Paging 2

Lesson Summary

- Paging is a good idea, but it has its problems
- Problems
  1. Address translation is slow
     Sol'n: Use a TLB
  2. : The Page Table shouldn't be contiguous
     Sol'n: Use a hierarchical structure
     hierarchical structure makes translation slower, but we don't care because we have a TLB anyway!

Paging is great, but it's expensive :(

- each address coming out of. the CPU is virtual
- **Address Translation** (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)
  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

  made our RAM twice as slow

Locality and Caching

Temporal Locality repeated access to the same memory location	Spatial Locality repeated access to nearby memory locations

`counter++`
`-> counter is accessed over and over`
`->the same frame is accessed over and over`
	`a[i] = a[i-1] + a[i-2]`
	`-> all three array elements are very likely in the same frame`
	`-> the same frame is accessed over and over`

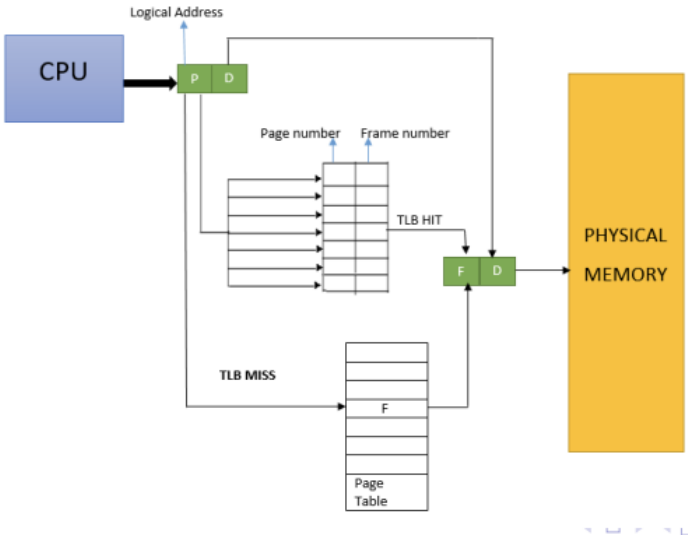
- Therefore, as a process executes, the address translation requests often look like:

```
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...
```

- should remember (i.e. cache) previous translation results

Translation Lookaside Buffer (TLB)

- hardware component that does the caching of previous translations
- 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 Performance

- Typical Characteristics
  - Contains 12 to 4,096 entries
  - Performance:
  On hit: less than 1 clock cycle
  On miss: 10-100 clock cycles
  - Miss rate: 0.01 - 1%
- **Replacement Policy**
  must be defined when the TLB is full
  - Least Recently Used (LRU)? Random?
- Some TLBs allow for some entries to be un-evictable
  - ex) kernel pages

The TLB and Context-Switches

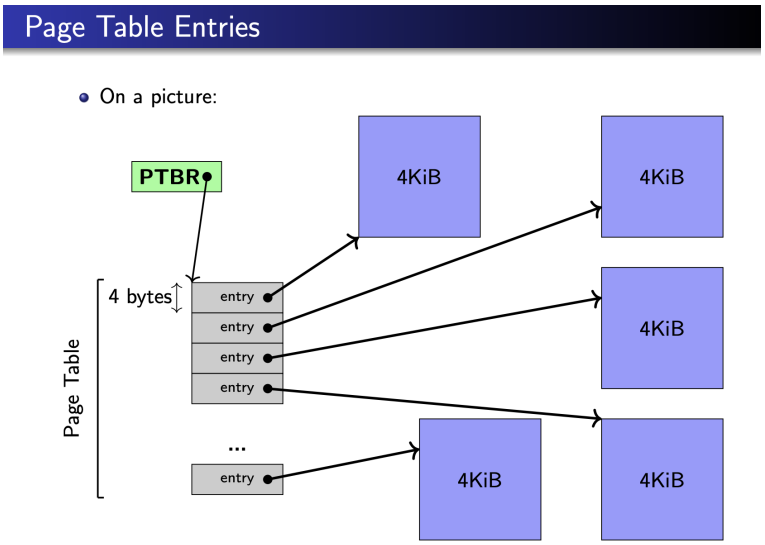
What happens with the TLB on a context-switch?
ASIDs: Address-Space IDentifiers

- Each TLB entry is annotated with a process identifier
- TLB can contain entries associated to multiple processes (kernel code, shared libraries, multi-threaded program, ...)
- Each lookup attempts to match entry ASIDs with the ASID of the current process (and if mismatch then it's a TLB miss)

Page Table Entries

How many bits are needed for a page table entry?

- the page table consumes space in 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, ASID bits, but there are other things)



A Note on Page Table Structure

- page table is just an array of entries
  - entry for page 0 is the first element
  - entry for page 1 is the second element
- looking up the entry for page i means

$$\text{PTBR} + i \times \text{entry size}$$

Hierarchical Page Tables

-

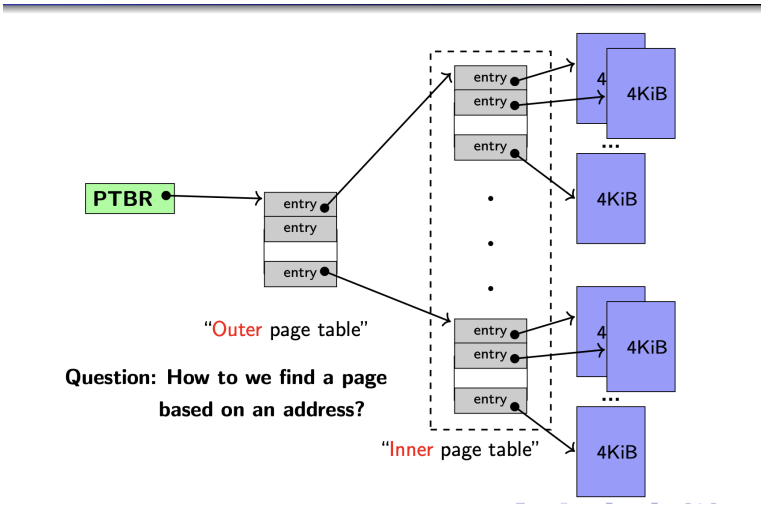


Image of a hierarchical page table

Hierarchical Page Tables: Address Translation

- Address of the outer page table: PTBR
- Address of the relevant outer page table entry

$$\text{PTBR} + 4 \times p1$$

- Address of the relevant page table page

$$[\ \text{PTBR} + 4 \times p1]$$

- Address of the relevant entry therein

$$[\ \text{PTBR} + 4 \times p1] + 4 \times \text{p2}$$

- Address of the page

$$[[\ \text{PTBR} + 4 \times p1] + 4 \times \text{p2}]$$

- Physical Address

$$[[\ \text{PTBR} + 4 \times p1] + 4 \times \text{p2}] + \text{ offset }]$$

In Class Exercises

Exercise 1

- Page size: 32 KiB
- Logical addresses: 39 bits
- Page table entry size: 8 bytes (= 64 bits)
- 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)

Exercise 2

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

- 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?

→ Hierarchical page tables become memory hogs for large address spaces with small pages

Hashed Page Tables

- 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)
- interesting but not done in practice

Inverted Page Tables

- 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
- And need for a mechanism to implement shared memory