contents of TLB:

A typical TLB might have 32, 64, or 128 entries and is **fully associative**

any given instruction can be anywhere in TLB, and the hardware will search the entire TLB in parallel to find the desired translation.

how? hardware compares the VPN against all entries at the same time

This is done with comparators → each TLB entry has a small circuit that checks "Does my VPN == the requested VPN?"

- parallel search → all comparators check simultaneously
- if one comparator matches → it signals hit, and its PFN is selected.
- if none → it's a miss.

A TLB entry might look like VPN | PFN | other bits.

What are these other bits?

valid, protection, dirty, etc...

Note that TLB valid Bit ≠ Page table valid bit

indicates whether the TLB translation is valid!

indicate whether page has been allocated or should be accessed by the current running program, etc...

How to manage TLBs on a context switch?

During context switch, hardware or OS must be careful to ensure that the about-to-be-run process does not accidentally use translations from some previously run process.

For eg:- consider the following situation.

problem: VPN 10 translates to either PFN 100 (P1) or PFN 170 (P2), but the hardware can't distinguish which entry is meant for which process!

What should the hardware or OS do inorder to solve this problem?

solution 1: flush the TLB on context switches.

↳ sets all valid bits to 0, when the page table base register is changed during context switch.

A problem with the above solution: costly if OS switches b/w processes frequently, as we lose all recently cached translations on flush.

solution 2: Track which entries are for which process.
- using Address space Identifier (ASID)
  ↳ an additional field in TLB. (typically 8 bits)

TLB Table

VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
-	-	-	-	-
10	170	1	rwx	2
-	-	-	-	-

With the help of ASIDs the TLB can hold translations from different processes at the same time without any confusion.

Question: Why not use PIDs instead of ASID?

* PIDs are large (16 bits, 32 bits or even larger) - so if the TLB used PID directly, it would waste space and make hardware more complex.

Eg:- MIPS has 8-bit ASIDs

Note that OS is responsible for assigning ASID values to processes.
- OS picks an ASID when it creates a process

- on context switch, OS "writes" the process's ASID into a privileged register

- TLB look up checks (VPN, ASID) instead of just VPN.

what happens when ASID space runs out?
Suppose all ASIDs are in use and new process needs one:
- the OS chooses an ASID to recycle (usually an old or inactive one)
- to avoid conflicts, the OS must flush all TLB entries with that ASID before re-use.

There could also be a situation in TLB where two entries of two different processes has two different VPNs that point to the same physical page.

↳ during page sharing.

VPN	PFN	valid	prot	ASID
10	101	1	r-x	1
—	—	0	—	—
50	101	1	r-x	2
—	—	0	—	—

How to design TLB replacement policy?

↳ i.e., which TLB entry should we replace, when a new entry needs to be inserted, and the TLB has no free slots left.

Goal: Design a replacement policy that minimize TLB misses!

one common approach: evict least-recently-used or LRU entry.

another approach: random policy → evicts a TLB mapping at random.

↳ useful to avoid corner-case behaviors.

Paging: smaller tables

Recall,

Problem 2: Page tables can be huge in size, as a result memory might ends up filled with page table instead of useful application data

32-bit address space (2^{32} bytes), 4KB (2^{12} bytes) pages, 4-byte PTE

$\#pages = 2^{20} \Rightarrow$ page table size = 4MB!

100 active process $\Rightarrow$ 400 MB for translations, HUGE!

Problem: How to make page tables smaller?

An easy solution → use bigger pages

($\Rightarrow$ #pages is less $\Rightarrow$ #PTE are less $\Rightarrow$ page table is small)

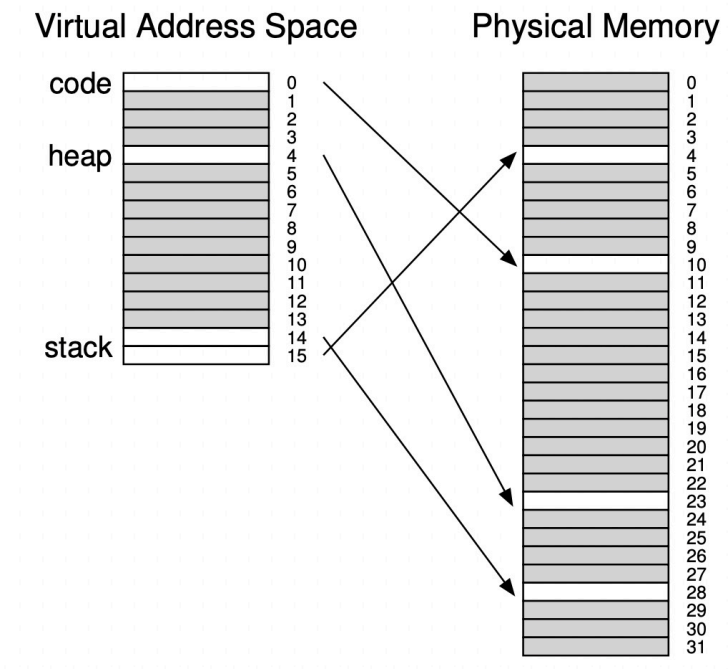
Major problem with this approach → Internal fragmentation

Another solution: Hybrid approach: Paging and segments

"can we get best of both worlds?"

The creators of Multics chanced upon such an idea to reduce the memory overhead of page tables.

Eg: consider a 16 KB address space with 1 KB pages (on the left) and its corresponding page table (on the right)



PFN	valid	prot	present	dirty
10	1	r-x	1	0
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
23	1	rw-	1	1
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
28	1	rw-	1	1
4	1	rw-	1	1

Used portions of the heap and stack are small!

most of the page table are unused, full of invalid entries!

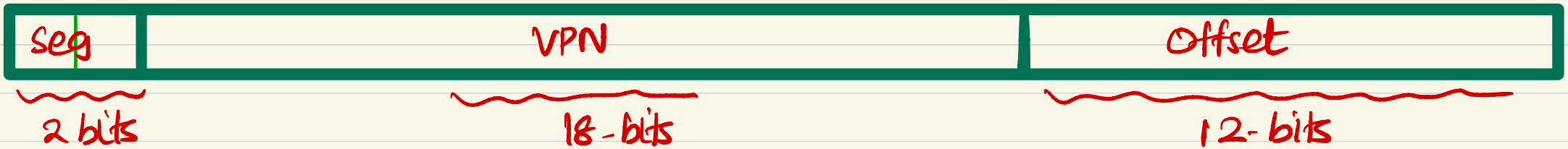
Too much wastage of space in page table!

Idea of segmentation + paging → Instead of having a single page table for the entire address space of the process, have one page table per logical segment.

i.e., Divide address space into segments → code, heap, stack

Divide each segment into fixed-sized pages.

A virtual address looks like: eg:- 32-bit address space, 4KB pages.



01- code, 10- heap, 11- stack.

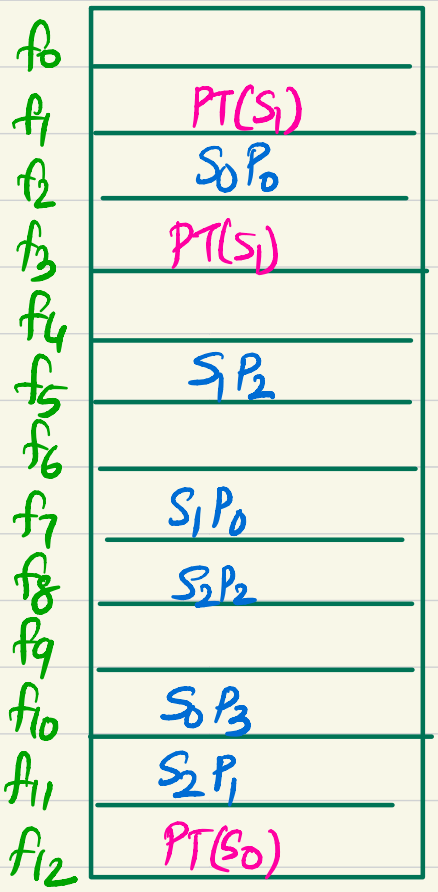
Now, for each segment, in the hardware, we need three base/bounds pairs of registers.

base register → contains the physical address of page table of that segment

bounds → value of the maximum valid page in that segment.

on a context switch, these registers must be changed to reflect the location of the page tables of newly running process.

Virtual memory



Physical memory

$PT(S_0)$

PFN	other bits
f_2	.....
—	
—	
f_{10}	.....

$PT(S_1)$

PFN	other bits
f_7	....
—	
f_5	.....

$PT(S_2)$

PFN	other bits
—	
f_{11}	
f_8	

on a TLB miss, what does the hardware do?

- extract the segment bits (SN) as follows:

  SN = (virtual address & seg_mask) >> SN_shift

- extract the VPN

  VPN = (virtual address & VPN_mask) >> VPN_shift

- bound check

  if VPN > bound → segmentation fault.

- find the address of PTE

  Address of PTE = Base[SN] + (VPN * size of(PTE))

Advantages of segmentation + paging

segmentation → supports logical, variable-sized divisions of the program

paging → eliminates external fragmentation in physical memory

Hybrid → local organization + efficient physical memory allocation.

Efficient page table management

- each segment has its own page table and therefore, no wastage for invalid pages lying b/w the segments.

Disadvantages

- page table overhead within segments

  If the segment is sparse within (many invalid pages) then there is wastage of memory in the corresponding page table.

- additional work for hardware

Although there is no external fragmentation in user space of the memory, with this approach, page tables are of different sizes.

↳ may cause external fragmentation.

Different approach: Multi-level page tables

#entries in the page table = total # pages.

Why keeping entries for so many invalid pages in the memory?

Question: How to get rid of all those invalid regions in the page table?

↳ multi-level page table.

(used by many modern systems)

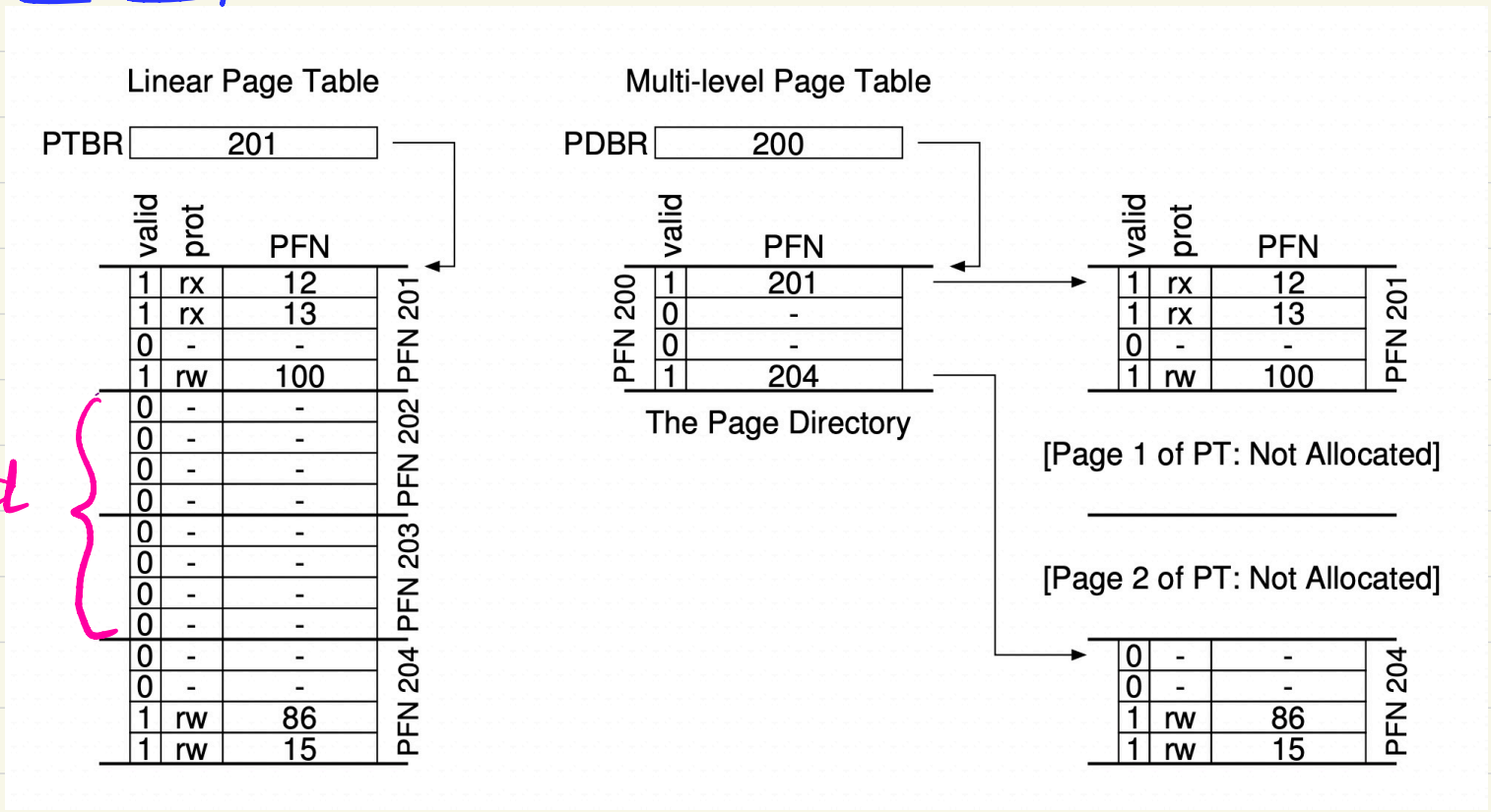
Idea: chop up the page table into page sized units;

Advantage: if all PTE in an entire page of a page table is invalid, no need to allocate that page of the page table.

How to track the valid pages of the page table? → use a page table directory.

For each page of a page table, the directory either gives the physical location of the page or marks it invalid.

An example



invalid pages

page directory entries = # pages of a page table.

A Page Directory Entry (PDE) contains a valid bit and page frame number (PFN).

PDE (X) = valid ⇒ atleast one of the PTE of the page table contained inside this page X is marked valid.

(X is a page of a page table ⇒ X is a page full of PTEs)