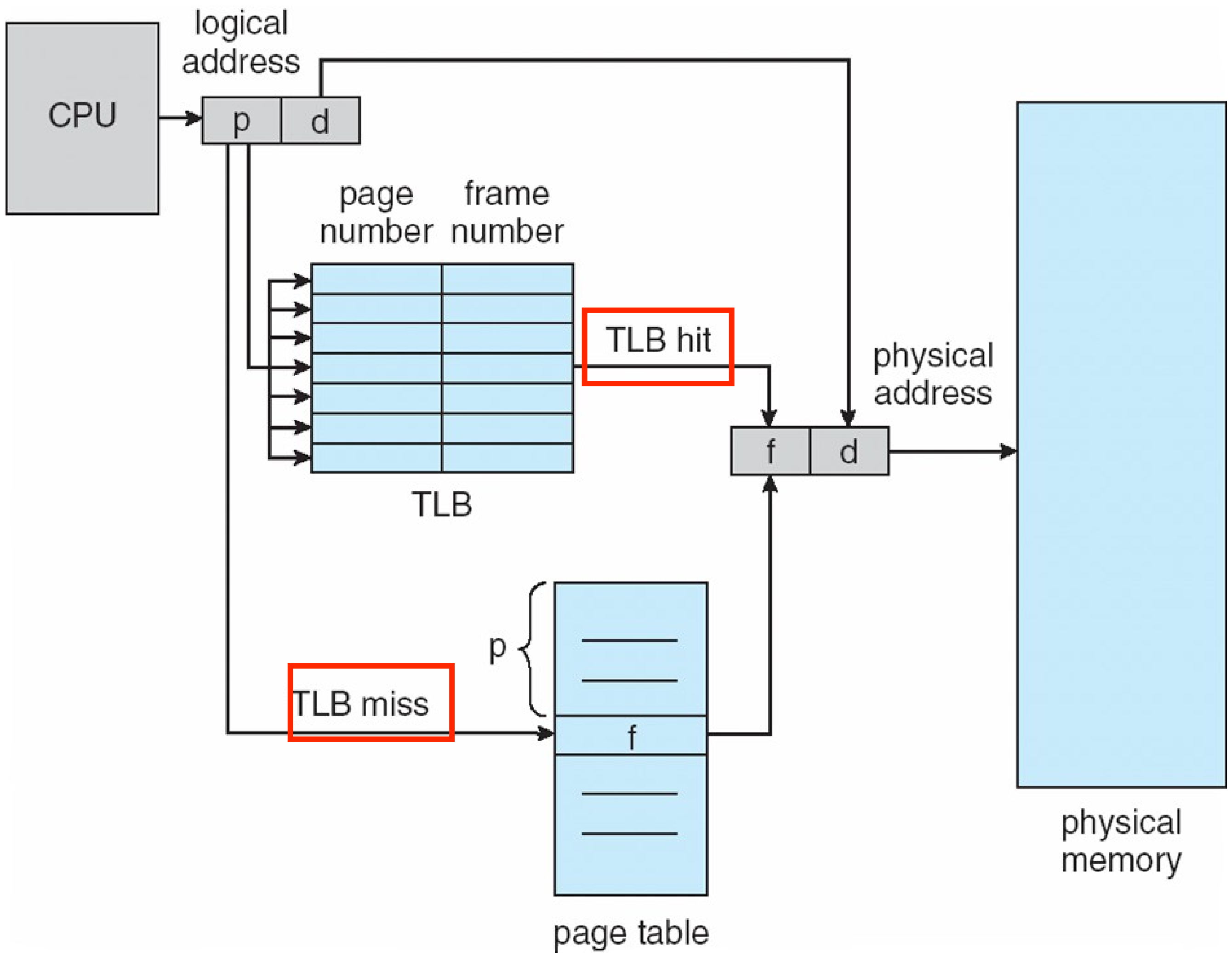
Caching, Locality, etc.

- Caching for memory
  - Temporal locality: repeated access to the same memory location
    - e.g., “sum[i] += x[j]” at each j-loop iteration
  - Spatial locality: repeated access to nearby memory locations
    - e.g., “x[j+1]” is accessed soon after “x[j]”
  - Therefore, we have caches with cache blocks
- We should have even better locality for memory pages:
  - A memory page is much bigger than a cache block
  - If a program makes an access to a memory page, it will most likely access that page again next
    - Programs rarely jump around many different pages
    - You can write one that does, and you’ll see how slow it is!
- Therefore, the same page table entries are looked up and the same physical pages are returned over, and over, and over
  - Seems like a lot of repeated (wasteful) work

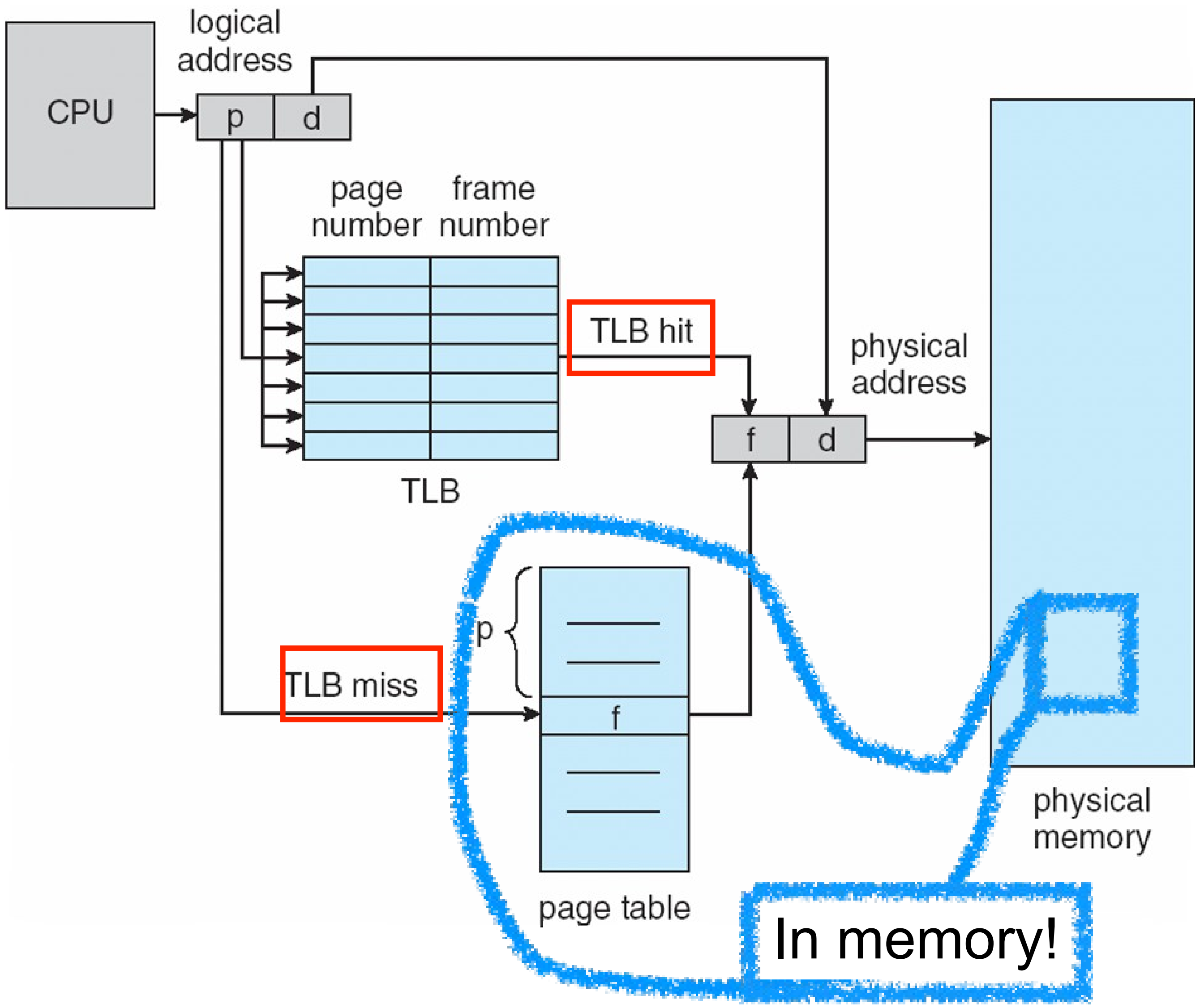
Caching Translations

- There should be a cache for recent page number translations
- Goal: avoid most page table lookups
- This can work if this cache is in hardware and is thus accessible within a cycle
    - Just like some special-purpose L1 cache
- The translation look-aside buffer (TLB)
    - Each entry is a <key,value> pair
    - You give it a key
    - That key is compared (in hardware) in parallel with all stored keys
    - If it is found, then a value is returned
- To be fast the TLB is only between 64 and 1,024 entries
    - And it’s still a pretty expensive piece of hardware
- TLB contains (a few) recently used page table entries

The TLB


CPU
logical address
p
d
page number
frame number
TLB
TLB hit
physical address
f
d
TLB miss
p
f
page table
physical memory

The TLB

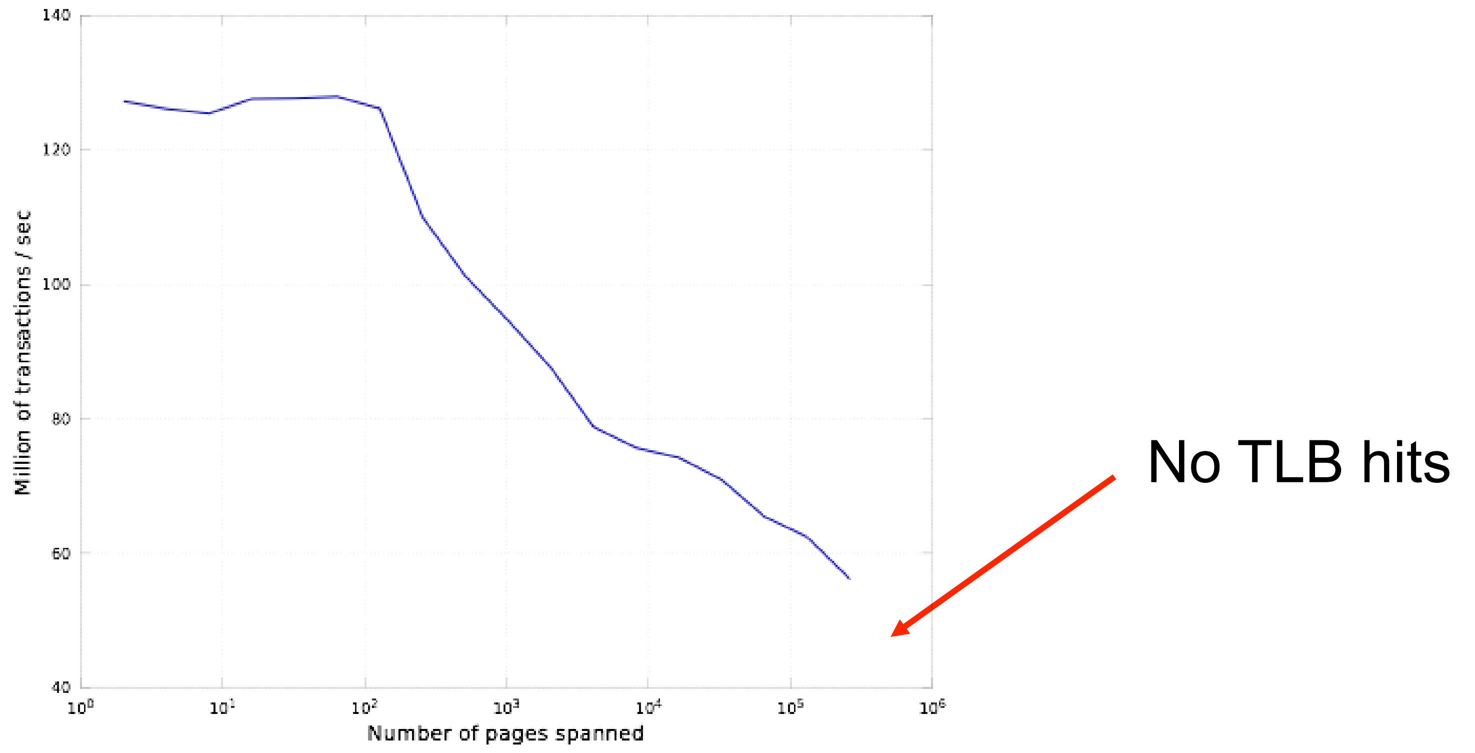

CPU
logical address
p
d
page number
frame number
TLB hit
TLB
physical address
f
d
p
TLB miss
f
page table
physical memory
In memory!

The TLB

- We talk of TLB hit rate and TLB miss rate
  - Like for any other cache
- There must be a replacement policy for the TLB when it’s full: which entry should be evicted to make room for a new one?
  - Least Recently Used (LRU) is probably good but expensive
  - Random is less good but very cheap
- Some TLBs allow for some entries to be marked as “un-evictable”
  - e.g., entries for Kernel code

What Happens with no TLB?

- I’ve written a program to stress the TLB
- tlb_stress.c (on the Web site)
- On my laptop, running this program gives:

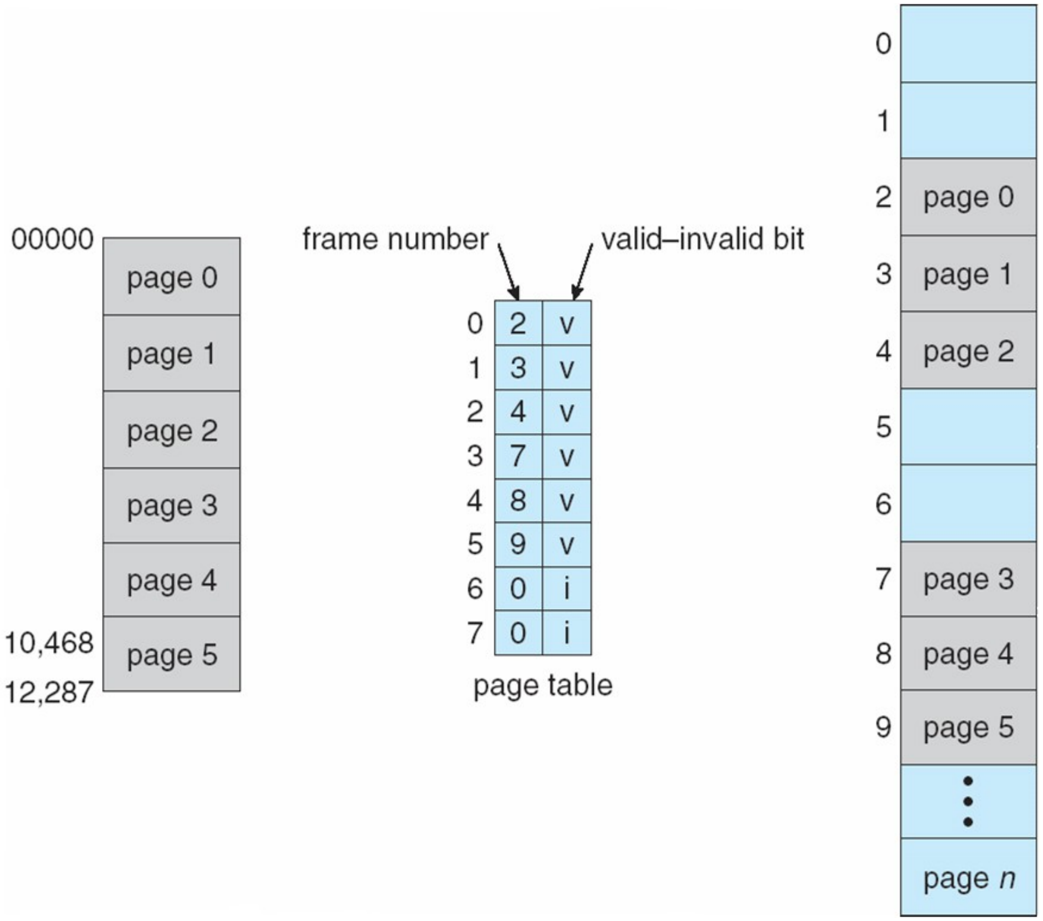


Context-Switch?

- What happens on a context-switch?
- Simple solution: wipe out the entire TLB
  - Called a “TLB flush”
  - Because logical page 7 of process A is not in the same frame as logical page 7 of process B
- ASIDs (Address-space identifiers):
  - Each TLB entry is annotated with a process identifier
  - The TLB can contain data from multiple processes
  - Each lookup attempts to match entry’s ASIDs with the ASID of the current process
    - If mismatch, then it’s a TLB miss

Valid Bit

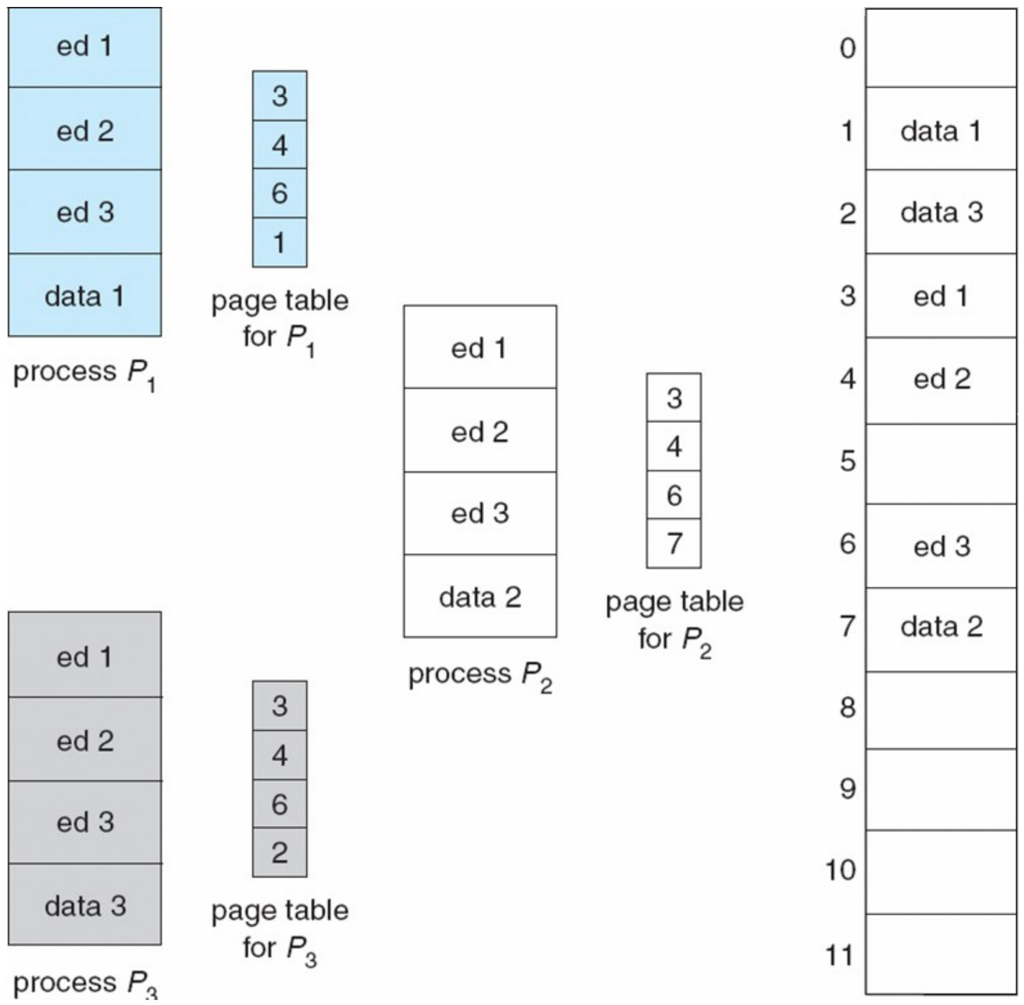
- Each page table entry is augmented by a valid bit
  - Set to valid if the process is allowed to access the page
    - i.e., it is in the process' address space
  - Set to invalid otherwise
- In this example, the address space could potentially reach 8 pages, but right now only 6 are used
  - More malloc() would use up the additional pages
- So if the process generates an address that maps outside of its current address space, a trap can be generated by looking at the valid bit



Shared Pages

- Setting entries in different processes' page tables to point to the same frame leads to memory sharing
- Useful for IPC
  - Can be implemented with special "shared" pages containing the shared memory segments
  - The Kernel can update all pages tables on the shmget/shmat system calls
- Useful for sharing code
  - Provided the code isn't self-modifying
    - The book says that non-self-modifying code is "re-entrant", but there are other conditions necessary to label code as re-entrant

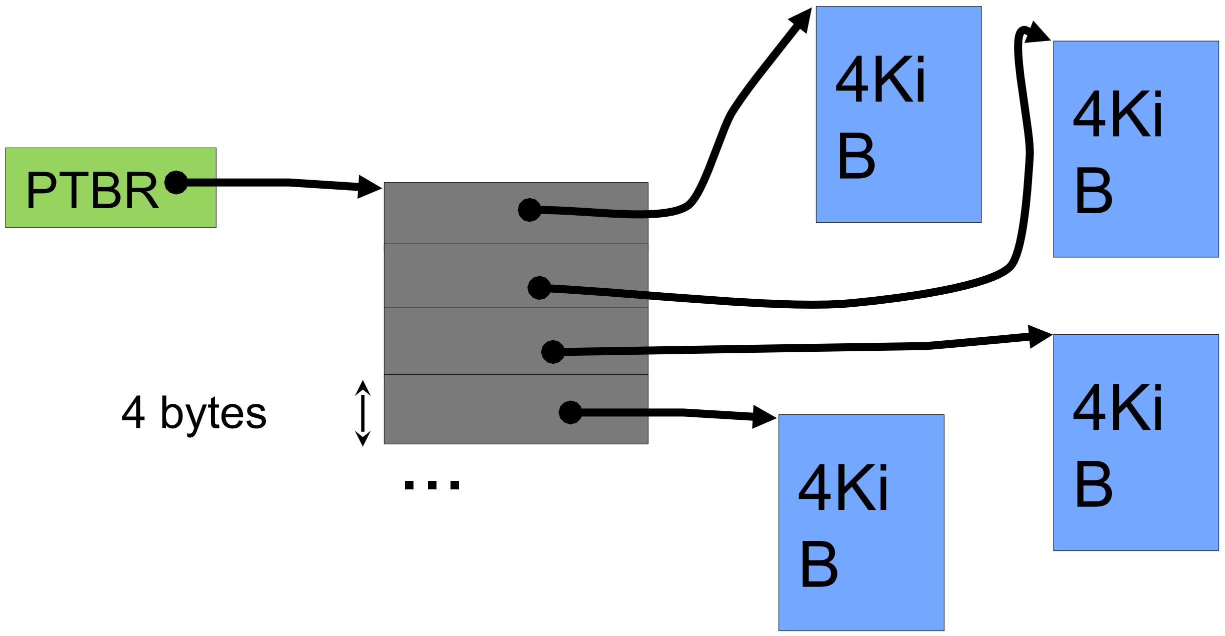
Sharing Code pages



- Three processes that all run a text editor whose code fits into 3 pages
  - Shared library

Page Table Entries

- So far we've shown page table entries as integers
- But we need to store page table entries in memory, which begs the question: How many bytes are in a page table entry?
- Let us consider a 32-bit memory (i.e., 4 GiBytes)
- Each entry in the page table can simply be the address of the first byte of a frame
  - We could store less since we know this address is a multiple of 4KiB = 2^12, meaning that its least significant bits are all 0's
- Since addresses are 32-bit, each page table entry is 4 bytes



Page Table Structure

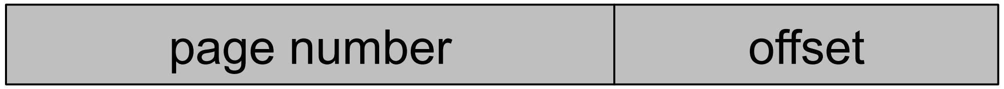
- We've shown page tables as long contiguous arrays
- This could cause a problem
- Example
  - 32-bit logical byte-addressable address space
  - 4KiB pages
  - # page table entries: $2^{32}/2^{12} = 2^{20}$
  - page table entry size: 4 bytes
  - Page table size: 2^{22} = 4 MiB
- Allocating this much contiguous memory is a problem
  - We've been trying to break things apart!
- So let's break the page table apart into... pages
  - that's right: page table pages!!!!

Why do we need page table pages?

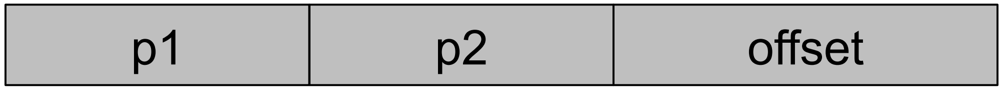
- 4KiB pages
- Page table entries are 4bytes
- I want my page table to fit in ONE page
- How many page table entries in ONE page?
  - $2^{12} / 2^{2} = 2^{10}$
- Therefore I can only have 2^{10} pages in my address space
- Therefore, my address space can be at most $2^{10} * 2^{12}$ bytes = 4 MiBytes
- That's WAY too small

Hierarchical Page Tables

- Consider a 32-bit logical address space, and a 4KiB page size
- The non-hierarchical (standard) view:
  - 12 low address bits: offset within a page
  - 20 high address bits: page number

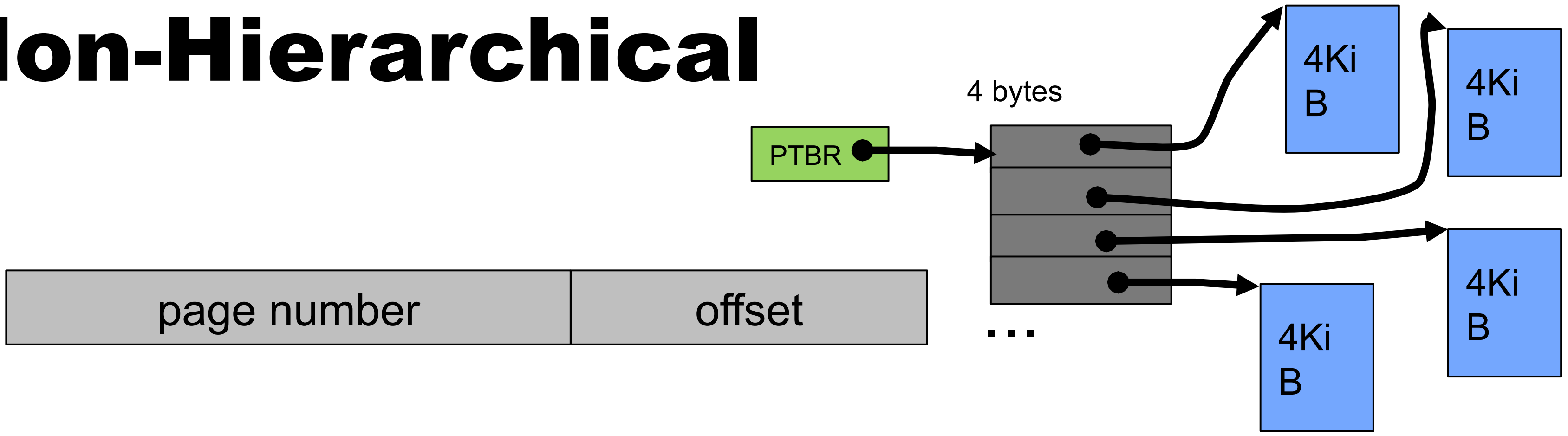


- Two-level hierarchical view:
  - 12 low address bits: offset within a page
  - 10 high address bits: inner page table's page number
  - 10 middle address bits: offset within an inner page table page



  - Let's see how this works...

Non-Hierarchical



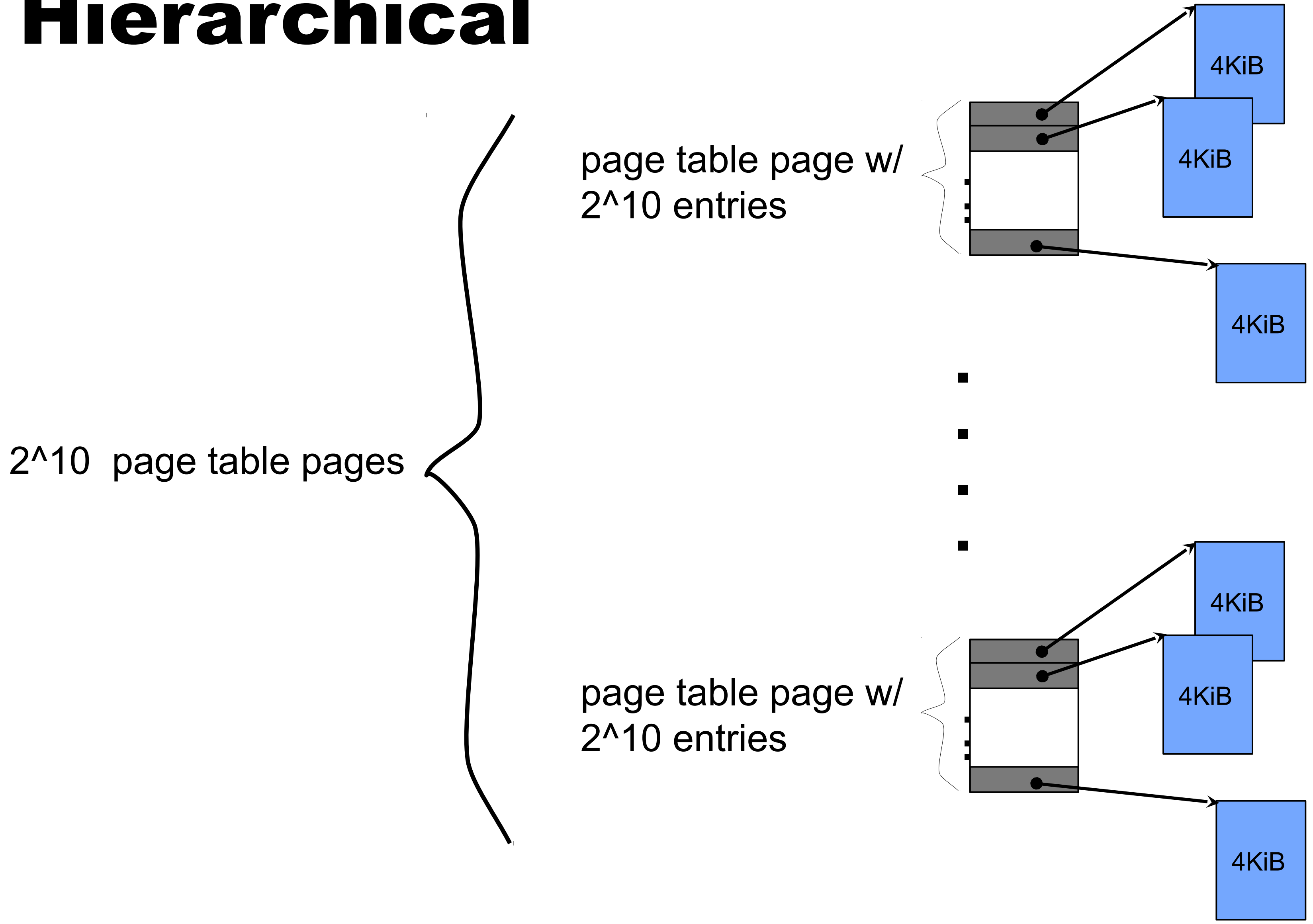
- Assuming a single-level page table, how is a logical address translated to a physical address?
- Address of the page table entry: PTBR + (page number) * 4
- Address of the page: [PTBR + (page number) * 4)]
  - Brackets indicate indirection
- Physical address: [PTBR + (page number) * 4)] + offset

- Let’s now split the page table into pages...

Hierarchical

- A page in the system is 4KiB
- Page table entries are 4 bytes
- Therefore, in a page, we can store 2^12 / 2^2 = 2^10 page table entries
- We call such a page a “page table page”
- Since we need a total of 2^32 / 2^12 = 2^20 page table entries, we need 2^20 / 2^10 page table pages
- Let’s see this on a picture...

Hierarchical

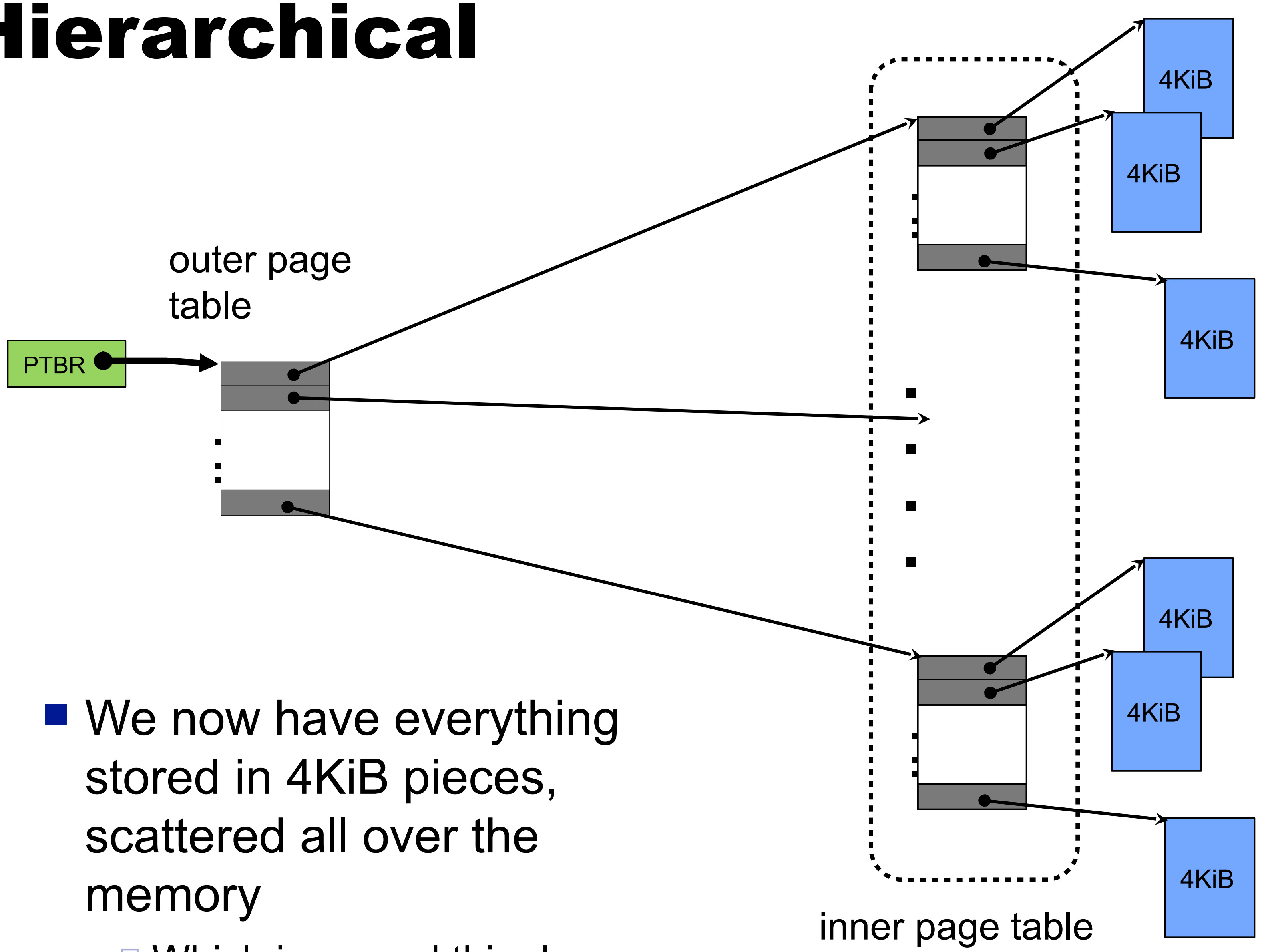


2^10 page table pages
page table page w/ 2^10 entries
4KiB
4KiB
4KiB
page table page w/ 2^10 entries
4KiB
4KiB
4KiB

Hierarchical

- Now we just need to keep track of all page table pages
    - We must have pointers to them
- Conveniently, we have 2^10 page table pages
- The address of a page is stored with 4 bytes
- So we can store the addresse of all the page table pages .... in a page!
    - 2^10 * 4 = 2^12 = 4KiB = page size
- Yet another picture...

Hierarchical



- We now have everything stored in 4KiB pieces, scattered all over the memory
  - Which is a good thing!

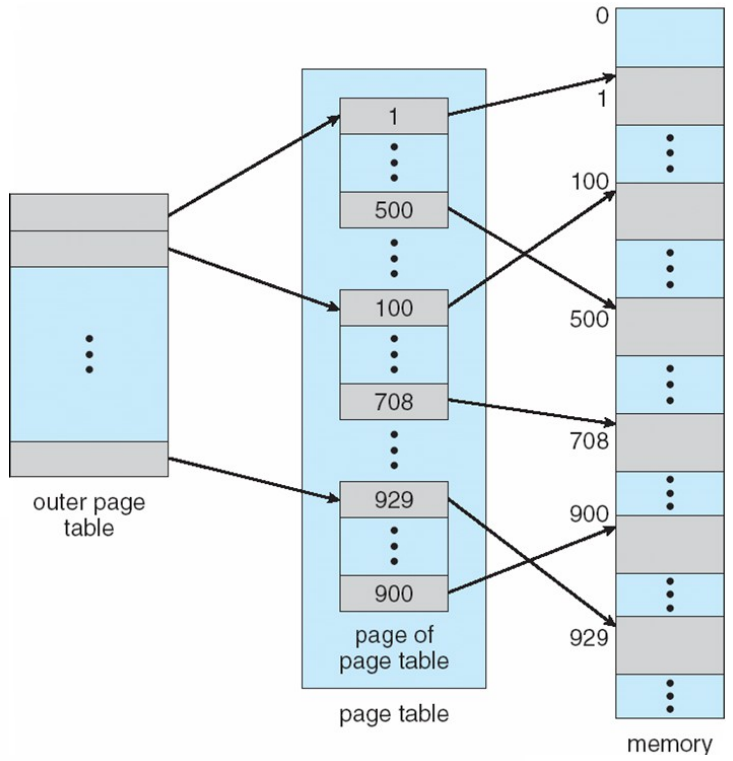
Hierarchical

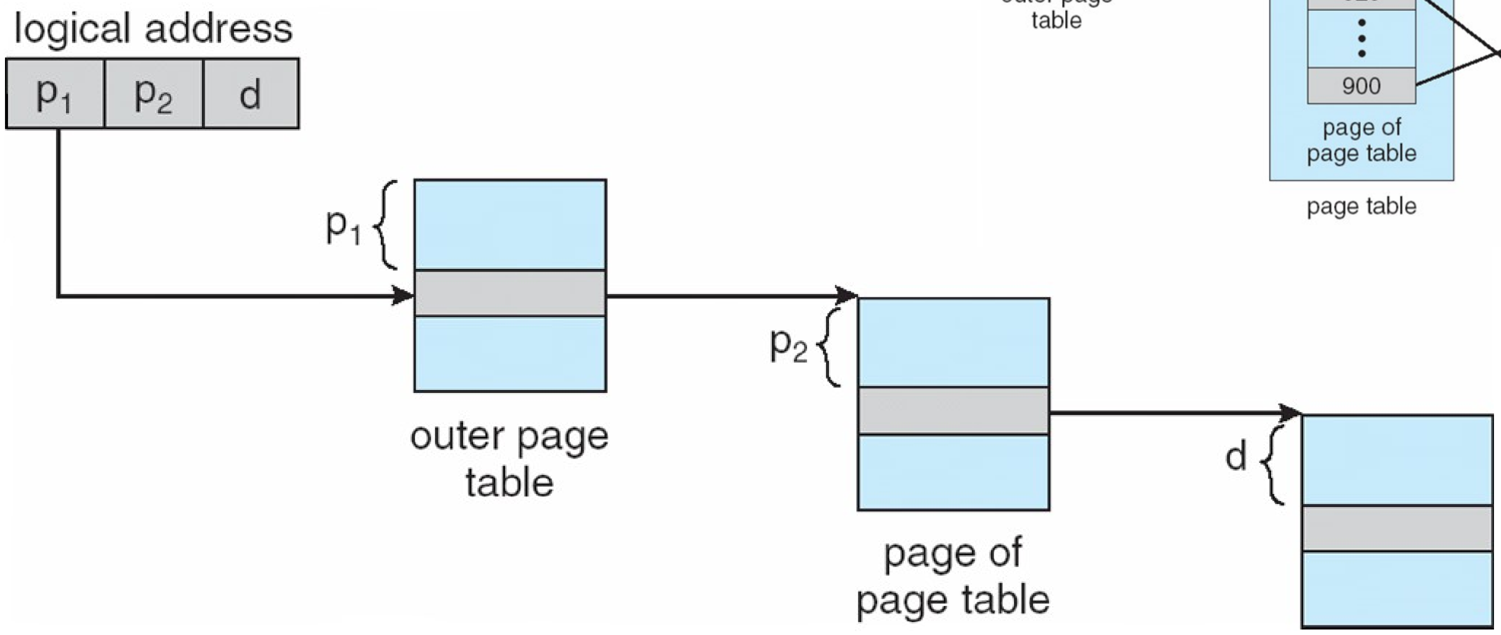
p1	p2	offset

- Logical to Physical translation
  - Address of the outer page table entry: PTBR + 4 * p1
  - Address of the page table page: [PTBR + 4 * p1]
  - Address of the page entry: [PTBR + 4 * p1] + 4 * p2
  - Address of the page: [[PTBR + 4 * p1] + 4 * p2]
  - Physical address: [[PTBR + 4 * p1] + 4 * p2] + offset
- In this case, p1 is 10-bit and p2 is 10-bit, which works out very well
  - outer page table = 1 page
  - inner page table = 2^10 pages
  - 10+10+12 = 32

Hierarchical Page Tables

- Figures from the book





Hierarchical Page Tables

- With 64-bit addresses, we're still in trouble
  - 4KiB page size
  - 4KiB inner page table page size
  - Remains: 64 - 12 - 10 = 42 high bits
  - Outer page table size: 2^{42}*4 = 2^{44} bytes = 16TB !!
- And this is still assuming that page table entries are 4 bytes!
  - They are likely 8 bytes, in which case:
    - 4KiB page size
    - 4KiB inner page table page size with 2^{9} entries
    - Remains 64 - 12 - 9 = 43 bits
    - Outer page table size: 2^{43} * 8 = 64TB!
- So we need a deeper hierarchy, for instance adding one level

second outer page

p1	p2	p3	offset
32	10	10	12

Hierarchical Page Tables

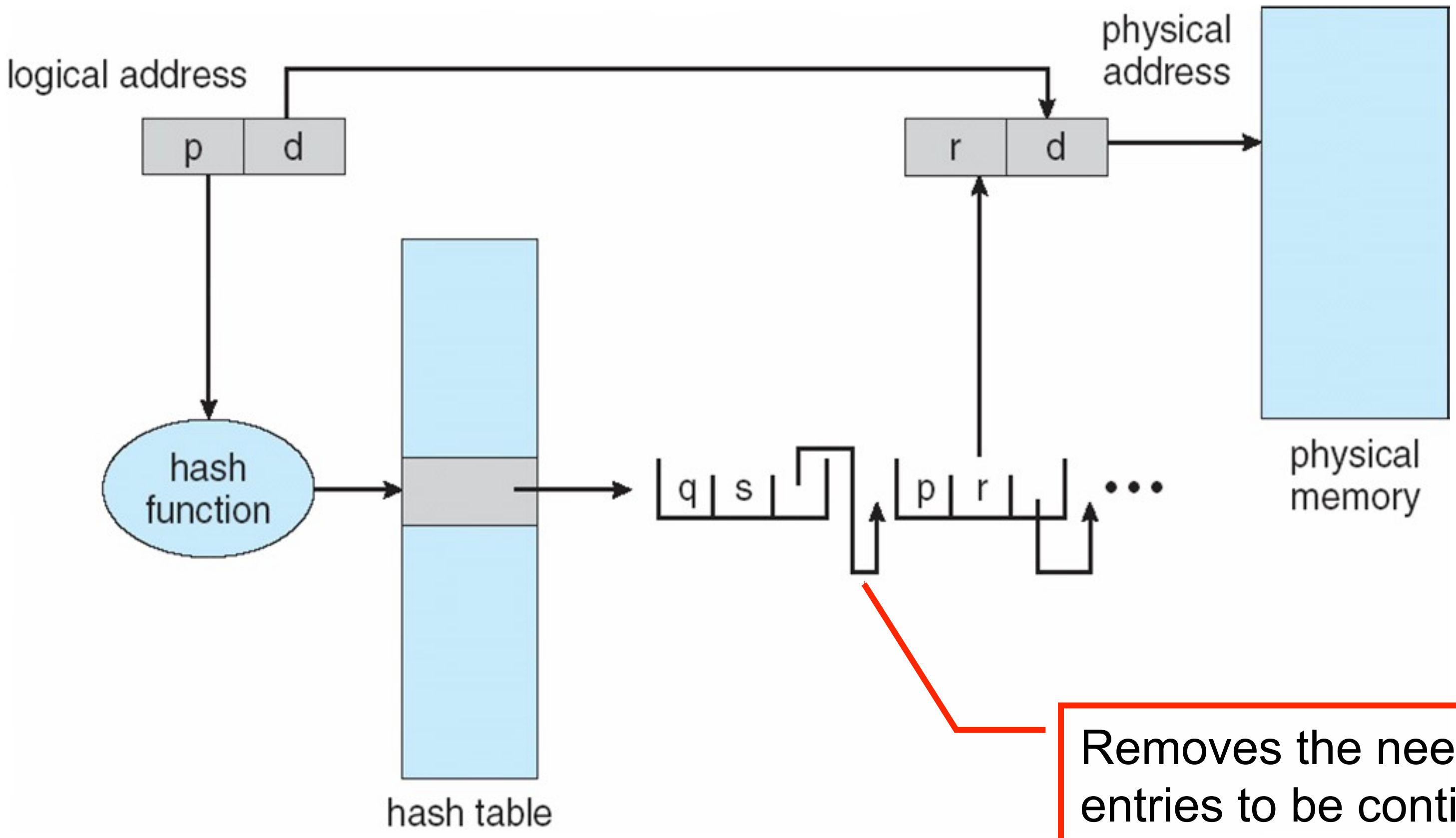
- Even with 3 levels we need 2^{34} = 16GiB for the second outer page table!
  - Again assuming 4-byte page table entries
- One could have many more levels
- But with each level there is one extra indirection, and thus extra overhead

- Conclusion: Hierarchical page tables become memory hogs for large address spaces with small pages
  - e.g., 64-bit architectures that would support processes that use large address spaces with 4KiB pages

Hashed Page Tables

- Pick a maximum (desirable) size for the page table, say N
- Come up with a hash function that’s applied to a logical page number and returns a number from 0 to N-1
- Structure the page table as a hash table using this hash function
  - Logical page numbers that hash to the same value have their entries stored in a linked list in a hash table entry

Hashed Page Table

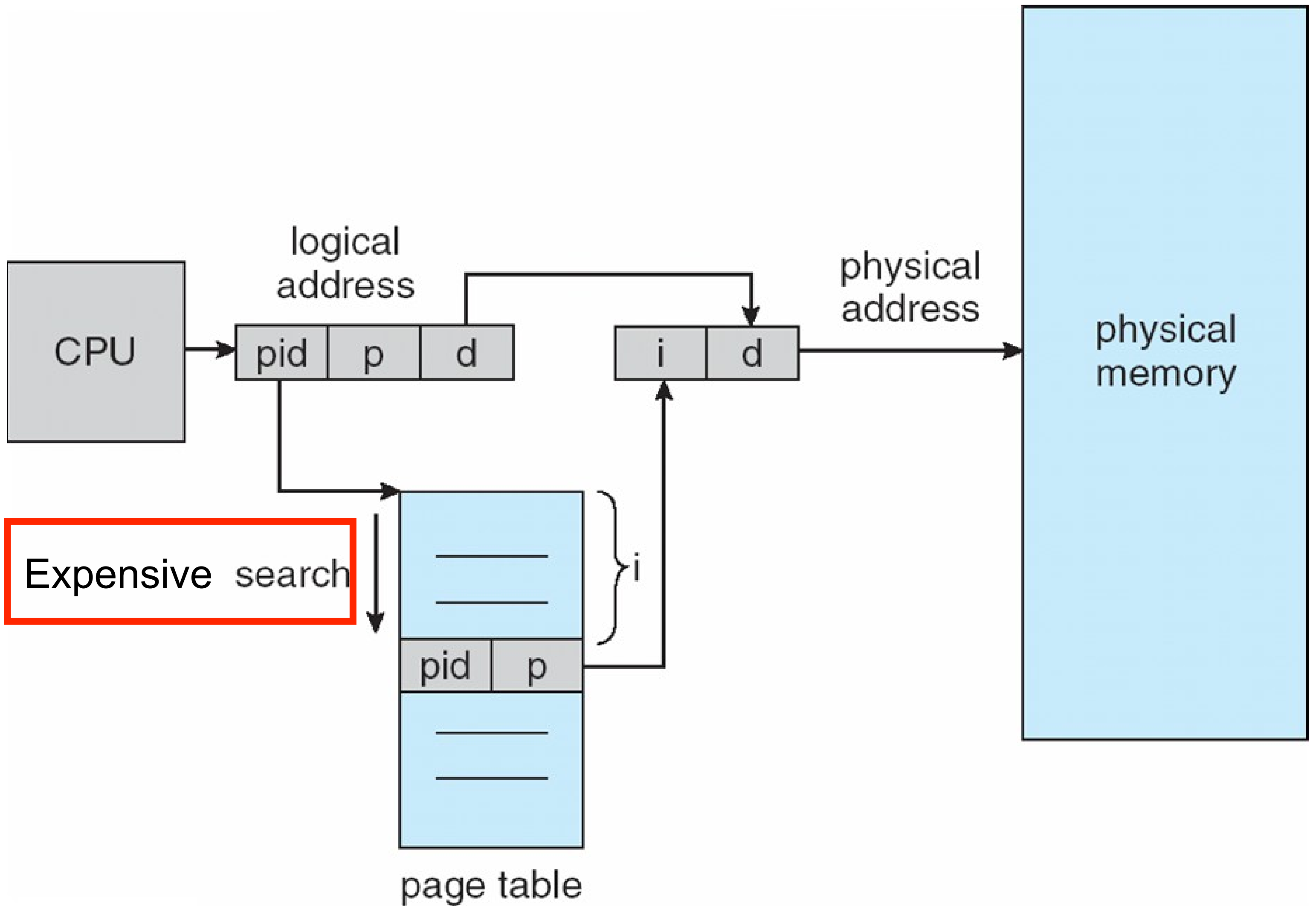


Removes the need for entries to be contiguous in memory (at the expense of much more overhead)

Inverted Page Tables

- Having one page table per process is fast
  - The logical page number is an index in the page table
    - Or multiple indices in a hierarchical scheme
  - Hashing is still pretty fast
- But it consumes a lot of memory
  - Especially if page tables are complete, and with with valid/invalid bits to invalidate unused entries
- The alternative: inverted page table
  - One table for all processes
  - One entry per physical memory frame
  - Each entry is: ASID + logical page number
- As opposed to knowing for each process where its logical pages are, now for which physical frame we know the process that owns it an what logical page it corresponds to

Inverted Page Tables



Inverted Page Tables

- Memory consumption is much reduced
- The time for a lookup is much larger
  - But the TLB helps
  - And one could use a hashed inverted page table
- One difficulty: how does one implement shared memory pages?

- Conclusion: you can have good time complexity, good space complexity, but not both

What’s done in practice

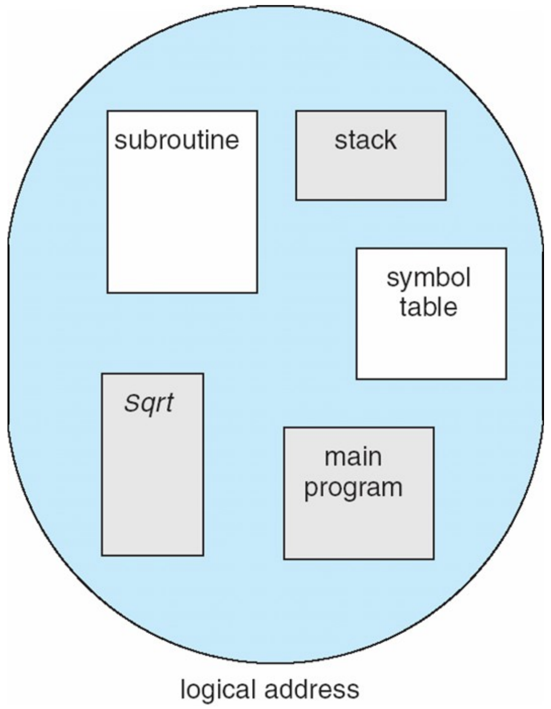
- Paging involves both the hardware (e.g., for splitting up address bits into the relevant pieces) and the kernel (e.g., to manage page tables)
- Most 32-bit architectures have a memory unit that assumes 2-level page tables
  - high bits are called the “directory” (index in the outer page table)
  - “middle” bits are called the “table” (index in an inner page table page)
  - low bits are called the offset (index in a page)
- 64-bit architectures add more levels to the hierarchy
  - Most use 3 levels, but x86_64 uses 4 levels
- Linux uses hierarchical page tables
  - Adapts the number of levels based on what the hardware provides
- How do we deal with page table being too large?
  - Systems are configured to limit a process’ maximum address space
  - Page tables grow “on demand” as the process’ memory footprint increases
    - The deeper the hierarchy, the bigger the saving in memory space
  - Large page sizes are becoming more popular (4KiB pages is really small on a system with 32GiB RAM)
- Some systems have used inverted page tables (e.g., IBM RS/6000, PowerPC), but hierarchical page tables seem to dominate at this point

Segmentation

- Segmentation is a way to structure logical memory
  - According to a typical user’s view of memory
- The Goal:
  - To allow address spaces to be broken up into logically independent address spaces
  - Makes sharing easier
  - Makes protection easier
    - e.g., one can prevent modifying code at runtime
    - e.g., one can prevent executing data at runtime
- Important: doesn’t have to replace paging
  - Paging is about not having big contiguous memory segments that lead to fragmentation
  - One can have both segmentation and paging
- In what follows we assume no paging to make figures simpler

Segmentation

- The address space is a set of (dynamically growing/shrinking) pieces
- The programmer doesn’t really care which piece comes after which other piece
- But the programmer cares that the pieces don’t overlap



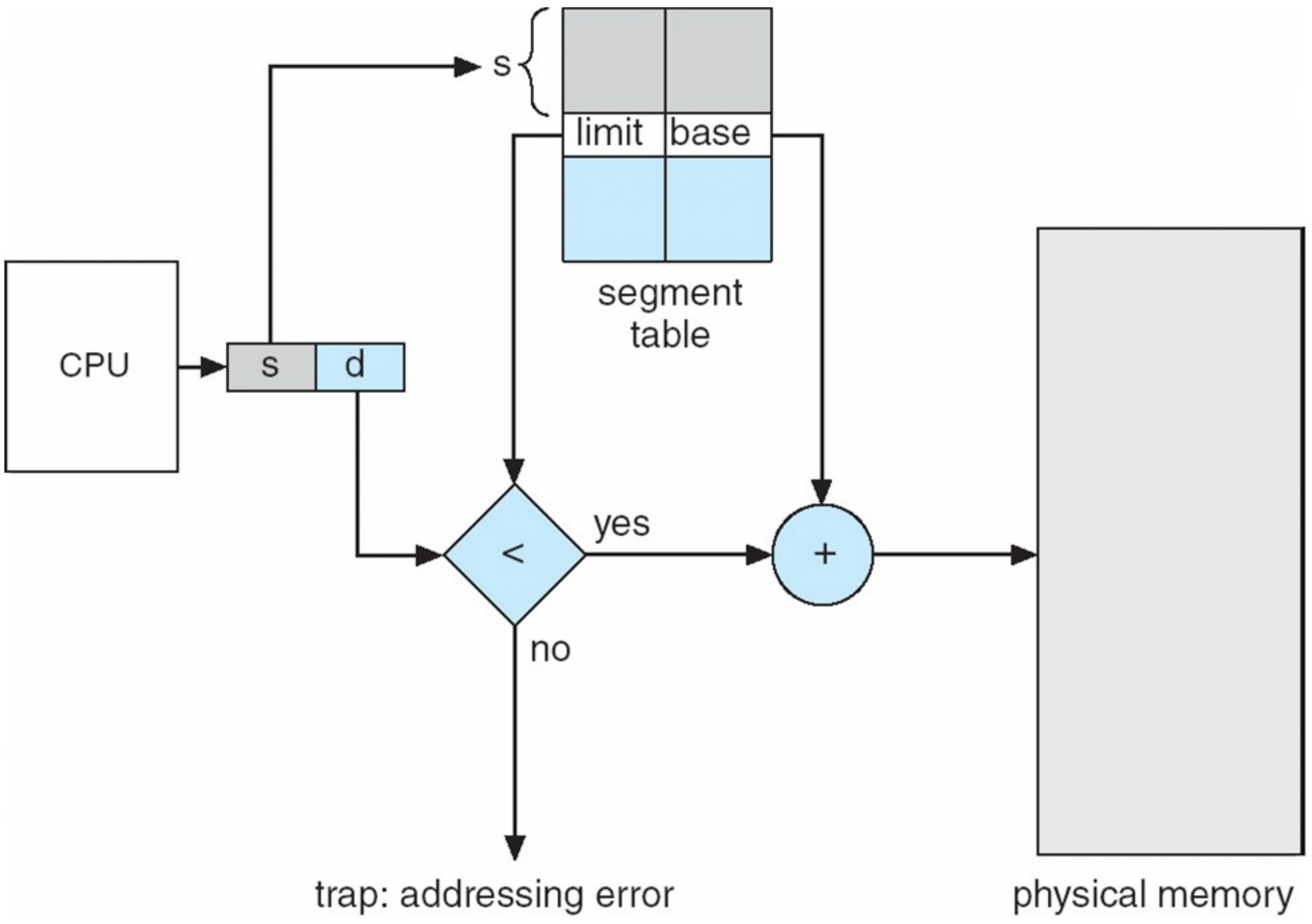
Segmentation

- The logical address space is a collection of segments
- A logical address is then:
  - A segment number
  - An offset within the segment
- The compiler handles segments and logical addresses produced contain appropriate segment numbers
  - If you write asseMiBly you may have to deal with segments
  - In ICS312 we use gcc to compile a driver, which freed us from dealing with segments by hand
- Typical segments used by a C compiler
  - text
  - data
  - heap
  - stacks
  - std C library

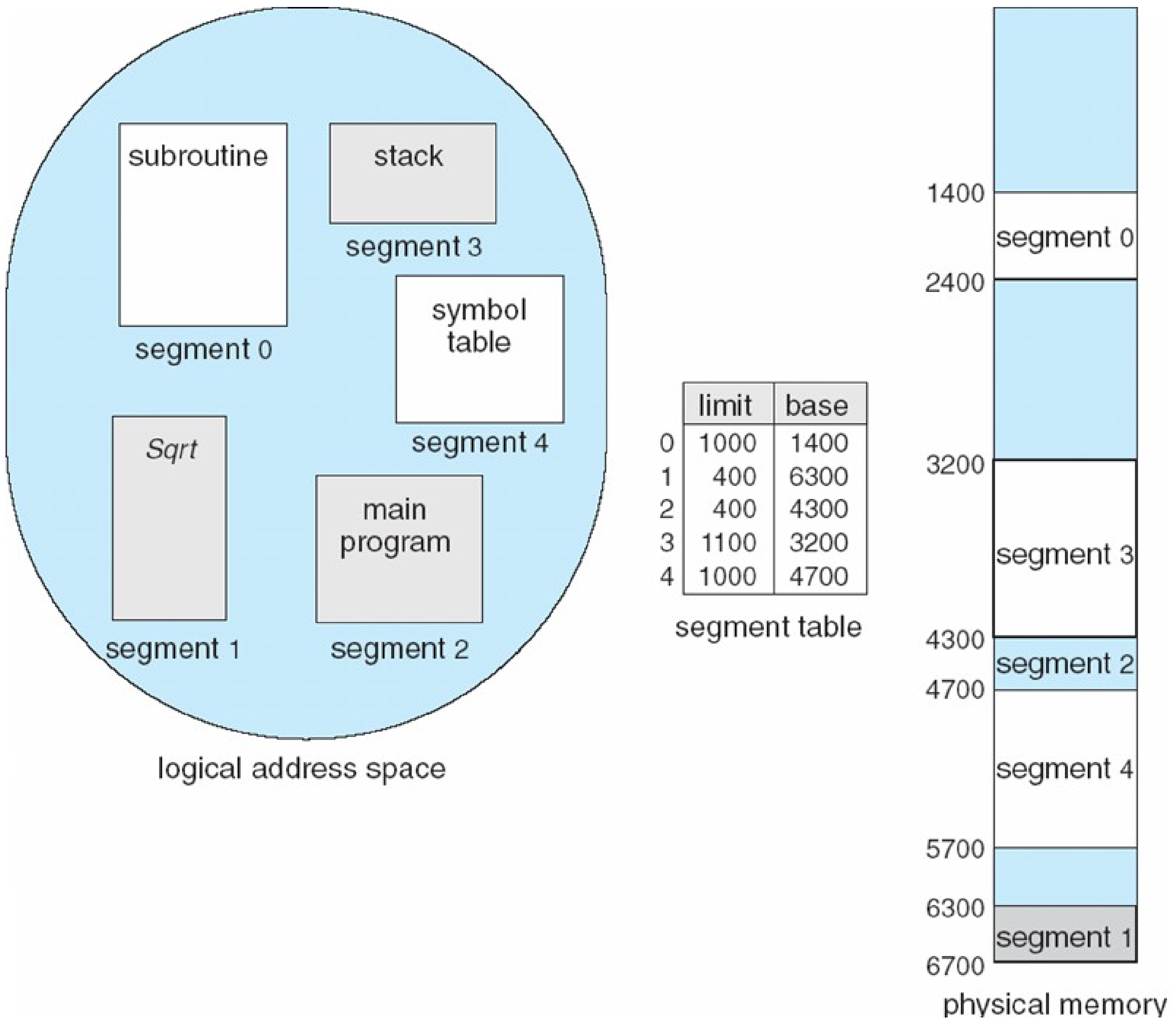
Segmentation

- We need a segment table
  - One entry per segment number
  - Each entry has
    - base: starting address of the segment
    - limit: the length of the segment
- The segment table is stored in memory
  - A Segment-Table Base Register (STBR)
    - Points to the segment table's address
  - A Segment-Table Length Register (STLR)
    - Gives the length of the segment table
    - Makes it easy to detect an invalid segment number

Segmentation Hardware



Segmentation Example

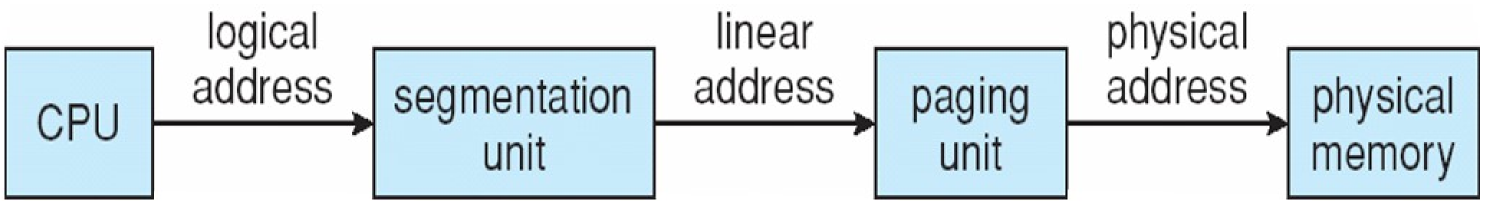


	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Sharing and Protection

- Segments make it simple to implement several sharing and protection mechanisms
- Segment-level R/W/X protection bits
  - The code segment is “R/X”
    - Read and execute
  - The data segment is “R/W”
    - Read and write
- Segment-level sharing
  - Share entire code segment
  - Share entire global data segment

Example: The IA 32/64



- The Intel architecture provides both segmentation and paging
- A logical address is transformed into a linear address via segmentation
    - logical address = (segment selector, offset)
- A linear address is transformed into a physical address via paging
    - linear address = (page number 1, page number 2, offset)
- All details are in Section 8.7

Conclusion

- Memory Management is at the boundary between Computer Architecture and Operating Systems
- Summary
  - Swapping
    - To have more processes than could fit in main memory
  - Paging
    - To avoid external fragmentation in main memory
    - Various page table structures
      - To trade off memory space with speed
    - Hierarchical pages used often in practice
  - Segmentation
    - To allow more convenient protection and sharing
  - Segmentation and Paging can be used together
    - e.g., in the Intel Pentium