Virtual Memory and Paging (2)

ICS332

Operating Systems

Henri Casanova (henric@hawaii.edu)

Paging is great but...

- The previous set of lecture notes ends with all the benefits of paging
- But there are some challenges / problems
- Two big problems:
- **Problem #1:** Paging has extra overhead
- **Problem #2:** Page tables can be very large
- Let’s understand these problems and come up with solutions

Paging Overhead

- Each address coming out of the CPU is virtual
- Address translation (from virtual to physical) has to be performed for **EVERY** address issued by the CPU
- **The page table is in RAM and will be accessed very frequently!**

- When a new process is dispatched to the CPU, the dispatcher loads a special register with the address of the beginning of the process's page table: the Page Table Base Register (PTBR)
- This makes it fast to switch between page tables at each context switch, but does not speed up translation

- Because of paging **the memory access time is doubled**: 1) Access an entry in the page table; 2) Based on that entry access the physical address
- We just made our RAM twice as slow :(
  - And it was already annoyingly slow!

Paging Locality

However!

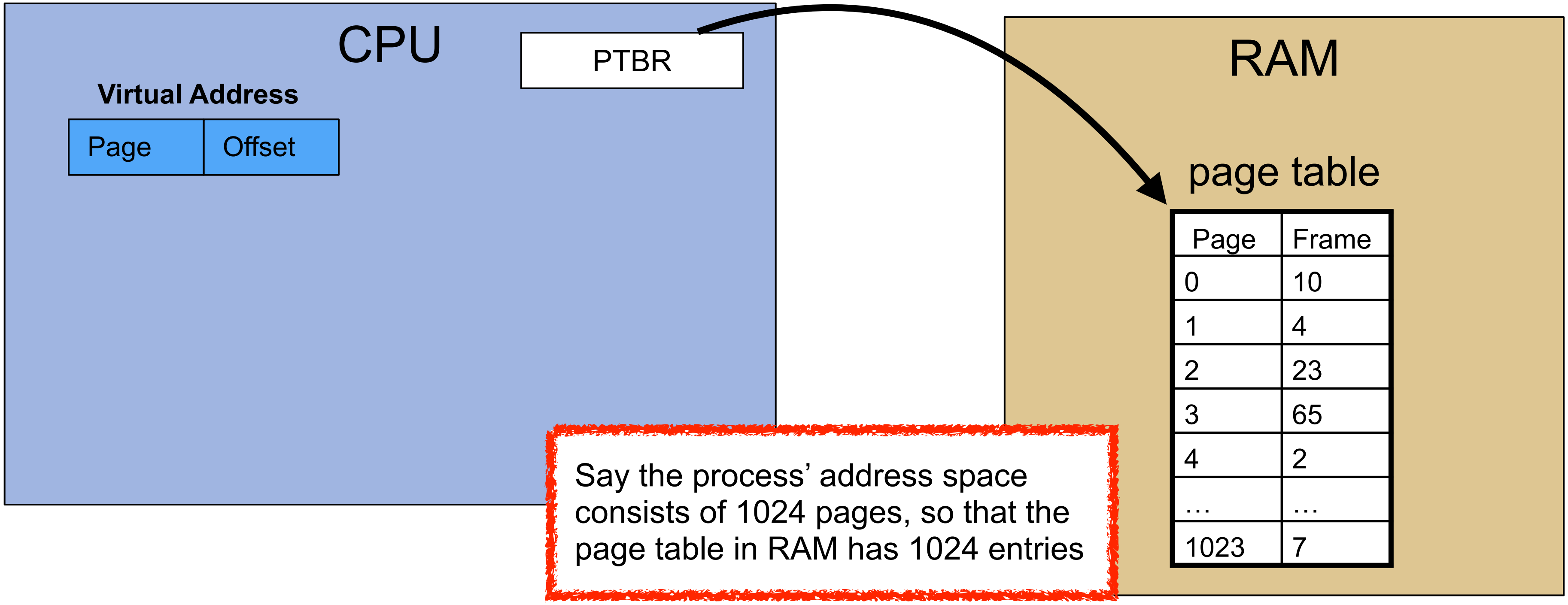
- Temporal locality: repeated access to the same memory location
  - e.g., counter++
  - counter is accessed over and over
  - the same page is accessed over and over
- Spatial locality: repeated access to nearby memory locations
  - e.g., a[i] = a[i-1] + a[i-2]
  - all three array elements are very likely in the same page
  - the same page is accessed over and over
- Therefore, as a process executes, the address translation requests often look like:
  - Give me the Frame Number for Page 12
  - Give me the Frame Number for Page 12 again
  - Give me the Frame Number for Page 12 again
  - and again, and again...
- We should REMEMBER (i.e., cache) previous translation results!!

The TLB

- Caching of previous translations is done by a hardware component called…
- The Translation Lookaside Buffer (TLB)
  - Each entry in the TLB is a <key, value> pair
  - You give it a key
  - The key is compared in parallel with all stored keys
  - If the key is found, then the associated value is returned

The TLB

- Caching of previous translations is done by a hardware component called…
- The Translation Lookaside Buffer (TLB)
  - Each entry in the TLB is a <key, value> pair
  - You give it a key
  - The key is compared in parallel with all stored keys
  - If the key is found, then the associated value is returned

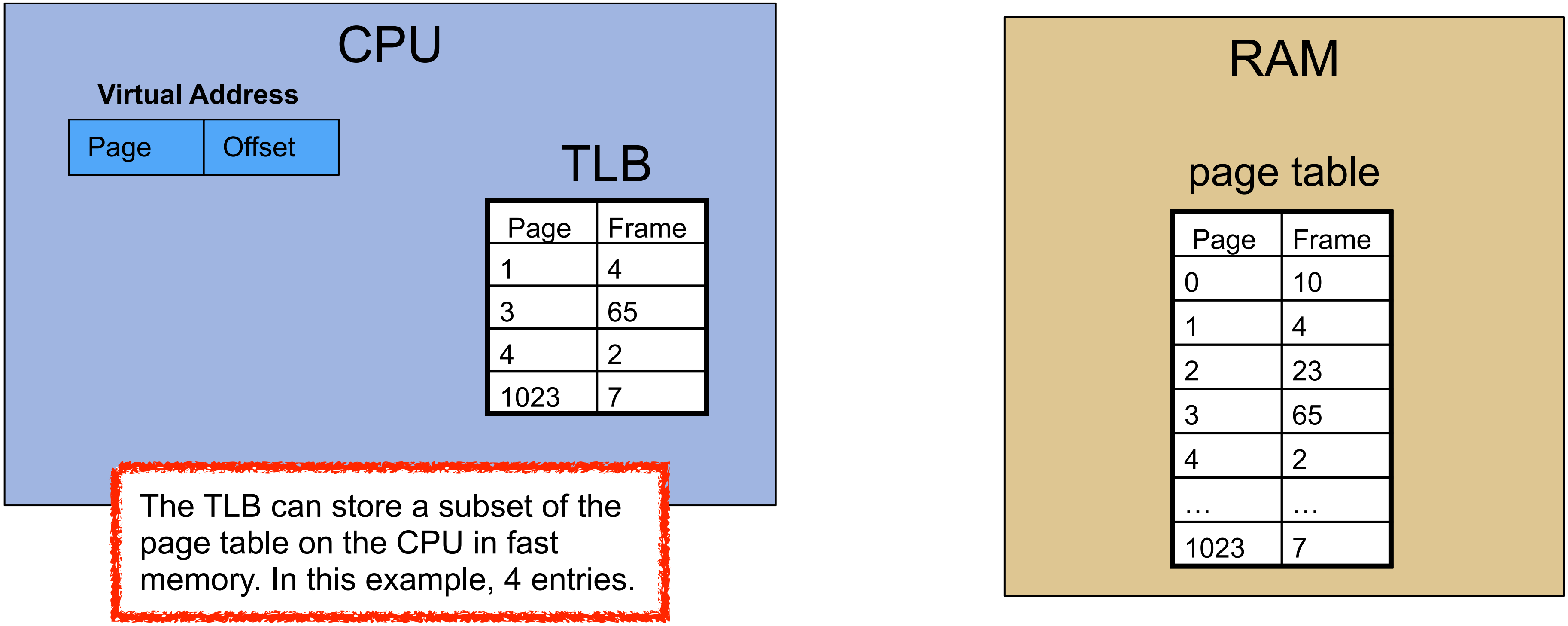


Page	Frame
0	10
1	4
2	23
3	65
4	2
…	…
1023	7

Say the process' address space consists of 1024 pages, so that the page table in RAM has 1024 entries

The TLB

- Caching of previous translations is done by a hardware component called…
- The Translation Lookaside Buffer (TLB)
  - Each entry in the TLB is a <key, value> pair
  - You give it a key
  - The key is compared in parallel with all stored keys
  - If the key is found, then the associated value is returned



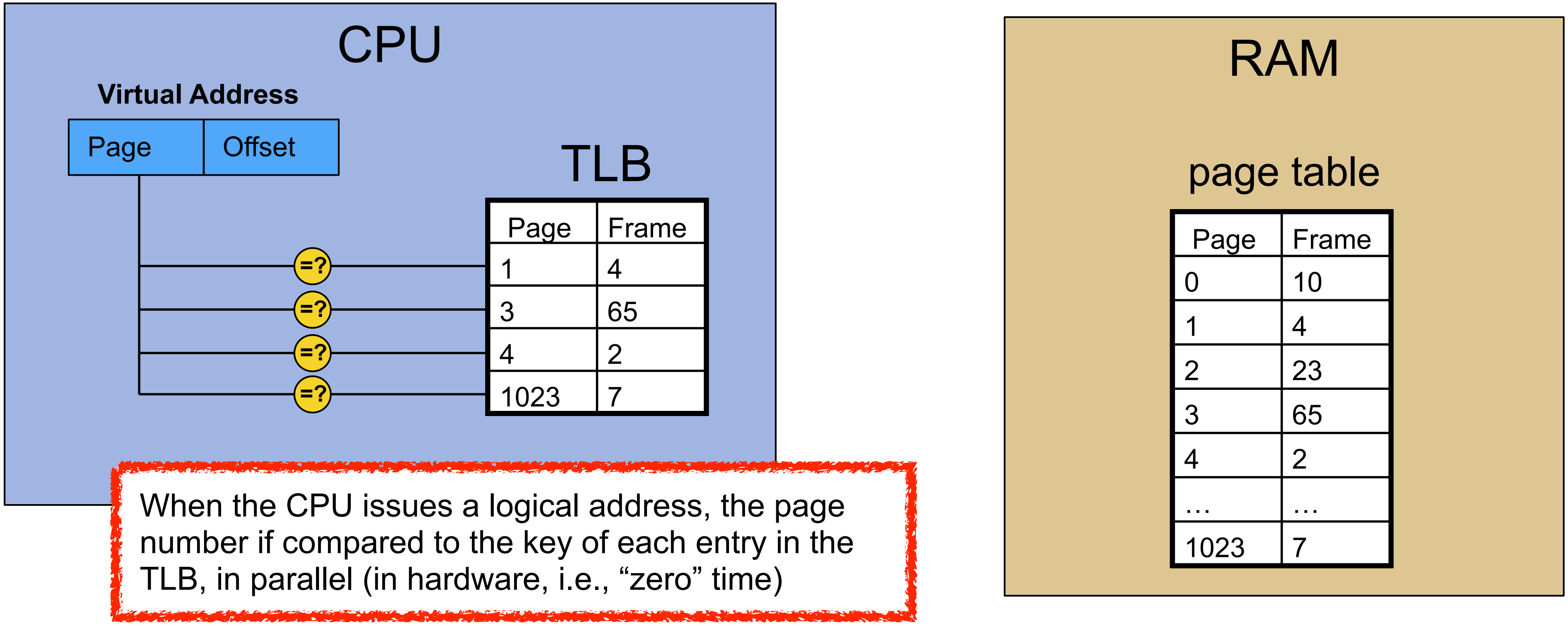
Page	Frame
1	4
3	65
4	2
1023	7

Page	Frame
0	10
1	4
2	23
3	65
4	2
…	…
1023	7

The TLB can store a subset of the page table on the CPU in fast memory. In this example, 4 entries.

The TLB

- Caching of previous translations is done by a hardware component called…
- The Translation Lookaside Buffer (TLB)
  - Each entry in the TLB is a <key, value> pair
  - You give it a key
  - The key is compared in parallel with all stored keys
  - If the key is found, then the associated value is returned



TLB

Page	Frame
1	4
3	65
4	2
1023	7

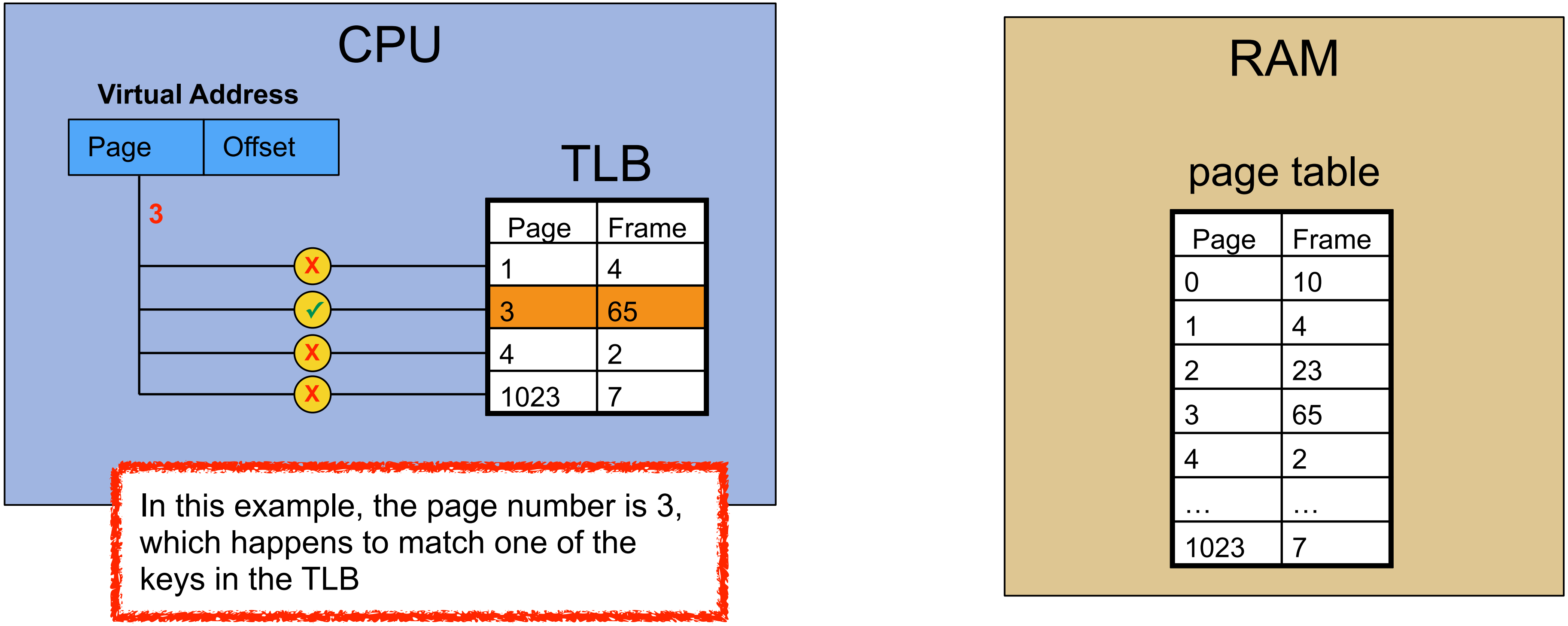
page table

Page	Frame
0	10
1	4
2	23
3	65
4	2
…	…
1023	7

When the CPU issues a logical address, the page number if compared to the key of each entry in the TLB, in parallel (in hardware, i.e., “zero” time)

The TLB

- Caching of previous translations is done by a hardware component called…
- The Translation Lookaside Buffer (TLB)
  - Each entry in the TLB is a <key, value> pair
  - You give it a key
  - The key is compared in parallel with all stored keys
  - If the key is found, then the associated value is returned



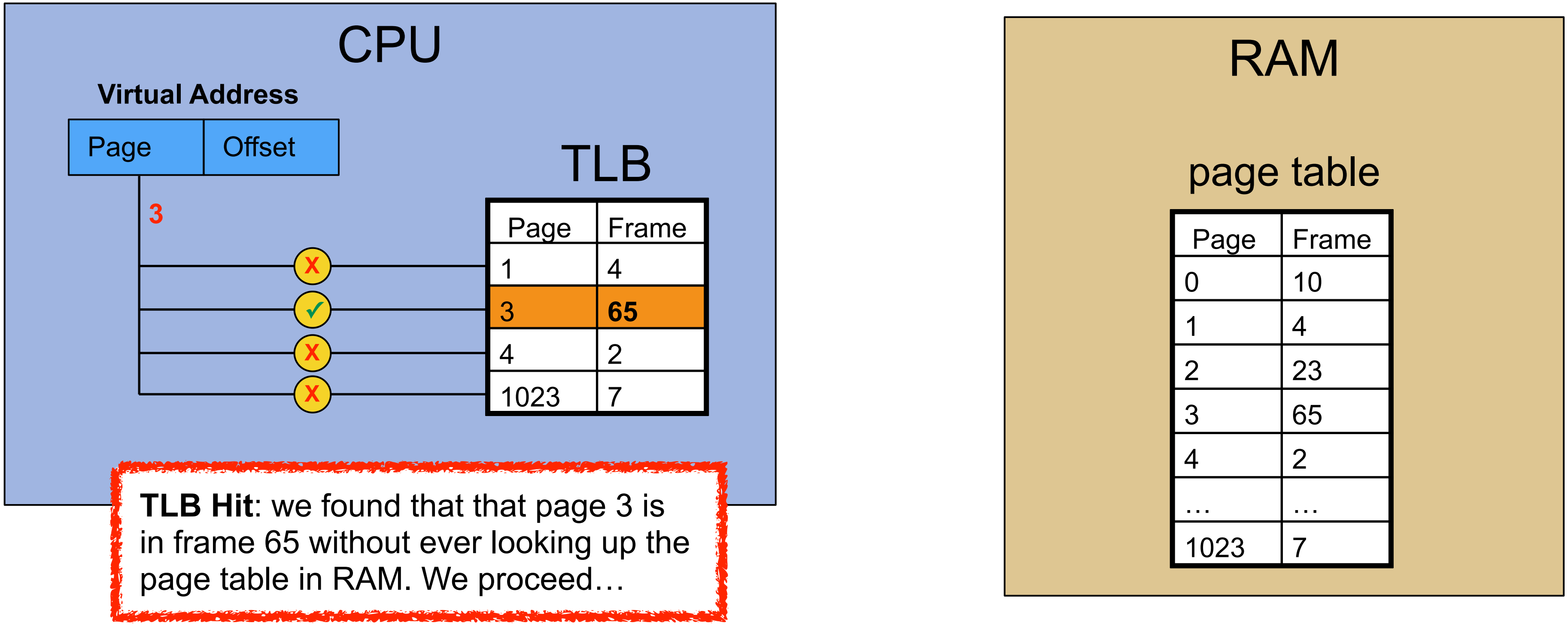
Page	Frame
1	4
3	65
4	2
1023	7

Page	Frame
0	10
1	4
2	23
3	65
4	2
…	…
1023	7

In this example, the page number is 3, which happens to match one of the keys in the TLB

The TLB

- Caching of previous translations is done by a hardware component called…
- The Translation Lookaside Buffer (TLB)
  - Each entry in the TLB is a <key, value> pair
  - You give it a key
  - The key is compared in parallel with all stored keys
  - If the key is found, then the associated value is returned



TLB

Page	Frame
1	4
3	**65**
4	2
1023	7

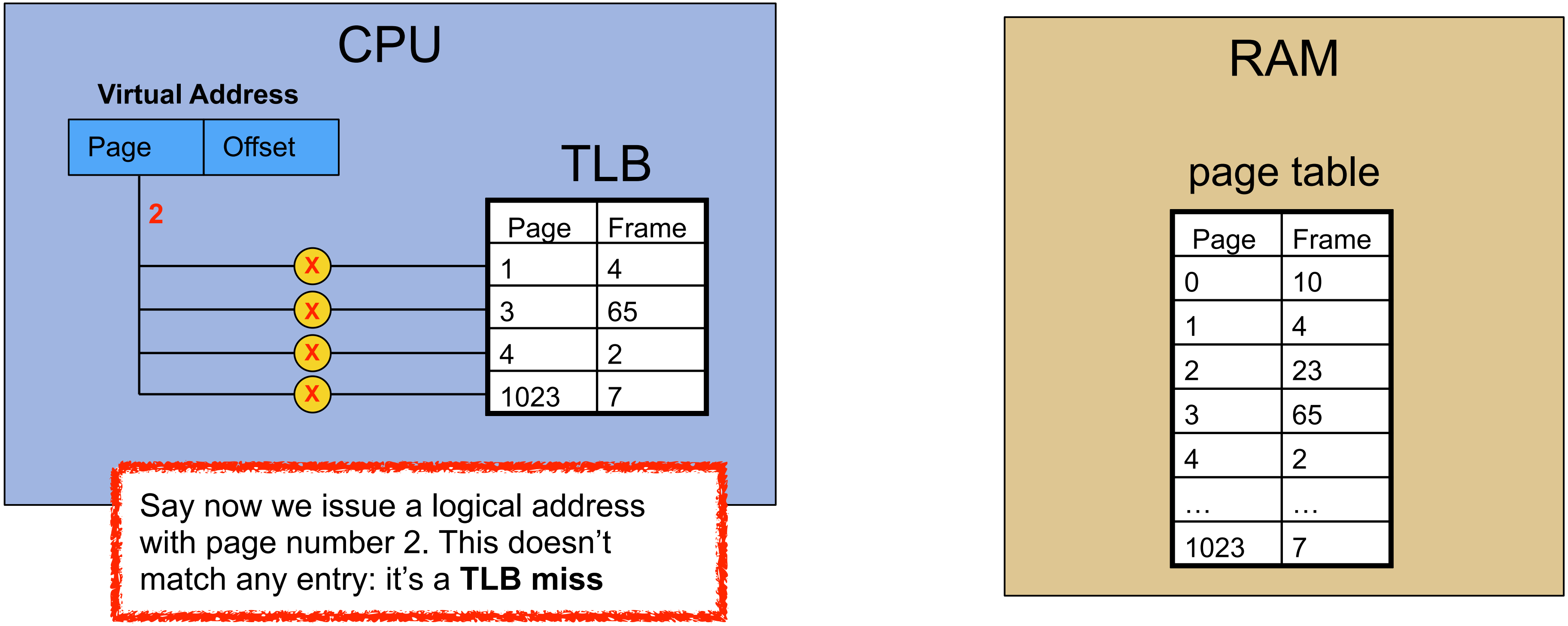
page table

Page	Frame
0	10
1	4
2	23
3	65
4	2
…	…
1023	7

TLB Hit: we found that that page 3 is in frame 65 without ever looking up the page table in RAM. We proceed…

The TLB

- Caching of previous translations is done by a hardware component called…
- The Translation Lookaside Buffer (TLB)
  - Each entry in the TLB is a <key, value> pair
  - You give it a key
  - The key is compared in parallel with all stored keys
  - If the key is found, then the associated value is returned



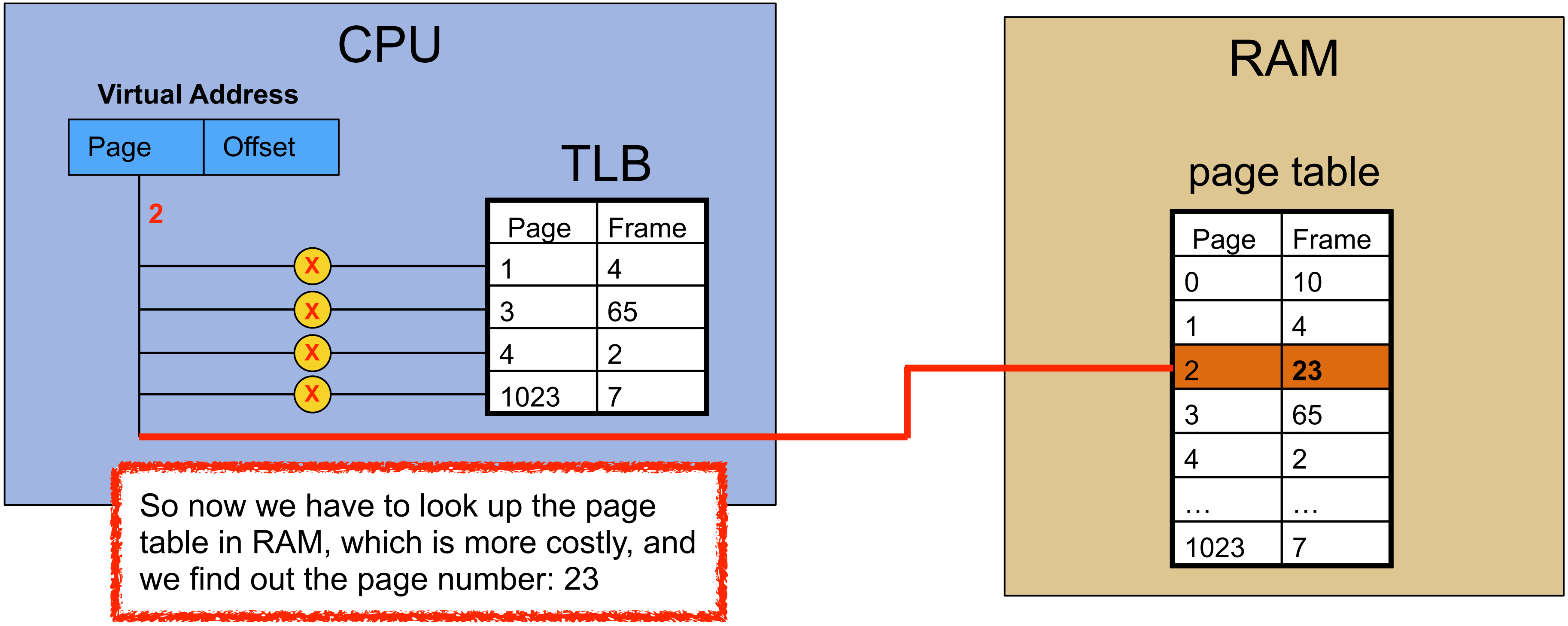
Page	Frame
1	4
3	65
4	2
1023	7

Page	Frame
0	10
1	4
2	23
3	65
4	2
…	…
1023	7

Say now we issue a logical address with page number 2. This doesn't match any entry: it's a **TLB miss**

The TLB

- Caching of previous translations is done by a hardware component called…
- The Translation Lookaside Buffer (TLB)
  - Each entry in the TLB is a <key, value> pair
  - You give it a key
  - The key is compared in parallel with all stored keys
  - If the key is found, then the associated value is returned



TLB

Page	Frame
1	4
3	65
4	2
1023	7

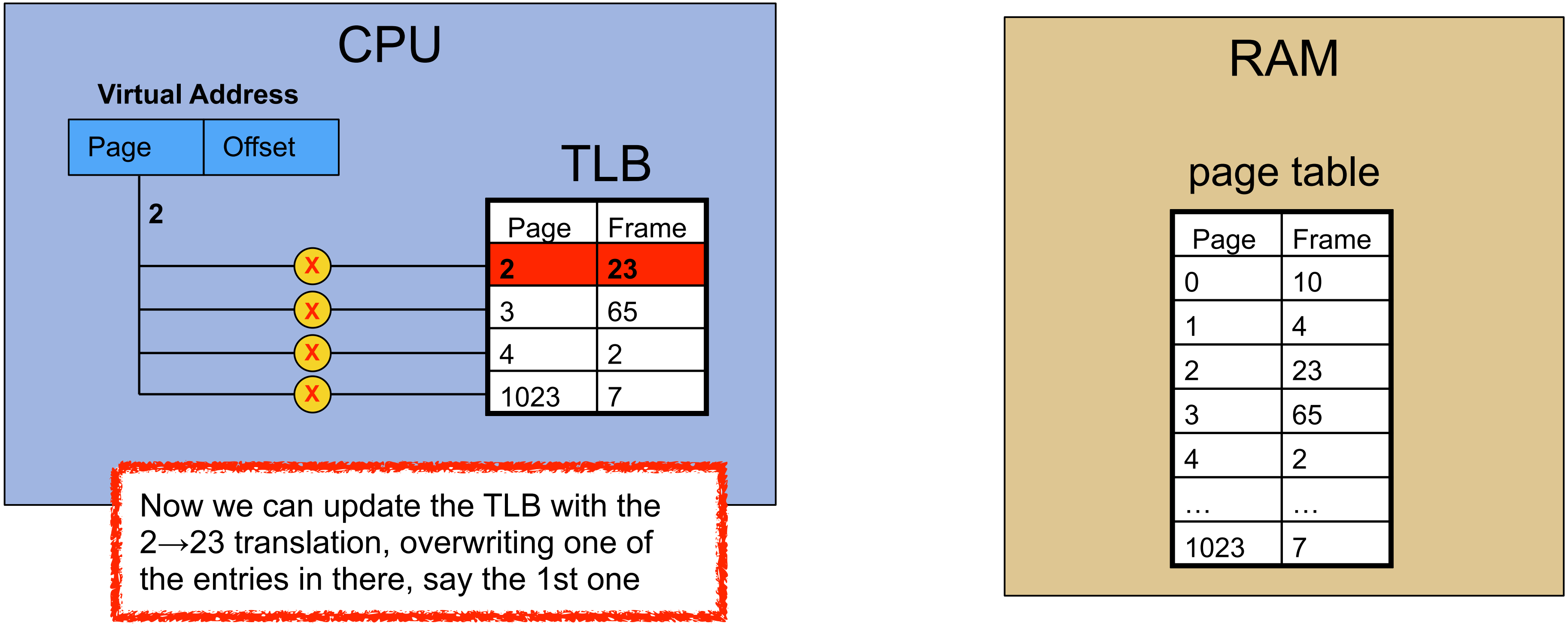
page table

Page	Frame
0	10
1	4
2	**23**
3	65
4	2
…	…
1023	7

So now we have to look up the page table in RAM, which is more costly, and we find out the page number: 23

The TLB

- Caching of previous translations is done by a hardware component called…
- The Translation Lookaside Buffer (TLB)
  - Each entry in the TLB is a <key, value> pair
  - You give it a key
  - The key is compared in parallel with all stored keys
  - If the key is found, then the associated value is returned



Page	Frame
2	23
3	65
4	2
1023	7

Page	Frame
0	10
1	4
2	23
3	65
4	2
…	…
1023	7

Now we can update the TLB with the 2→23 translation, overwriting one of the entries in there, say the 1st one

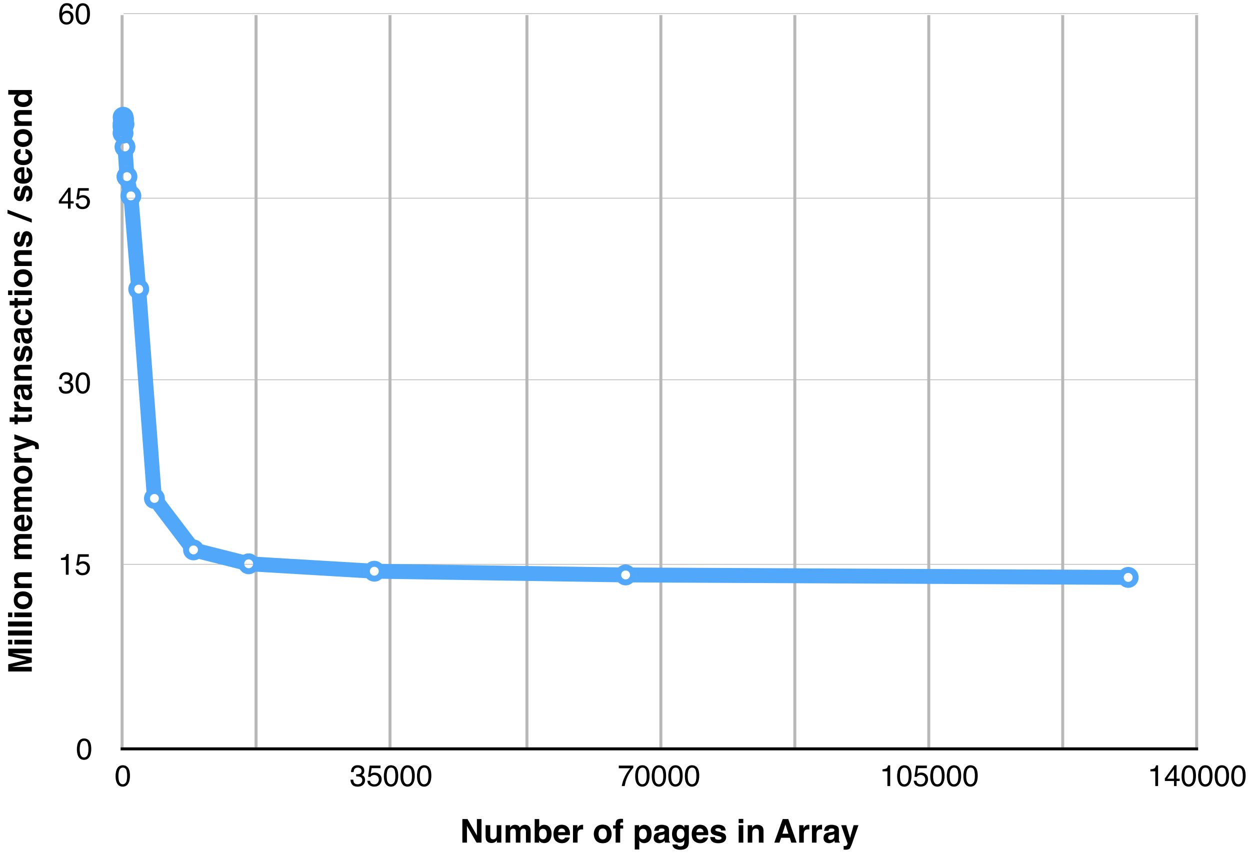
TLB Performance

- Typical TLB characteristics:
  - Contains 12 to 4,096 entries
  - Performance:
    - On a hit: less than 1 clock cycle
    - On a miss: 10-100 clock cycles
  - Typical miss rate: 0.01 - 1%

- A Replacement Policy must be defined to deal with what to do when the TLB is full:
  - Least Recently Used (LRU)? Random?
- Some TLBs allow for some entries to be un-evictable
  - e.g., kernel pages

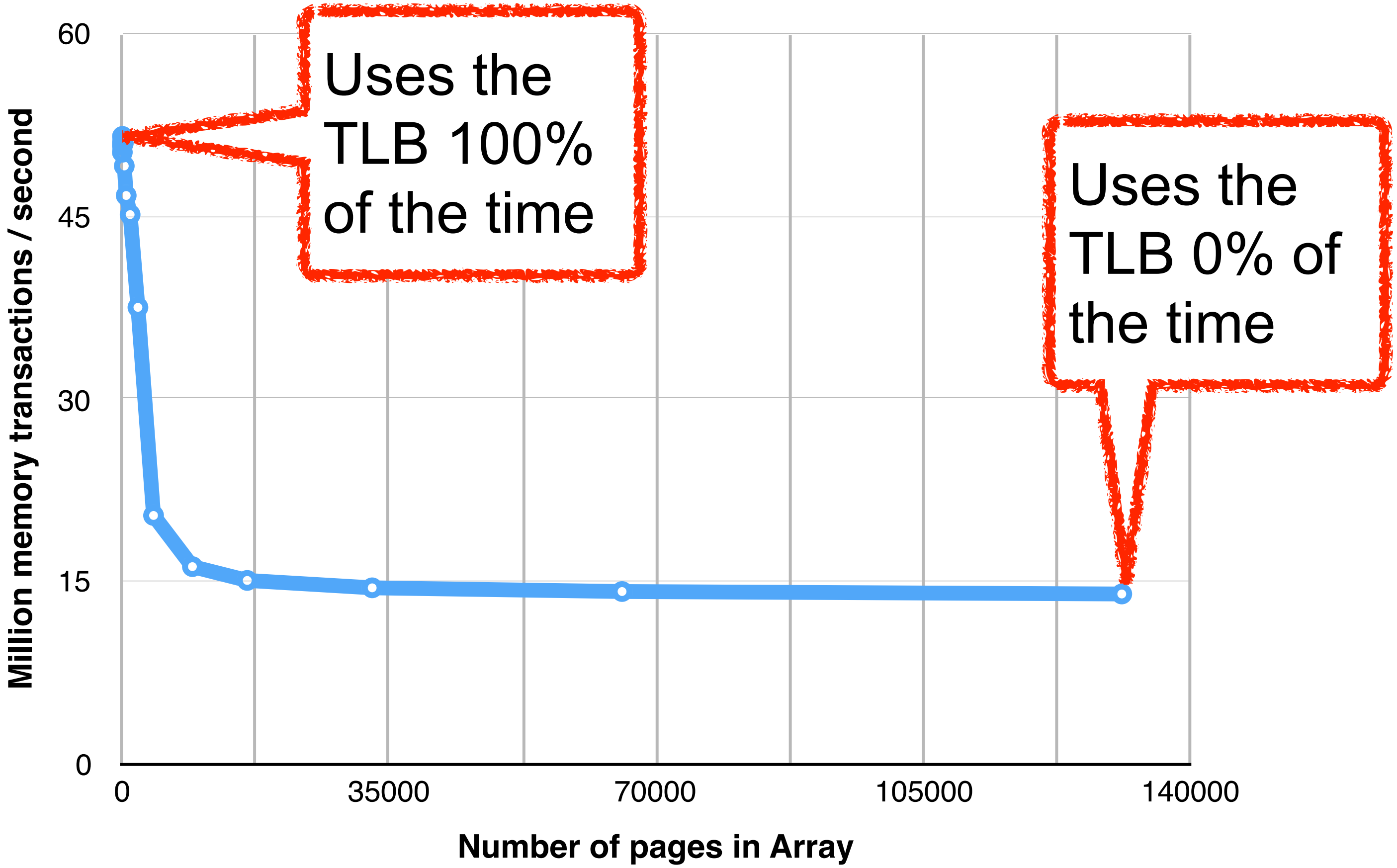
Experiment: How useful is the TLB?

- On the course web site, `tlb_stress.c`: a piece of code that allocates an array spanning multiple pages and then writes values at random locations (runs for some 20 seconds each time)



Experiment: How useful is the TLB?

- On the course web site, `tlb_stress.c`: a piece of code that allocates an array spanning multiple pages and then writes values at random locations (runs for some 20 seconds each time)



Uses the TLB 100% of the time

Uses the TLB 0% of the time

The TLB and Context-Switches

- What happens with the TLB on a context-switch?
- Wipe the TLB clean?
  - Page 7 of process A is not the same in the same frame as Page 7 of process B
  - Called a “TLB flush”
  - But perhaps unnecessary aggressive (the two processes could happily share the TLB)
  - So your machine doesn’t do TLB flushes these days
- ASIDs: Address-Space IDentifiers
  - Each TLB entry is annotated with a process identifier
  - The TLB can contain entries associated to multiple processes
  - Each lookup attempts to match entry ASIDs with the ASID of the current process (and if mismatch then it’s a TLB miss)

One down, one to go...

- **Problem #1:** Paging has extra overhead
- **Solution: Use a TLB**
  - Only works because our programs have locality "naturally"
  - Which is why caches work, and the TLB is just another kind of cache

- **Problem #2:** Page tables can be very large
- Let's look at this one now...

Page Table Structure

- We’ve shown page tables like this:

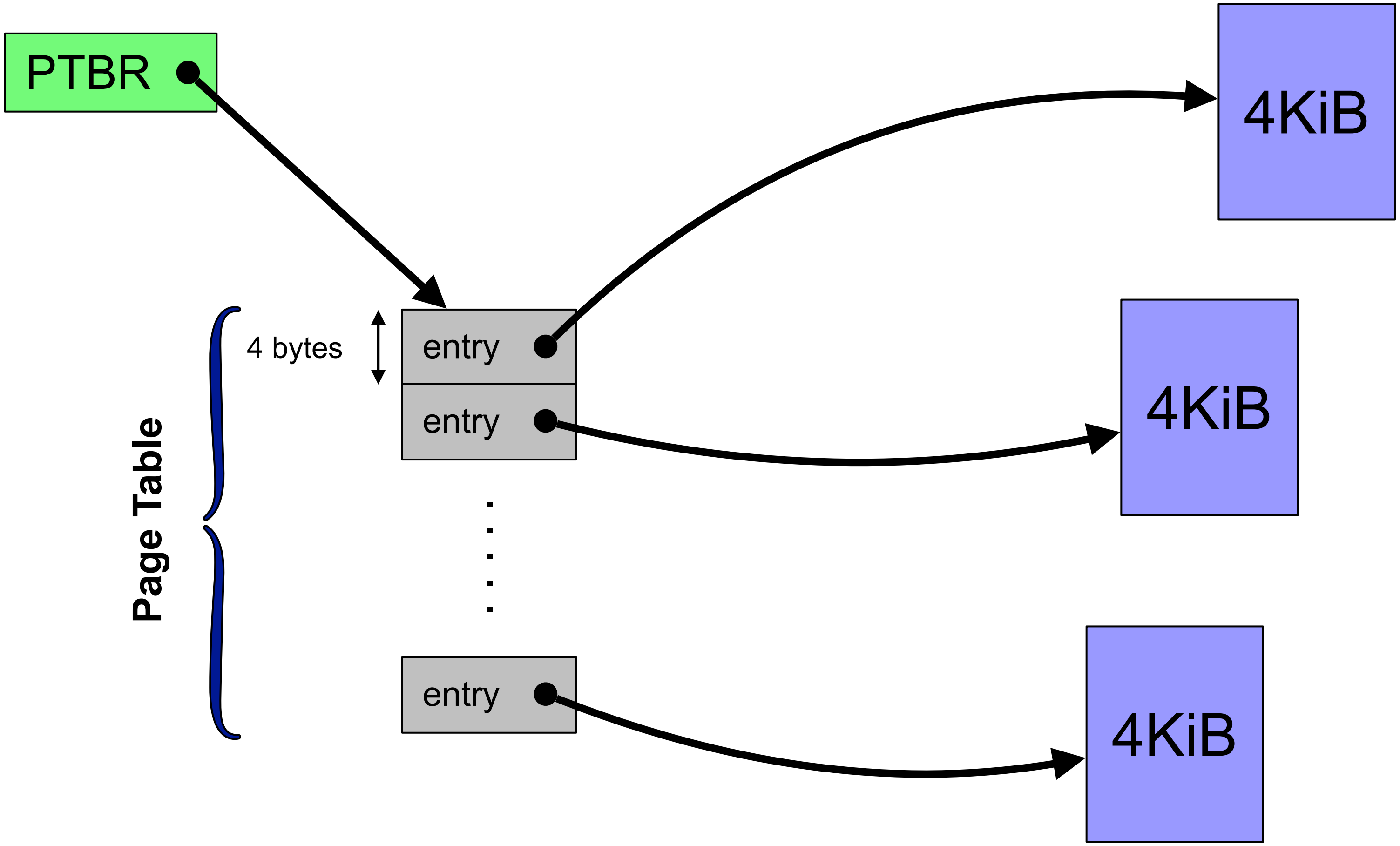
Page	Frame	Valid
0	1	✓
1	4	✓
2	3	✓
3	7	✓
4	xx	x
5	xx	x
6	xx	x
7	xx	x

- But once again, this is not quite right…

Page Table Entries

- One thing we haven't talked about yet: how many bits are needed for a page table entry?
  - I've shown the page table as just a table with numbers in it (and the valid bit)
  - But **the page table consumes space in RAM**
- Let us consider a system with 32-bit physical addresses, i.e., a 4GiB RAM
- The n-th entry in the page table is:
  - The physical frame number
  - A few bits (for now we've seen the valid bit, but there are other things - stay tuned)
  - The page number is just the index of the entry - see in a few slides
- Let us assume a page/frame size of 4 KiB = 2^{12} bytes
- We have $2^{32}/2^{12} = 2^{20}$ frames in RAM
- So the frame number can be encoded on 20 bits
- So a page table entry is 20 bits for the frame number, and then extra bits for "other stuff"
- Let's say that 32 bits = 4 bytes are used for each page table entry (which is typical for a 32-bit architecture)

Page Table Entries

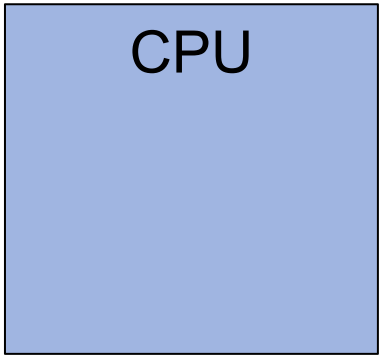


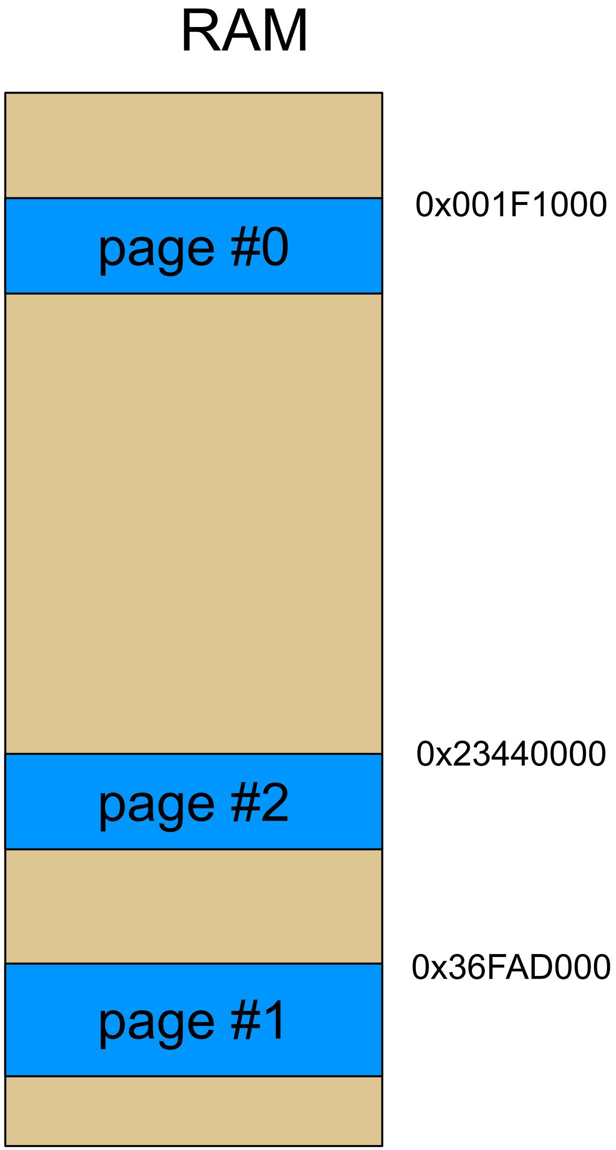
A Note on Page Table Structure

- The page table is just an array of entries
  - The entry for page 0 is the first element of the array
  - The entry for page 1 is the second element of the array
  - The entry for page i is the i-th element of the array
- So when we say “lookup an entry” we *don’t mean a search*
- Looking up the entry for page i means: **PTBR + i × entry size**

- For instance:
  - The PTBR contains address 0xAAAA0000
  - The page table entry size is 4-bytes
  - I want to “lookup” the entry for page 10
  - The entry for that page is at address 0xAAAA0028
    - (i.e., PTBR + 4 × 10)
  - We get the 4 bytes at that address
  - These bytes are: the frame number, the valid bit, other useful bits
- Let’s see an example that shows addresses…

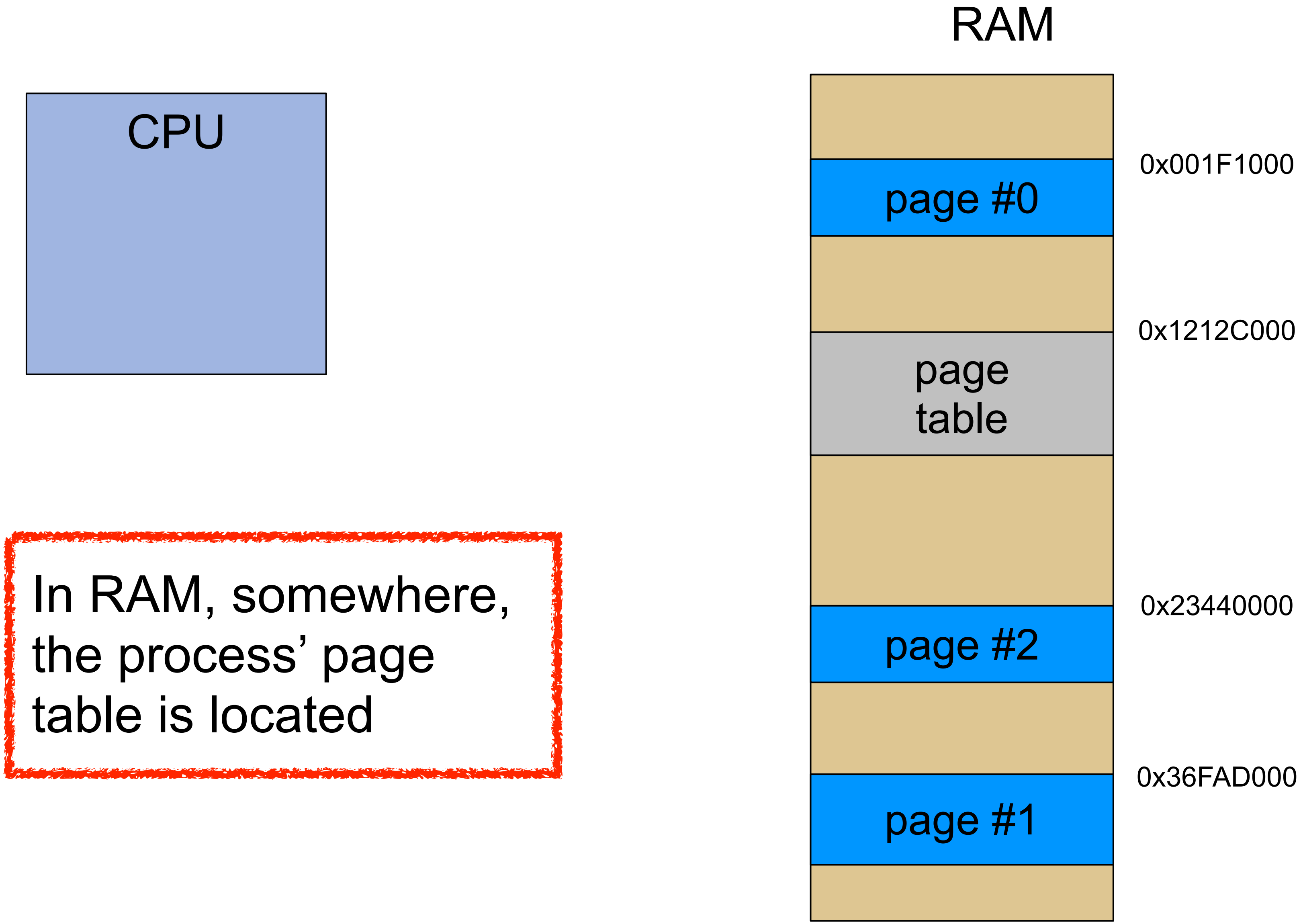
A Process in RAM





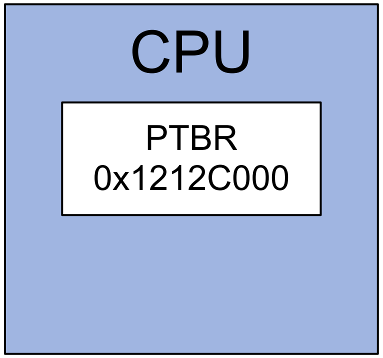
Let’s say we have a process with three pages in RAM

A Process in RAM

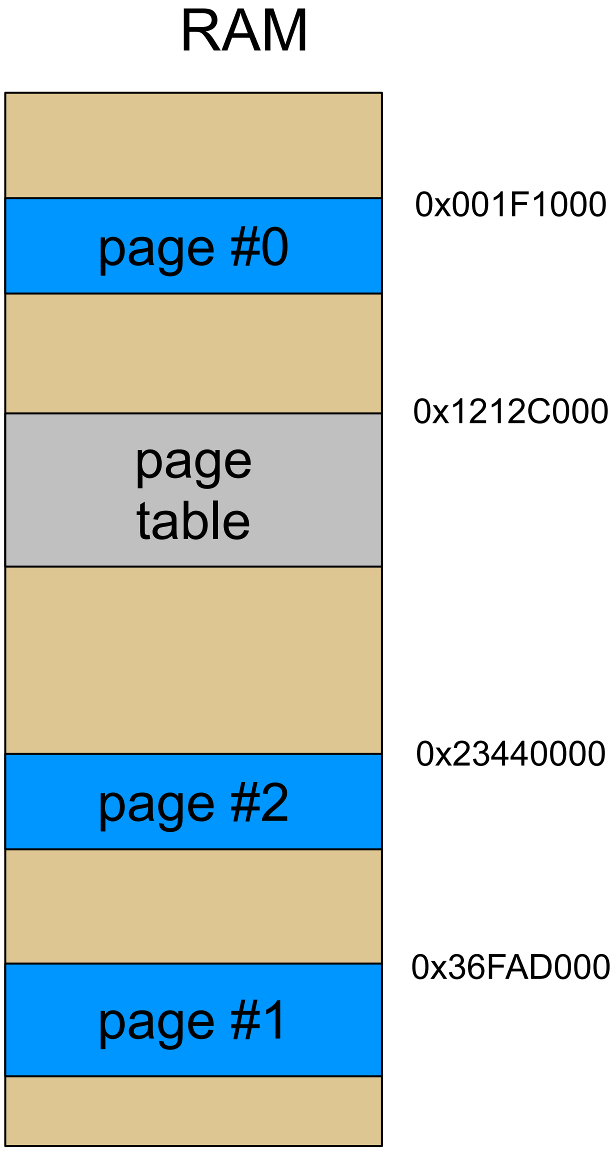


In RAM, somewhere, the process' page table is located

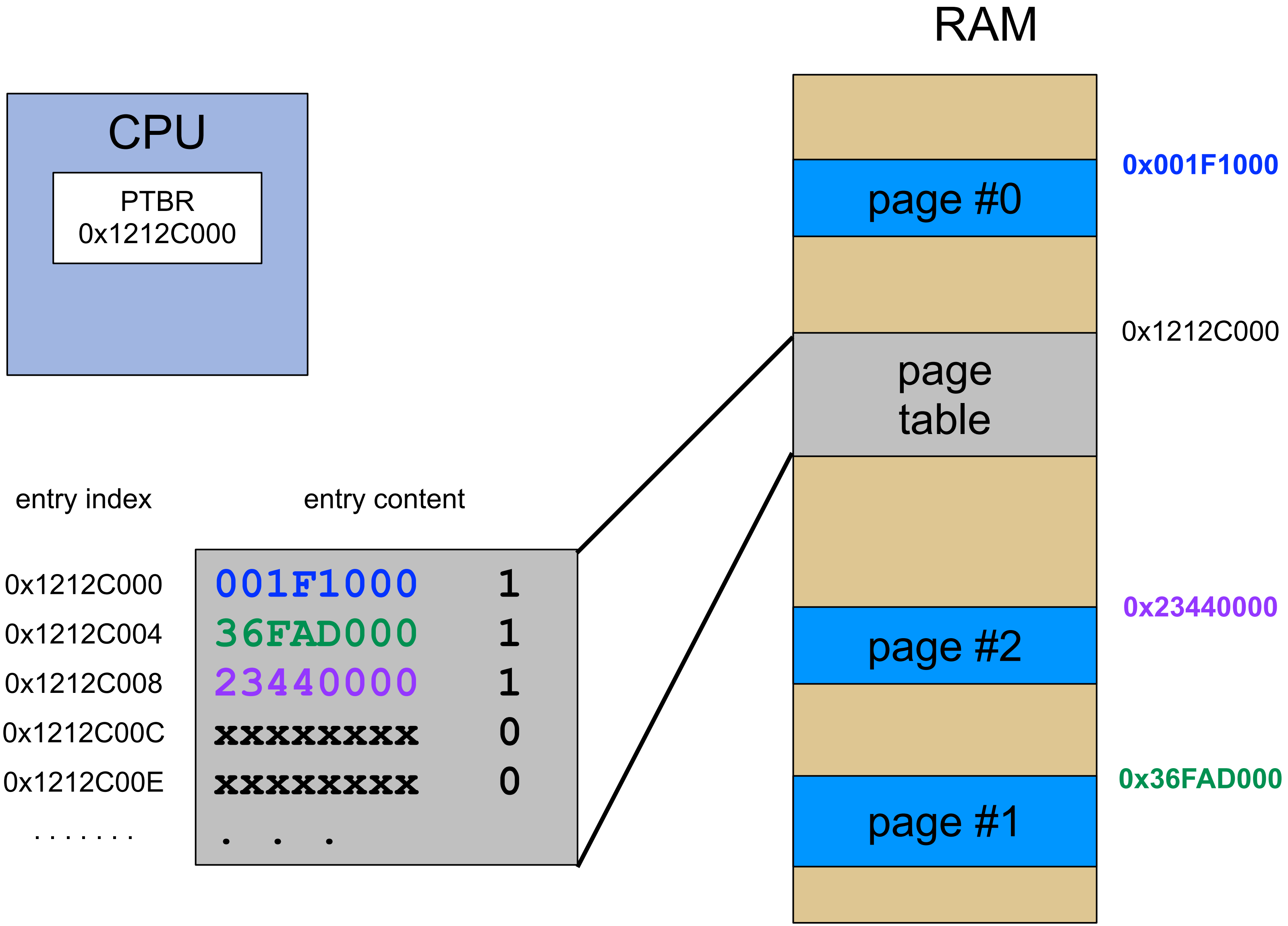
A Process in RAM



When the process was scheduled, the PTBR register was set to the address of the first entry in the page table



A Process in RAM



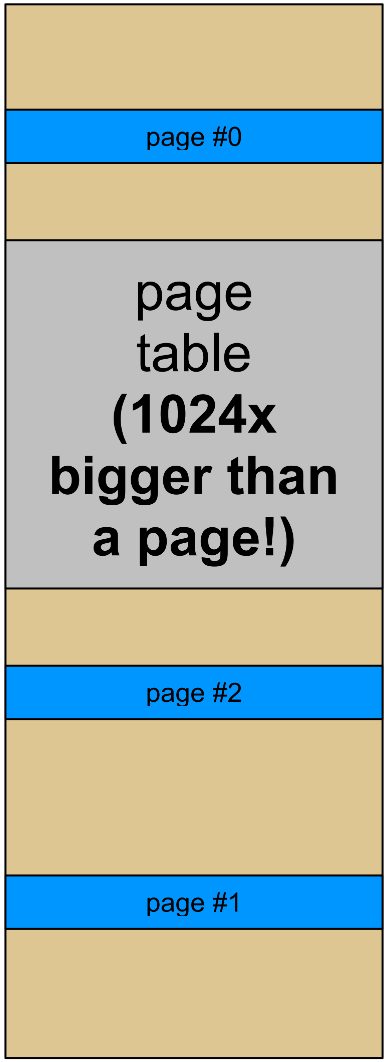
entry index	entry content	
0x1212C000	001F1000	1
0x1212C004	36FAD000	1
0x1212C008	23440000	1
0x1212C00C	xxxxxxxx	0
0x1212C00E	xxxxxxxx	0
. . . .	. . .	

Page Table Size

- So we have page table entries that are each 4 bytes
- Let's consider a process with a 4GiB address space
- This process has $2^{32}/2^{12} = 2^{20}$ pages
  - Because the page size is 2^{12} bytes
- The process' page table thus has 2^{20} entries
- Therefore, the page table takes up $2^{20} \times 2^{2} = 2^{22}$ bytes
  - which is 4 MiB
- So we need 4 MiB of contiguous RAM space to store the page table

Page Table Size

- So we have page table entries that are each 4 bytes
- Let’s consider a process with a 4GiB address space
- This process has $2^{32}/2^{12} = 2^{20}$ pages
  - Because the page size is 2^{12} bytes
- The process’ page table thus has 2^{20} entries
- Therefore, the page table takes up $2^{20} \times 2^{2} = 2^{22}$ bytes
  - which is 4 MiB
- So we need 4 MiB of contiguous RAM space to store the page table
- We need 4 MiB of contiguous RAM space!!!!



We have a Huge Problem

- We use paging to avoid large contiguous slabs of RAM
- To implement paging we use page tables
- But page tables are large contiguous slabs of RAM

We have a Huge Problem

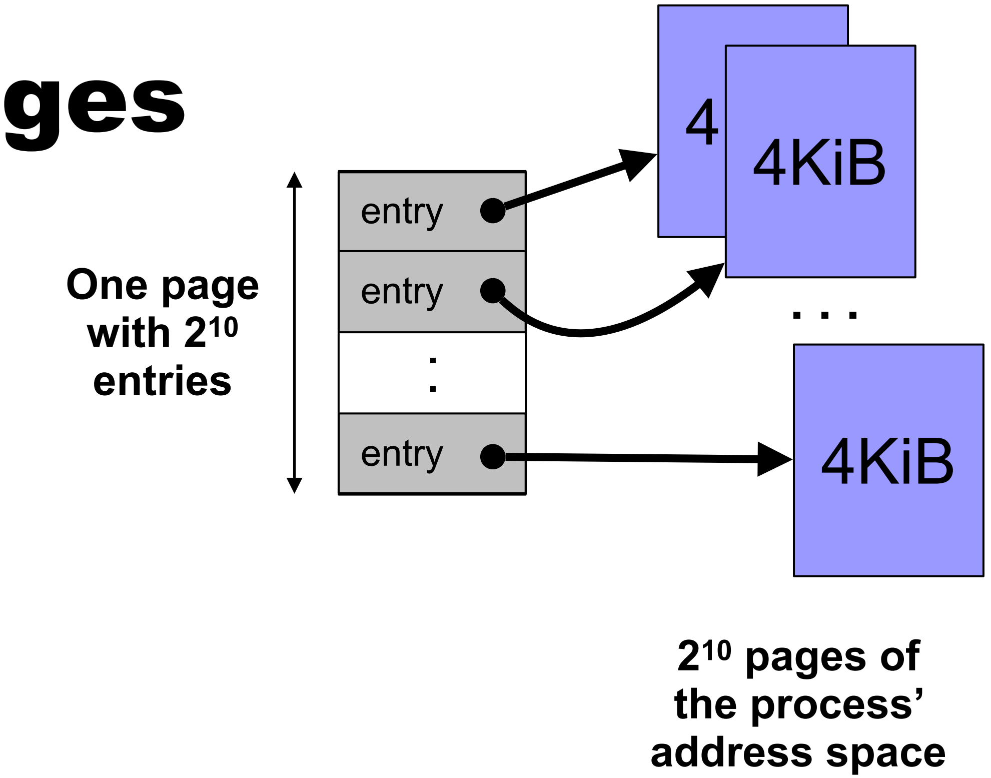
- We use paging to avoid large contiguous slabs of RAM
- To implement paging we use page tables
- But page tables are large contiguous slabs of RAM
- Sooooooo… to avoid big slabs to RAM we need big slabs of RAM 😱

Splitting the Page into Pages!

- What do we do when we have big slabs or RAM?
- We split them into pages!
- So the (large) page table is stored in multiple, possible non-contiguous pages
- The main question is now: how many page table entries can fit in a page?
- In our example, a page is 4KiB and an entry is 4 bytes
- So a page can contain 2^{10} (1,024) entries
- In the previous slide we said that our page table needs to have 2^{20} entries
- Therefore, we need $2^{20}/2^{10} = 2^{10}$ pages of page table entries
  - That's right: "page table pages"

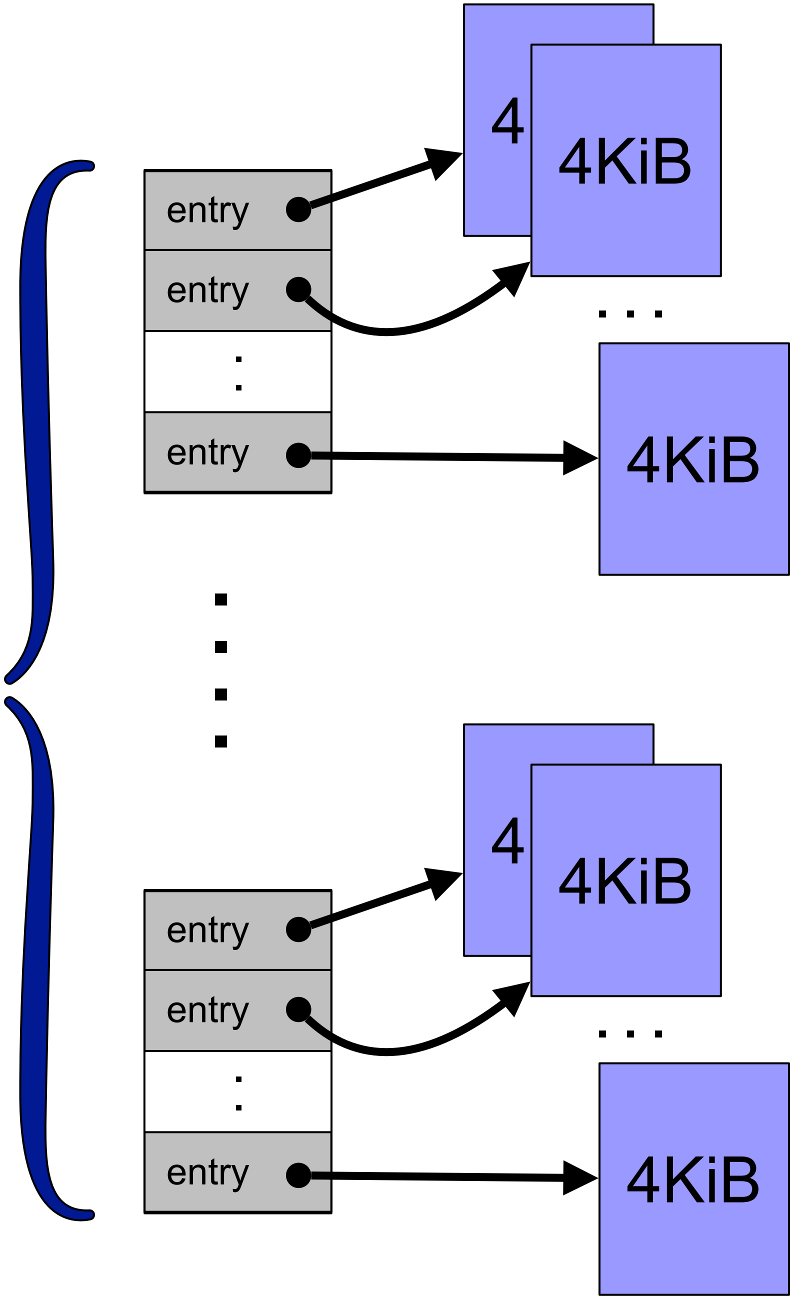
- Let's see this on a picture...

Page Table Pages


One page with 2^{10} entries
entry
entry
:
entry
4
4KiB
. . .
4KiB
2^{10} pages of the process' address space

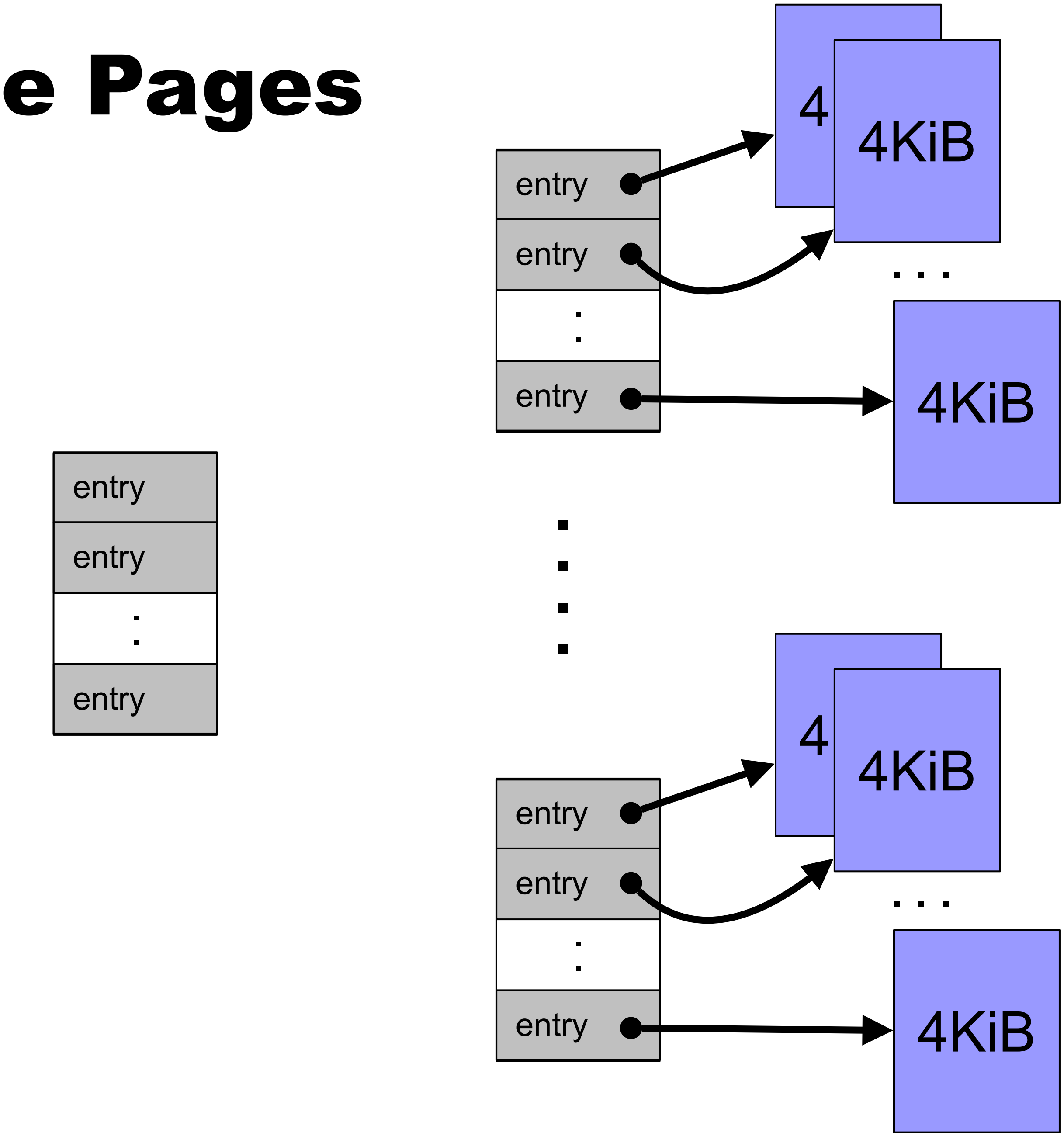
Page Table Pages

2^{10} pages that each contain 2^{10} entries for a total of 2^{10} x 2^{10} = 2^{20} pages in the process' address space



Page Table Pages

One page
with 2^{10}
entries

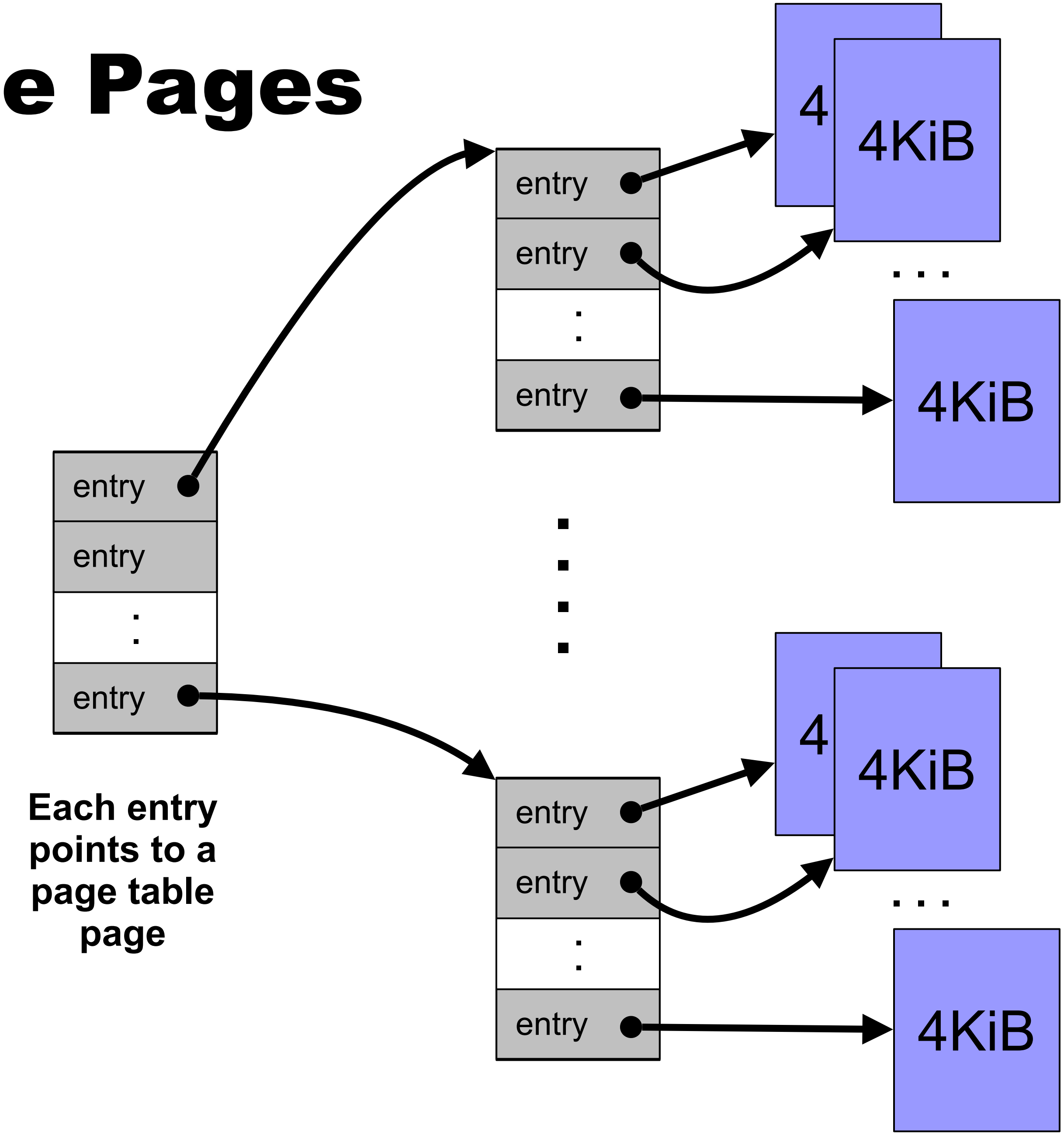


entry
entry
:
entry

entry
entry
:
entry
4
4KiB
. . .
4KiB

entry
entry
:
entry
4
4KiB
. . .
4KiB

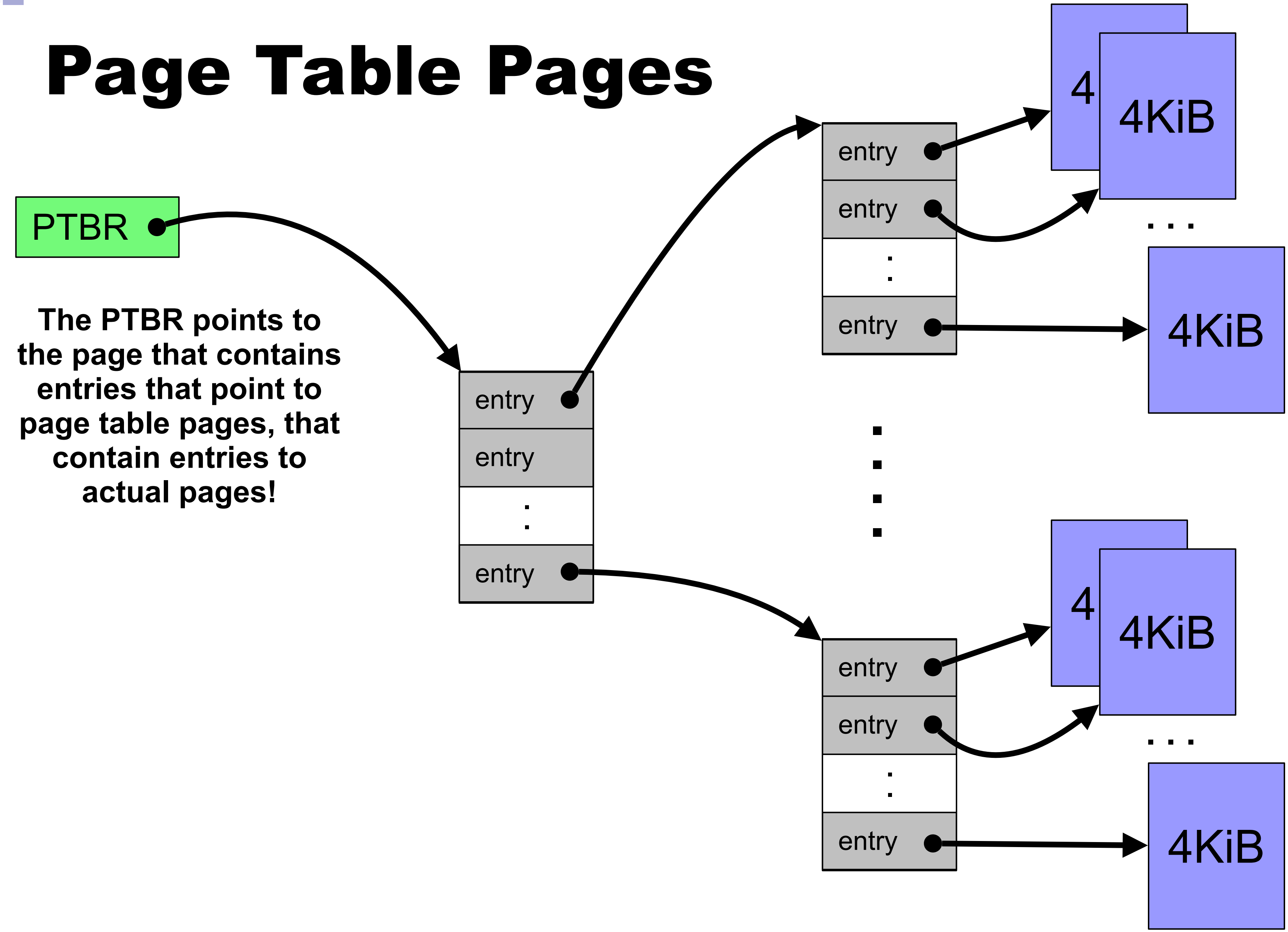
Page Table Pages



One page with 2^{10} entries

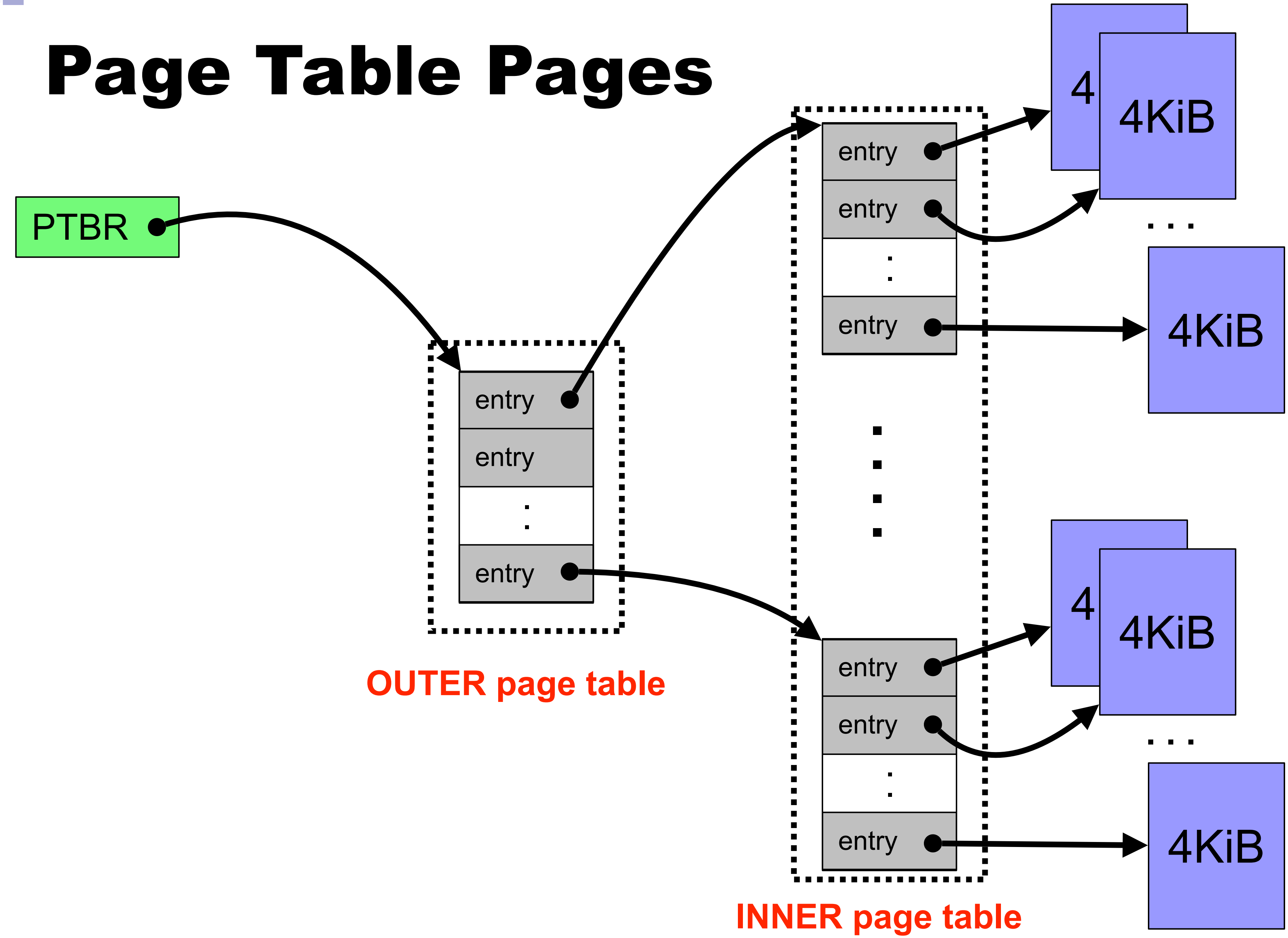
Each entry points to a page table page

Page Table Pages



The PTBR points to the page that contains entries that point to page table pages, that contain entries to actual pages!

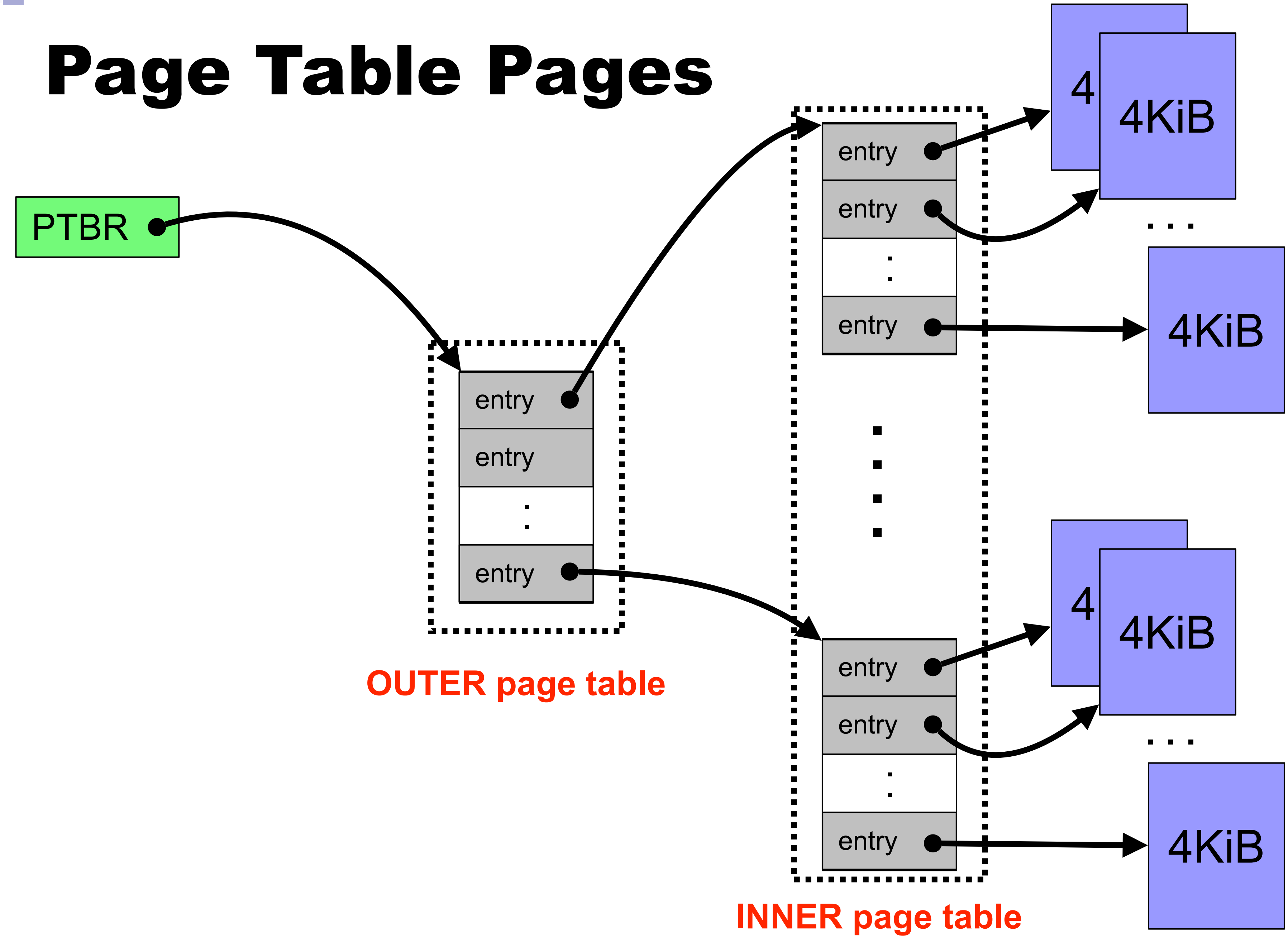
Page Table Pages



OUTER page table

INNER page table

Page Table Pages

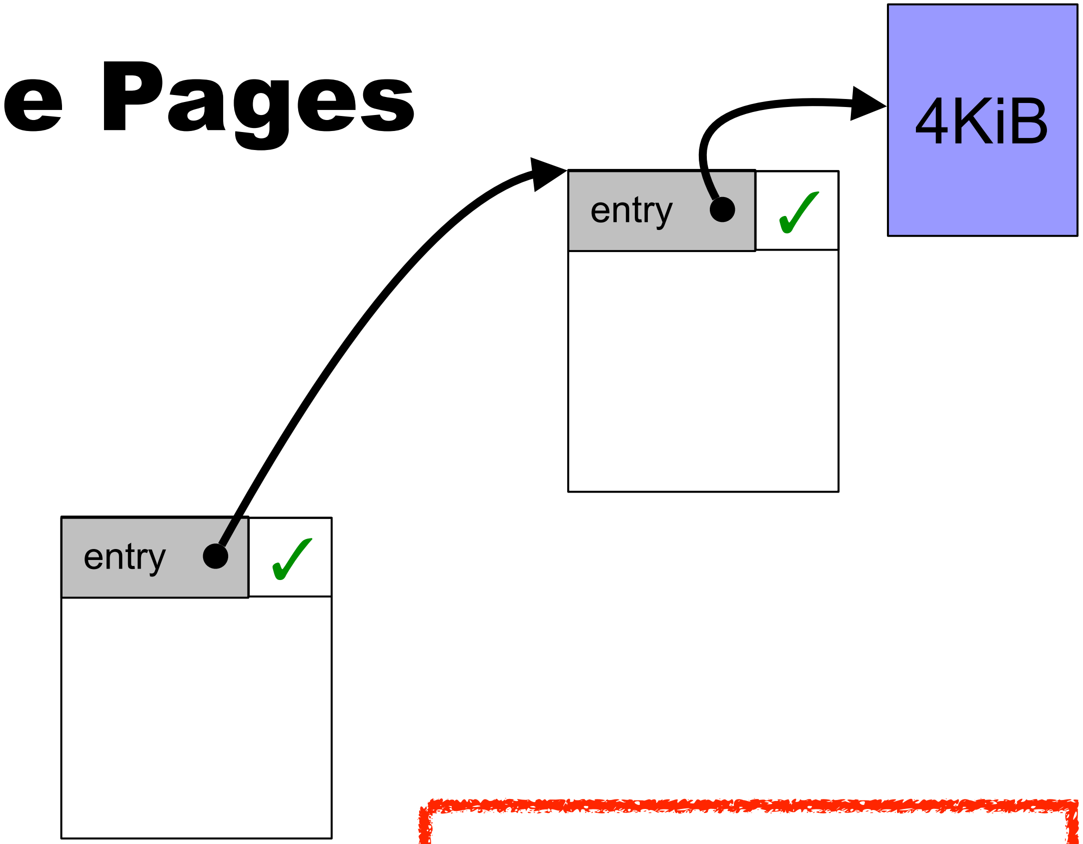


Page Table Pages

- In practice, an inner page table page is not allocated until its needed

Page Table Pages

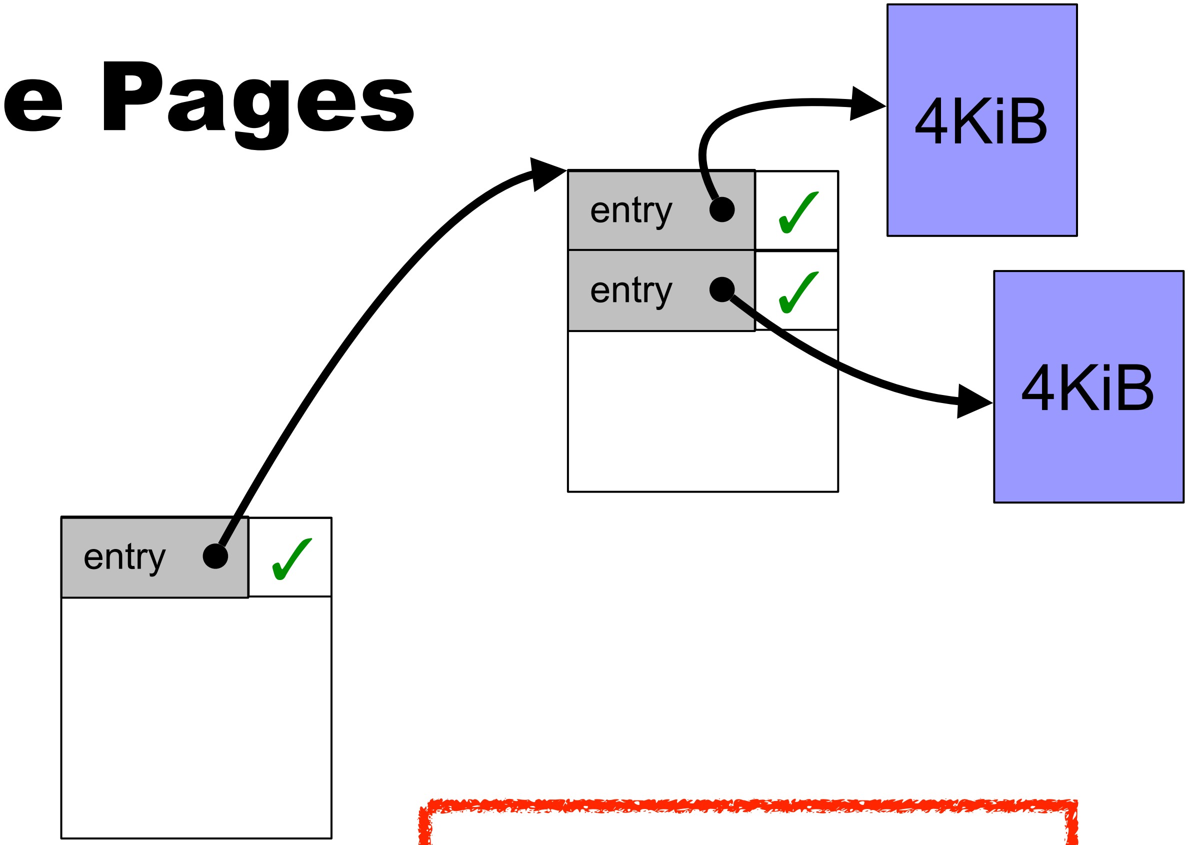
- In practice, an inner page table page is not allocated until its needed



First page of the process’ address space is allocated

Page Table Pages

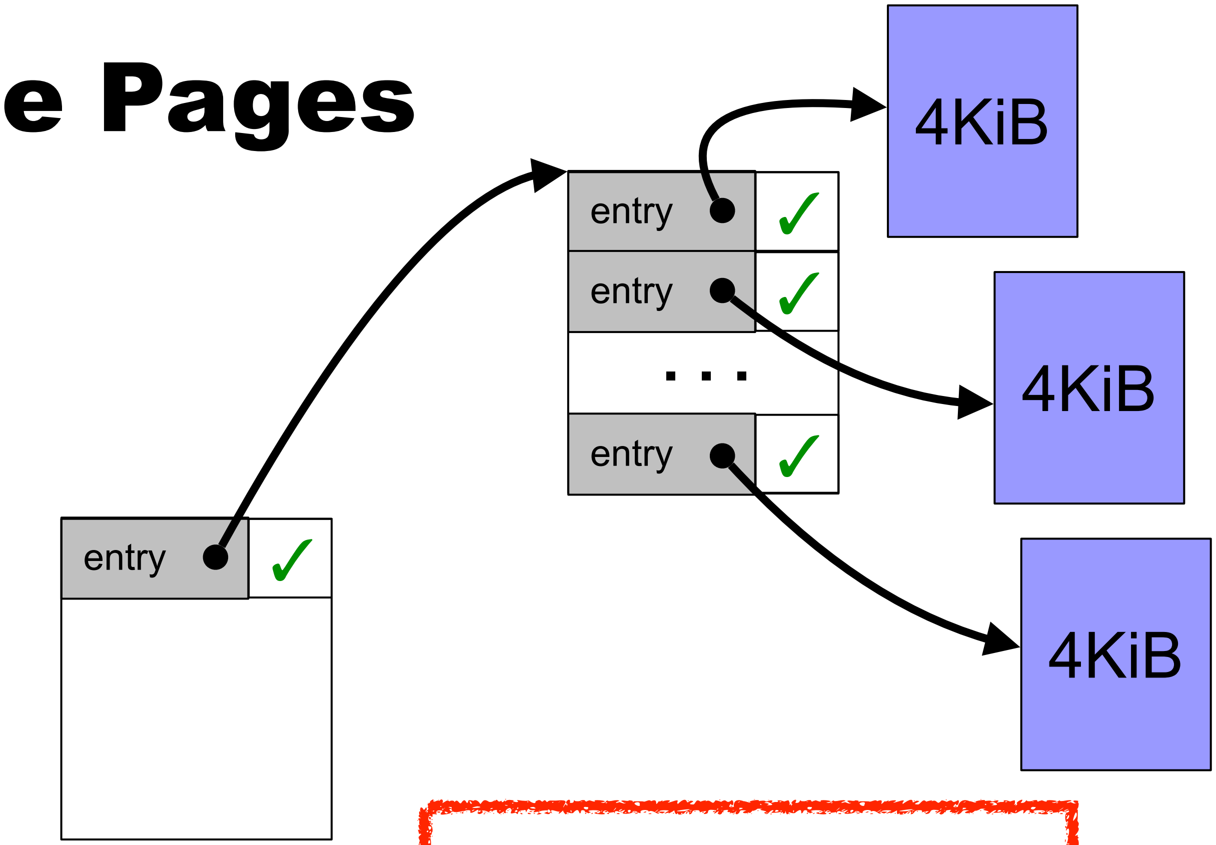
- In practice, an inner page table page is not allocated until its needed



Second page of the process' address space is allocated

Page Table Pages

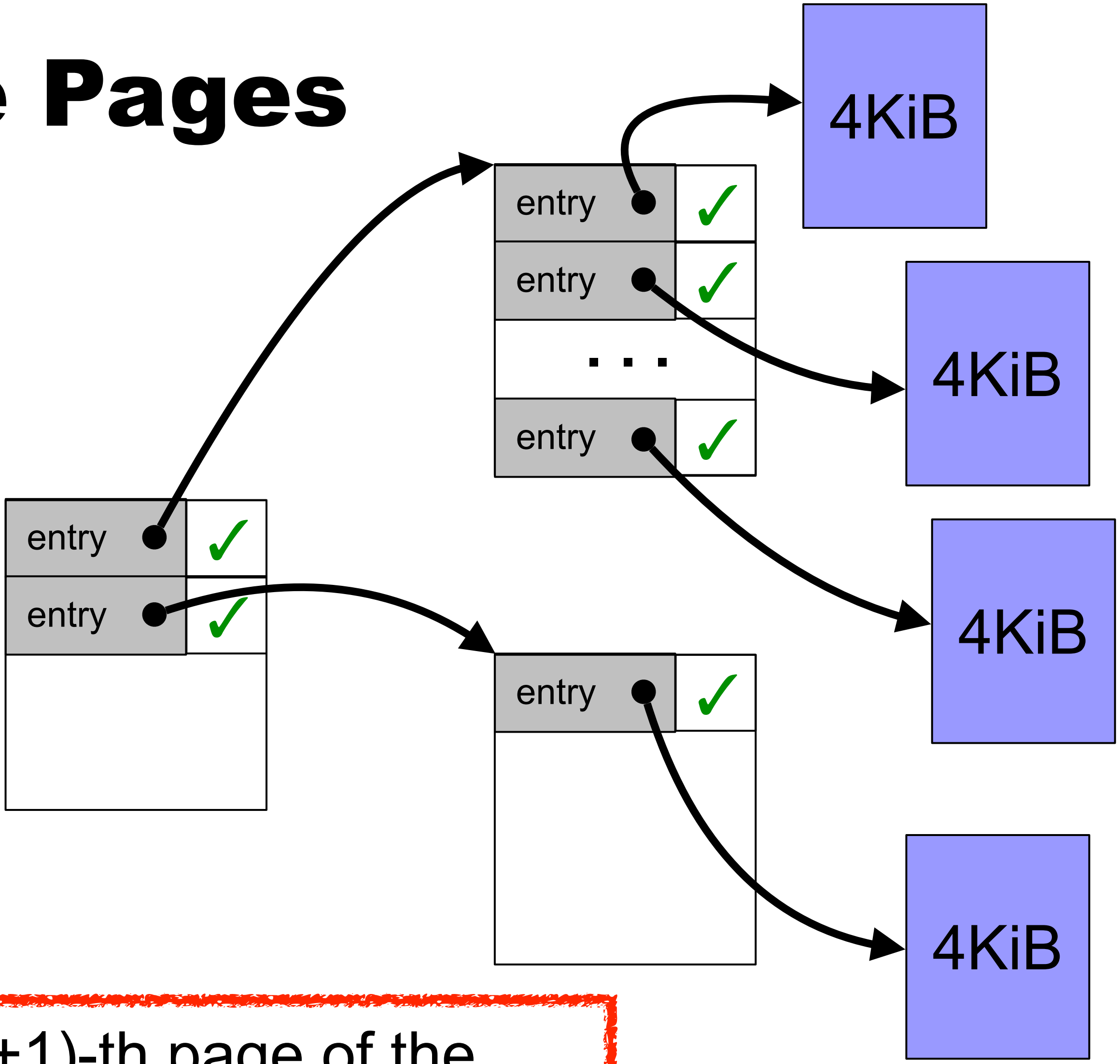
- In practice, an inner page table page is not allocated until its needed



2^{10}-th page of the process' address space is allocated

Page Table Pages

- In practice, an inner page table page is not allocated until its needed



(2^{10} +1)-th page of the process' address space is allocated, and so-on…

Hierarchical Page Tables

- The scheme on the previous slide is called hierarchical page tables
- The question now is: how to we perform address translation?
- It’s more complicated because we have one more level of indirection
  - But indirection is what we do as Computer Scientists all the time, so it’s good news :)

Hierarchical Page Tables

- For the previous example, given a 32-bit virtual address, we split it as follows:

10-bit index into **outer** page table	10-bit index into **inner** page table	12-bit offset in the page

- The first 10 address bits: to pick one of the 2^{10} entries in the outer page table should we use to find an inner page table page
- The next 10 address bits: to pick one the 2^{10} entries in the inner page table page should we use to find an address space page
- The next 12 address the offset in that page

- This works perfectly, in this example, because a page contains 2^{10} entries and 2^{12} bytes

Hierarchical Page Tables: Address Translation

p1	p2	Offset

- (Note: [@] means ”Contents at address @”)
- Assume that a page table entry is 4 bytes
- Address of the outer page table: PTBR
- Address of the relevant outer page table entry: $\mathrm{PTBR} + 4 \times p1$
- Address of the relevant page table page: $[\mathrm{PTBR} + 4 \times p1]$
- Address of the relevant entry therein: $[\mathrm{PTBR} + 4 \times p1] + 4 \times p2$
- Address of the page: $[[\mathrm{PTBR}+4\times p1]+4\times p2]$
- Physical address: $[[\mathrm{PTBR} + 4 \times p1] + 4 \times p2] + \mathrm{offset}$

(See OSTEP Section 20.3)

In-class Exercise

- Page size: 32 KiB
- Logical addresses: 39 bits
- Page table entry size: 8 bytes

- Using 2-level (aka hierarchical) paging, how is a logical address split into 3 outer page, inner page, and offset (denoted p1, p2, offset)?

- Typical approach:
  - How many bits for the offset?
  - How many page table entries can fit in a page? (gives us p2)
  - Then compute p1 as 39 - p2 - offset

In-class Exercise (Solution)

- Page size: 32 KiB
- Logical addresses: 39 bits
- Page table entry size: 8 bytes
- Using 2-level paging, how is a logical address split into 3 outer page, inner page, and offset (denoted p1, p2, offset)?

- There are $2^5 \times 2^{10} = 2^{15}$ bytes in a page, offset = 15
- We can have up to $2^{39-15} = 2^{24}$ pages in the address space
- We have $2^{15}/2^3 = 2^{12}$ page table entries in a page
- Therefore an inner page table page points to 2^{12} pages: p2 = 12
- Therefore, p1 = 39 - p2 - offset = 39 - 12 - 15 = 12
- This is yet another “lucky” case in which everything fits perfectly (because the inner page table has exactly 2^{12} entries)

When Things don’t Fit Perfectly

- Page size: 64 KiB
- Logical addresses: 41 bits
- Page table entry size: 4 bytes

- Using 2-level paging, how is a logical address split into 3 outer page, inner page, and offset (denoted p1, p2, offset)?
- What fraction of the outer page table is utilized?
- Let’s do the same reasoning as in the previous exercise…

When Things don’t Fit Perfectly

- Page size: 64 KiB
- Logical addresses: 41 bits
- Page table entry size: 4 bytes
- Using 2-level paging, how is a logical address split into 3 outer page, inner page, and offset (denoted p1, p2, offset)?
- What fraction of the outer page table is utilized?

- offset = 16 bits (because 2^{16} bytes in a page)
- An inner page table page points to $2^{16}/2^{2} = 2^{14}$ pages
- Therefore, p2 = 14
- And p1 = 41 - 14 - 16 = 11
- The outer page table page thus needs to hold 2^{11} entries
- **But it could hold up to 2^{14} entries**
- Therefore, only $2^{11}/2^{14} = 1/8 = 12.5\%$ of it are used!

Hierarchical Page Tables are it then?

- For 64-bit addresses, with 2-level paging, we are still in trouble though...
  - 4 KiB page size, 4-byte page table entry
  - Assume 64-bit virtual addresses
  - One outer page can address $2^{12}/4 = 2^{12}/2^2 = 2^{10}$ inner pages
  - Therefore: 64 - 10 - 12 = 42 bits to address all outer pages
  - The outer page size must be: $2^{42}×4=16×2^{40}$ =16 TiB!
  - So we need an extra level: 32 (second outer page) + 10 + 10 + 12
  - But the second outer page is still 2^{32} × 8 = 32 GiB and we now have three indirections
- Conclusion: Hierarchical page tables become memory hogs for large address spaces with small pages
- In practice: Virtual addresses are not 64-bit (`cat /proc/cpuinfo`) but more like 48-bit
- In practice: 4 levels are used

Isn’t this Slooooooow?

- With 4 levels, the page-to-frame translation is pretty slow
- It requires 4 memory accesses (outer, inner #1, inner #2, inner #3) to follow the chain of indirections
  - Plus some multiplications and additions
- So yes, it’s more expensive, but we don’t really care… anybody sees why?

Isn’t this Slooooow?

- With 4 levels, the page-to-frame translation is pretty slow
- It requires 4 memory accesses (outer, inner #1, inner #2, inner #3) to follow the chain of indirections
  - Plus some multiplications and additions
- So yes, it’s more expensive, but we don’t really care… anybody sees why?

- Because we have a TLB! So we don’t perform the translation very often because all our programs have locality!

Hashed Page Tables

- A completely different idea:
  - Pick a maximum (desirable) size for the page table (say N)
  - Create a hash function that associates any VPN to an integer of 0..N-1
  - Structure the page table as a hash table using the hash function (each entry in 0..N-1 is a list of PFN)

- This is interesting but not really done in practice

Inverted Page Tables

- Yet another idea:
  - One table for all processes
  - One entry per physical memory frame
  - Each entry is: ASID + logical page number
  - CPU issues addresses like: PID + VPN + offset
  - And page table contains entries like (PID, p) to PFN
  - Searching for (PID, p) is expensive
    - You can’t have both great space- and time-complexity :)
  - And need for a mechanism to implement shared memory

- Was used in: PowerPC, UltraSPARC, IA-64 (Itanium) – Discontinued

Main Takeaways

- Paging is a good idea, but it has its problems
- Problem #1: Address translation is slow
  - **Solution: Use a TLB**
- Problem #2: The Page Table can’t be contiguous memory that’s larger than a page
  - **Solution: Use a hierarchical structure**
  - The hierarchical structure makes translation slower, but we don’t care because we have a TLB anyway!
  - It requires that logical address bits be split into different parts
  - For instance, for a 2-level we would have a 3-way split: outer, inner, offset

Conclusion

- We still have **one big question:** What happens when a process needs a new page, and there is no free frame???
- This is the topic of the next set of lecture notes
- But first…
- Let’s look at Sample Problems…
- And at Sample Homework #7...