Memory Management

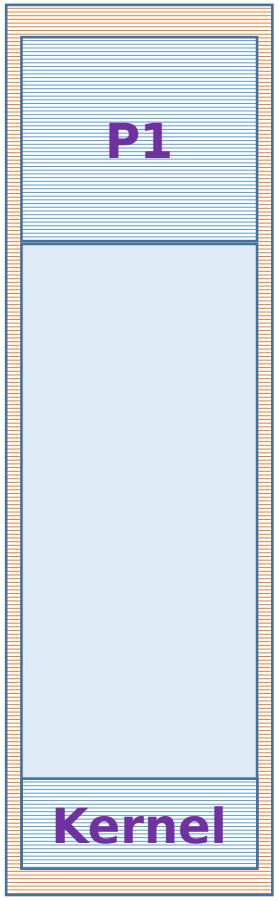
Everything needs memory-RAM

- Processes live in the memory when they are executed
- They access data in the memory
- Before processes read data from disk, they need it to be in memory
- The kernel is in the memory
- ...

- But memory is limited space, so memory management is important
- We start by asking how to allocate memory to a new process

Allocating memory to the process- Flat

- Why not allocate the whole RAM to a single process?
  - We could just keep the current active process's memory in the RAM, the rest on the disk
  - So, works with time slices too
- **Absolute address** code with **compile-time binding**
- **Single contiguous address space (Flat)** same for all processes



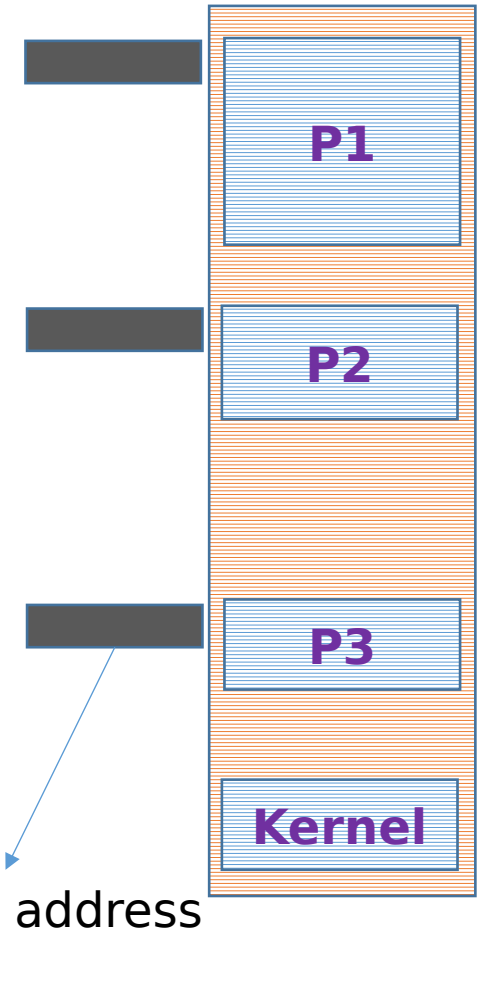
- We could compile programs to this address spaces
- BTW, the kernel too resides in the same physical memory!

Allocating memory to the process: Partitioning the physical memory available

- Different address spaces
  - Makes compiling difficult with absolute addresses
  - Allocate a set of **contiguous areas** to each process
  - Possibly allocate in blocks

ALTERNATELY –

- Provide **relocatable** code
  - Actual address determined by combination of compile time address and starting location, say.
  - Need address translation support from CPU



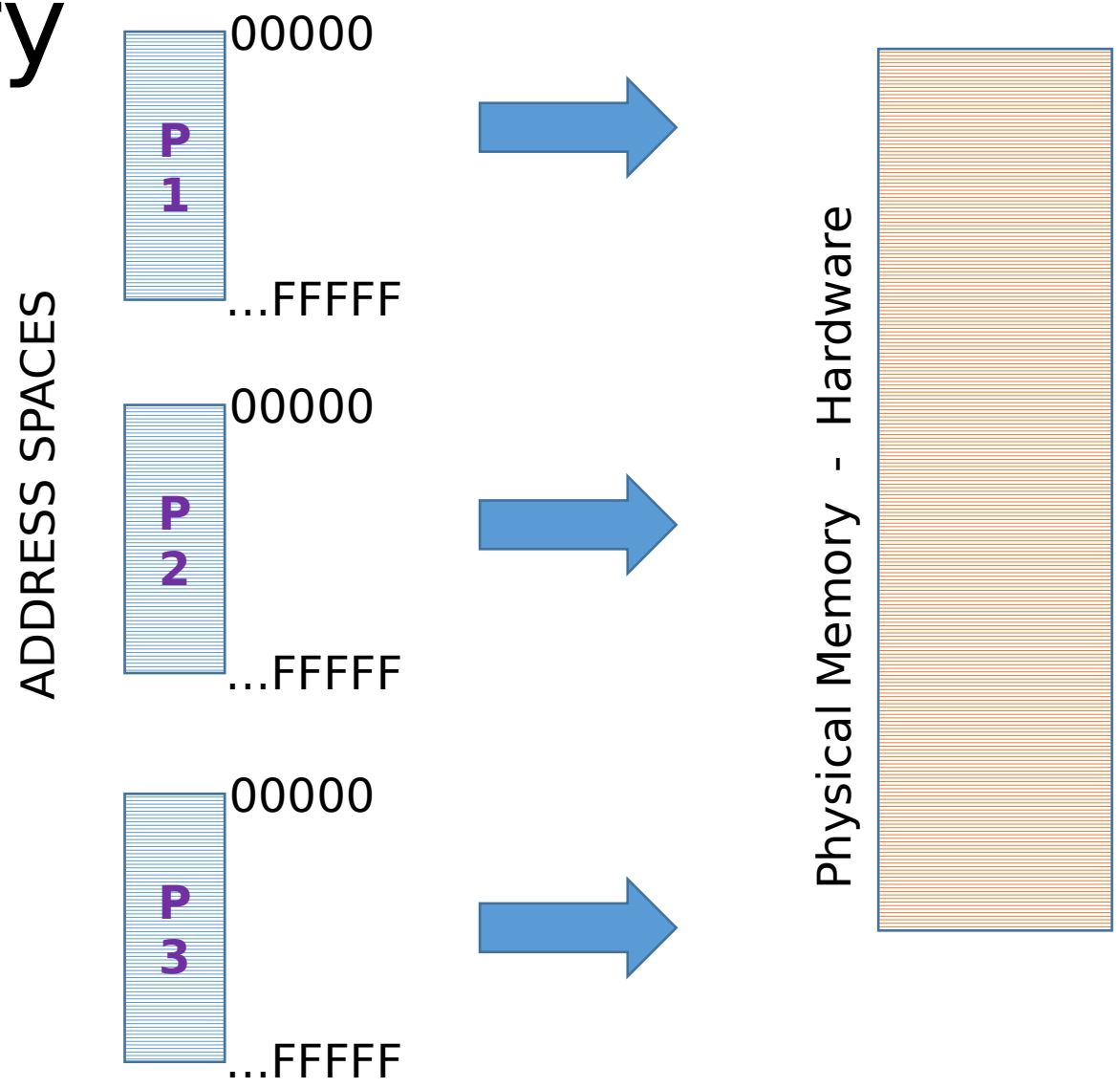
- We wish to support multiple process in memory at the same time.
- We need to provide protection.
- First step to separate physical address from compile time address

Core concept: Separation of Process address space and physical memory

- Each process runs in its own space of addresses which are purely virtual
  - **Process Address Space**
  - All addresses of variables and functions belong to this space only!
  - So ***by design*** there is no way for process P_1 to address a location in process P_2 (protection for free)
  - `printf("%p",&x);` prints such an address
- The CPU however needs to access the physical memory
  - Data and instruction reside in the hardware/RAM **physical memory space**.
  - The physical address is only determined when the process is loaded or when it is run.
- The above two address spaces are ***not*** the same
  - So be careful not to confuse the two.

Virtual (Process) Address Spaces and Physical Memory

- The physical RAM could be smaller than the virtual address space.
- The virtual address space of one process has nothing to do with the virtual address space of another process.
- We need to somehow do a mapping.
- We need to account for proc address spaces adding up to more than physical memory on the machine.

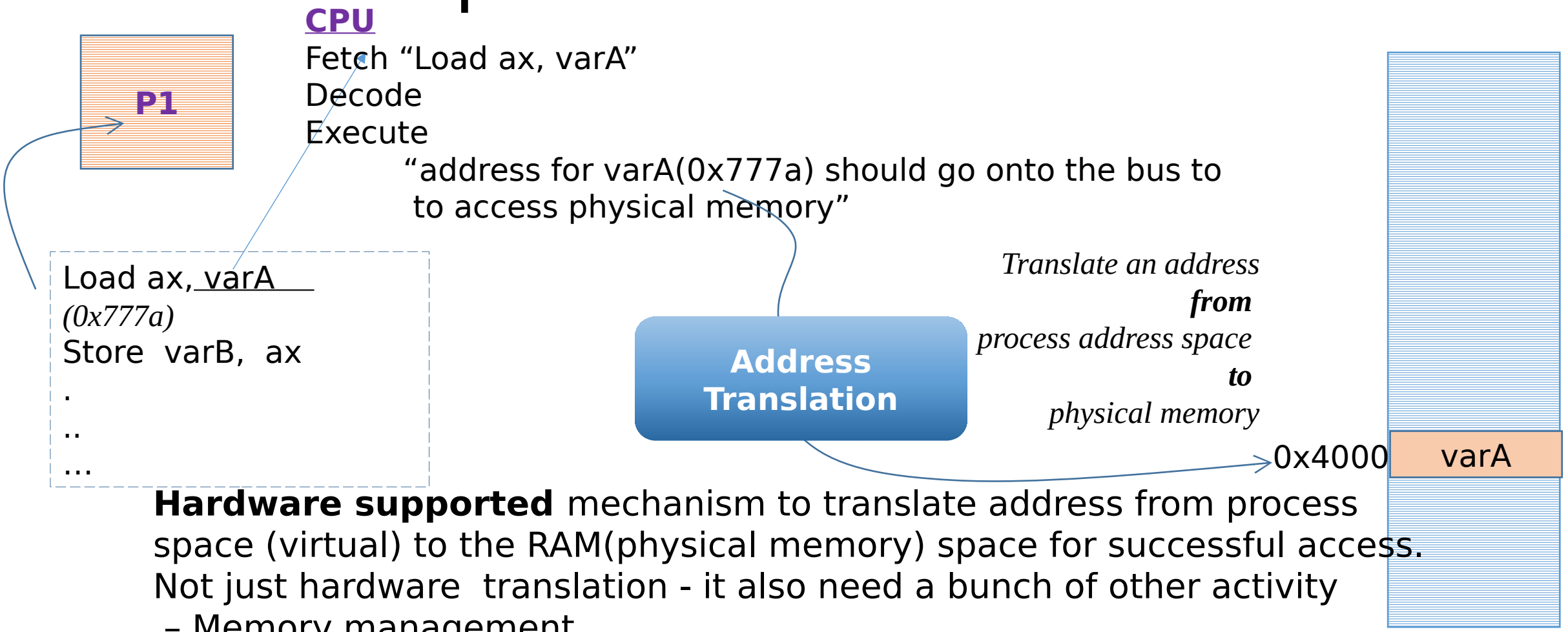


Goals of supporting virtual address spaces

- Transparent (needs no modification to the way the user uses it)
- Efficient (should not cause system to slow down significantly)
- Protection (should provide protection against wrong access – isolation)

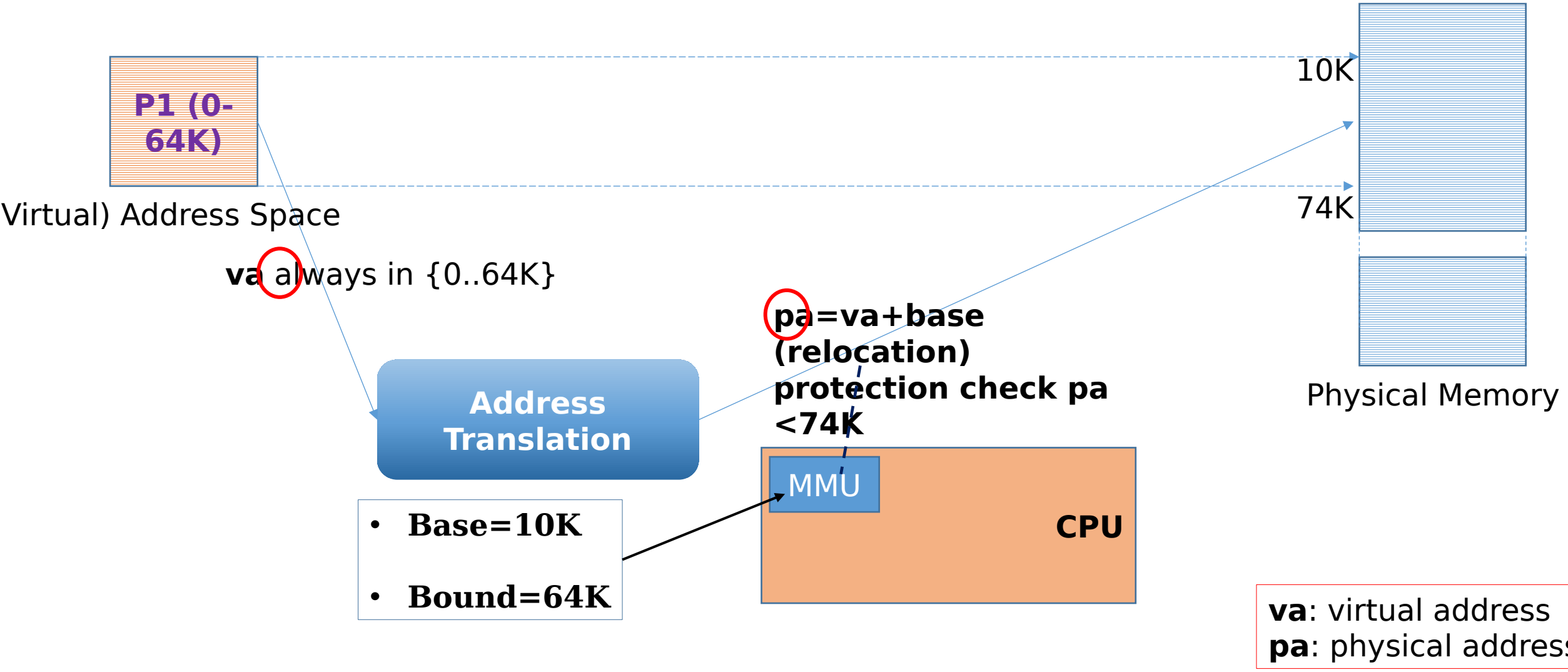
- All these are the goals of *Address Translation*…

Address translation: Core mechanism for to support virtual address spaces



Hardware supported mechanism to translate address from process space (virtual) to the RAM(physical memory) space for successful access.
Not just hardware translation - it also need a bunch of other activity
– Memory management

The (*per process*) Base and Bound method: Simple address translation hardware support

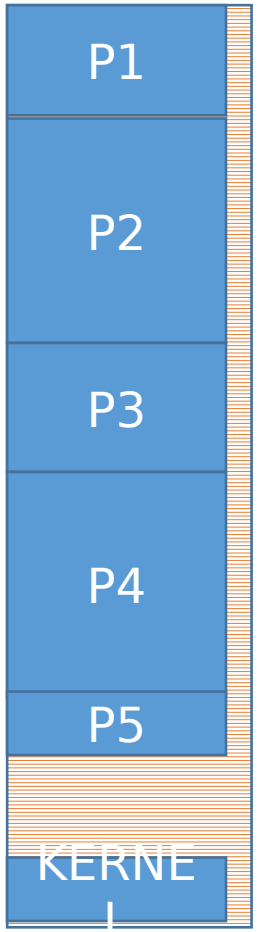


Support needed

- MMU with base and bound registers
- MMU need to do the translation each time an address is sent out on the wire
- Protected instructions
  to updated “base” & “bound”
- “OOB” Exception support/handler
- Also need to store “base” and “bound”
  in process context during context-switch
- Need management *code* to do allocation and track free/ allocated physical memory.

Contiguous Memory allocation strategies for base and bound translation scheme

- Models: Either contiguous or fixed size blocks
- Over times we have many free ‘holes’ (contiguous chunk of free space)
- When a new request comes we need to find a free ‘hole’ of the right size.
  - First fit
  - Best fit
  - Worst fit
  - …
- May become difficult –
  - We may not have a **single big enough** hole
  - The **fragmentation problem**!

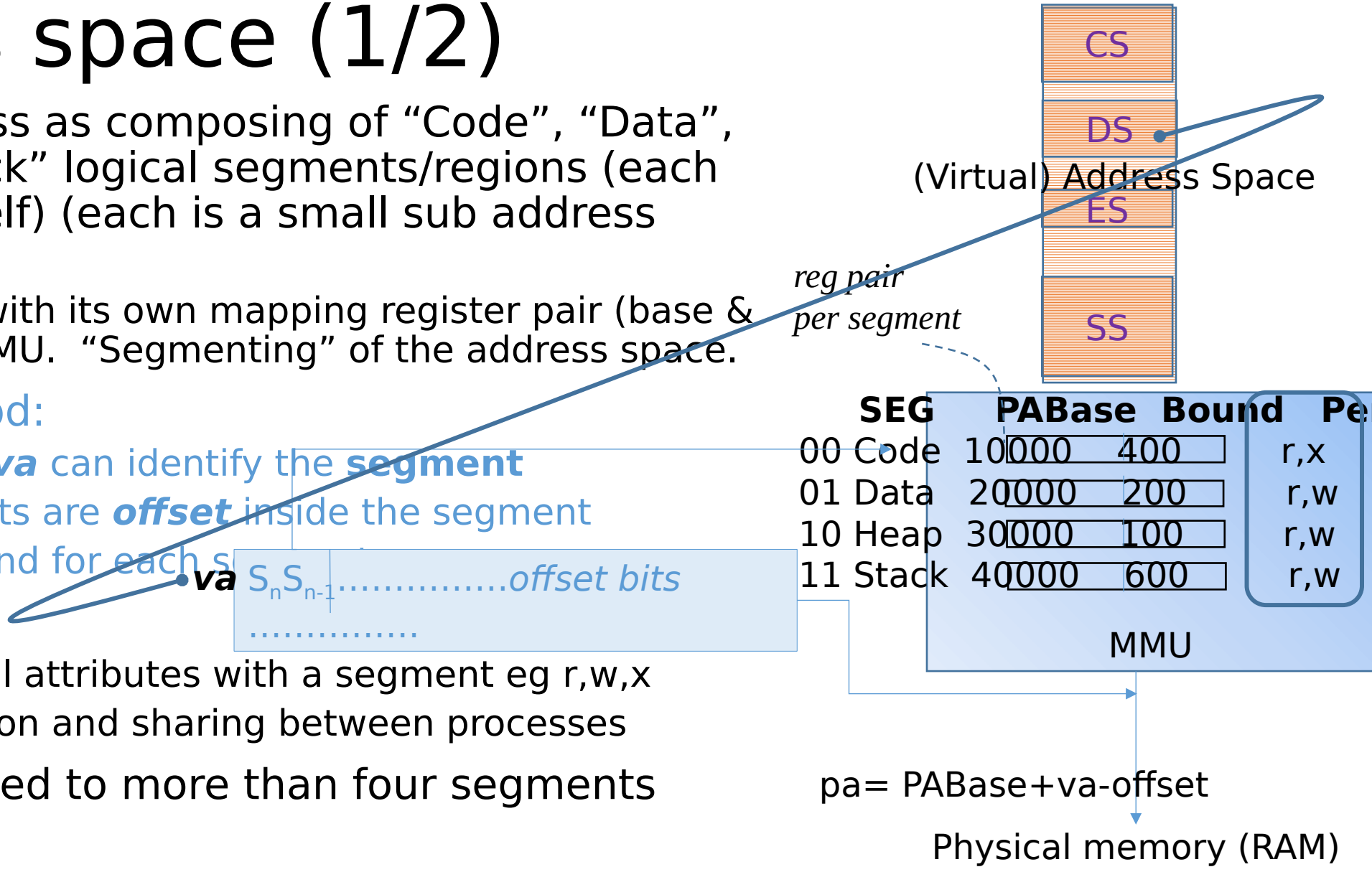


A Commentary on the simple solution

- + simple
- + provides Transparency, Efficiency and Isolation(protection)
- + Is actually dynamically relocatable (ie the process can be remapped as it is executing).
- + can support multiple processes (by loading new MMU regs when doing context switch.
- + Isolation among processes provided
- + Compiler can assume a simple address space starting at zero.
- - it assumes the address space is contiguously used
- - it causes fragmentation
- - it maps all of the address space to the physical memory
  - - can go ‘out of memory’ if we have lots of large process address spaces
- - it doesn’t allow protection within the processes (process can overwrite its own code, by mistake)
- HOWEVER, this is the core idea on which other methods are built

Concept: Segmenting the process address space (1/2)

- Consider a process as composing of “Code”, “Data”, “Heap” and “Stack” logical segments/regions (each contiguous in itself) (each is a small sub address space)
  - Providing each with its own mapping register pair (base & bound) in the MMU. “Segmenting” of the address space.
- Translation Method:
  - First two bits of ***va*** can identify the **segment**
  - Remaining ***va*** bits are ***offset*** inside the segment
  - Needs Base bound for each s
- Additionally
  - Associate special attributes with a segment eg r,w,x
  - Enables protection and sharing between processes
- Can be generalized to more than four segments



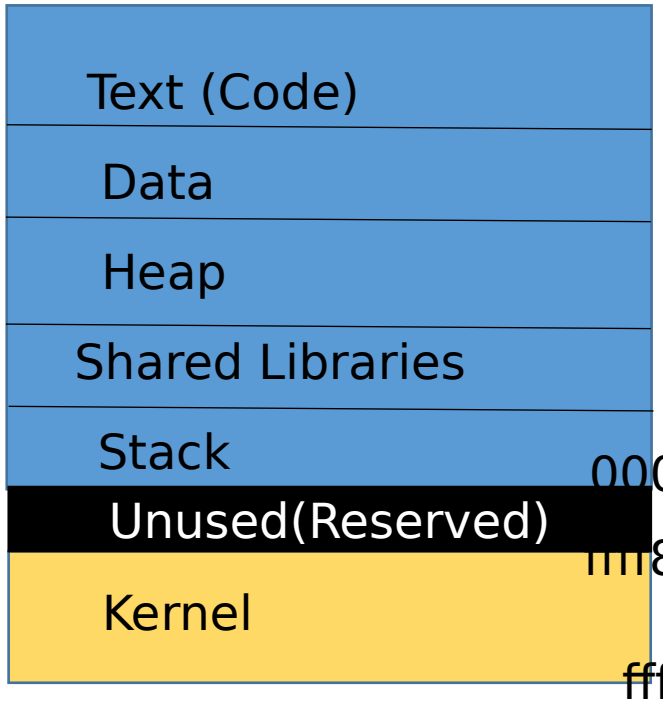
Concept: Segmenting the process address space (2/2)

- Each segment is independently relocatable, dynamically
- We could limit the size of each and thus not allocate physical space to all the address space in between
- Set of segment registers (we may call that segment table) now become part of the context.
- **Compiler** needs to flag each segment (in the executable file) so that the loader can the right thing at run time.
- Look at `pmap [-X] <PID>` ; `readelf –e a.out` or `objdump –hj a.out`
  - These are not really segments, but VMAs … see next slide

Virtual address space and ***pmap*** on linux x86-64

```
iiitb@iiitb-Vm:~/courses/os/misc$ ./address
writecode) is at 0x7fcbb3d3d090
main(code) is at 0x55df5f025229
g   (data) is at 0x55df5f028014
a   (heap) is    0x55df604db2a0
x   (stak) is at 0x7ffc58c8d0b4
a   (stak) is at 0x7ffc58c8d0b8
z   (stak) is at 0x7ffc58c8d0c0
```

```
iiitb@iiitb-Vm:~/courses/zzz$ pmap `pidof address`
72963:   ./address
000055df5f024000      4K r---- address
000055df5f025000      4K r-x-- address
000055df5f026000      4K r---- address
000055df5f027000      4K r---- address
000055df5f028000      4K rw--- address
000055df604db000      4K rw---   [ anon ]
000055df604dc000    128K rw---   [ anon ]
00007fcbb3c2f000    136K r---- libc-2.31.so
00007fcbb3c51000   1504K r-x-- libc-2.31.so
00007fcbb3dc9000    312K r---- libc-2.31.so
00007fcbb3e17000     16K r---- libc-2.31.so
00007fcbb3e1b000      8K rw--- libc-2.31.so
00007fcbb3e1d000     24K rw---   [ anon ]
00007fcbb3e34000      4K r---- ld-2.31.so
00007fcbb3e35000    140K r-x-- ld-2.31.so
00007fcbb3e58000     32K r---- ld-2.31.so
00007fcbb3e61000      4K r---- ld-2.31.so
00007fcbb3e62000      4K rw--- ld-2.31.so
00007fcbb3e63000      4K rw---   [ anon ]
00007ffc58c6e000    132K rw---   [ stack ]
00007ffc58de5000     16K r----   [ anon ]
00007ffc58de9000      8K r-x--   [ anon ]
ffffffffff600000      4K --x--   [ anon ]
 total             2500K
```



0

128TiB

00007fffff... (48th bit and above, all 0s)

ffff80000... (48th bit and above, all 1s)

ffff... (64bit, all 1s)

Design choice to map all the kernel into the same address space

OS Memory management for segments

- **New processes** will ask for segments to be created.
- We may need to increase a segment's span/size for a process when its **segment grows**(eg it mallocs a lot of memory)
- Track the memory space available – free vs used parts.
- Assuming each segment has one base & bound, it must be **allocated contiguously**
- Each segment is smaller than the whole contiguous code, so we may be able to find the right 'holes'.
  - **External fragmentation**. Continues to be a problem.
- Thus we can have situations where we have available memory , but cannot accommodate a segment because of contiguity issues

Remember: Memory management is not just about hardware support for translation

Dealing with external fragmentation in segmentation

- Do allocations carefully to make sure contiguous spaces remain
  - First-fit, best-fit, worst-fit, buddy algorithms are example of the effort in this direction
- Compact the segments
  - Move the segments so that it creates contiguous free memory.
  - CPU and memory operation intensive, too much time
- But fragments continue to be a problem with segmentation
- Further more if a segment is used sparsely (because of a lot of mallocs and frees), that creates wastage within the logical segment!

Paging:
Solving the external fragmentation problem

Concept: “page frames” in the RAM

- Suppose we change the model of ***physical memory(RAM)***
  - it is considered to be chopped into **equal size chunks**, say 4K bytes each
  - These are called **page frames**.
- Thus the RAM is a sequence of page frames.
- For example if we have 64MiB bytes of RAM. How many page frames to we have ?
  - 64MiB = 2^6*2^10KiB =2^16 KiB
  - 4KiB = 2^2 KiB
  - Thus 64MiB/4KiB = 2^14 page frames = 16384 page frames
- **Core idea**: Each allocation is an integer number of fram
  - A segment or VMA gets a number of frames
  - The frames even inside a segment need not be contiguous

Aside: KB or KiB, MiB or MB …?

Concept: “Pages” in the address space

- Consider any virtual address space, perhaps with regions or segments.
- We may consider this address space as consisting of pages of fixed size... eg 4KB pages.

```
iiitb@iiitb-Vm:~/courses/zzz$ pmap `pidof address`
72963:   ./address
000055df5f024000      4K r---- address
000055df5f025000      4K r-x-- address
000055df5f026000      4K r---- address
000055df5f027000      4K r---- address
000055df5f028000      4K rw--- address
000055df604db000      4K rw---   [ anon ]
000055df604dc000    128K rw---   [ anon ]
00007fcbb3c2f000    136K r---- libc-2.31.so
00007fcbb3c51000   1504K r-x-- libc-2.31.so
00007fcbb3dc9000    312K r---- libc-2.31.so
00007fcbb3e17000     16K r---- libc-2.31.so
00007fcbb3e1b000      8K rw--- libc-2.31.so
00007fcbb3e1d000     24K rw---   [ anon ]
00007fcbb3e34000      4K r---- ld-2.31.so
00007fcbb3e35000    140K r-x-- ld-2.31.so
00007fcbb3e58000     32K r---- ld-2.31.so
```

Each of these is one page each in size

32 pages

34 pages

376 pages

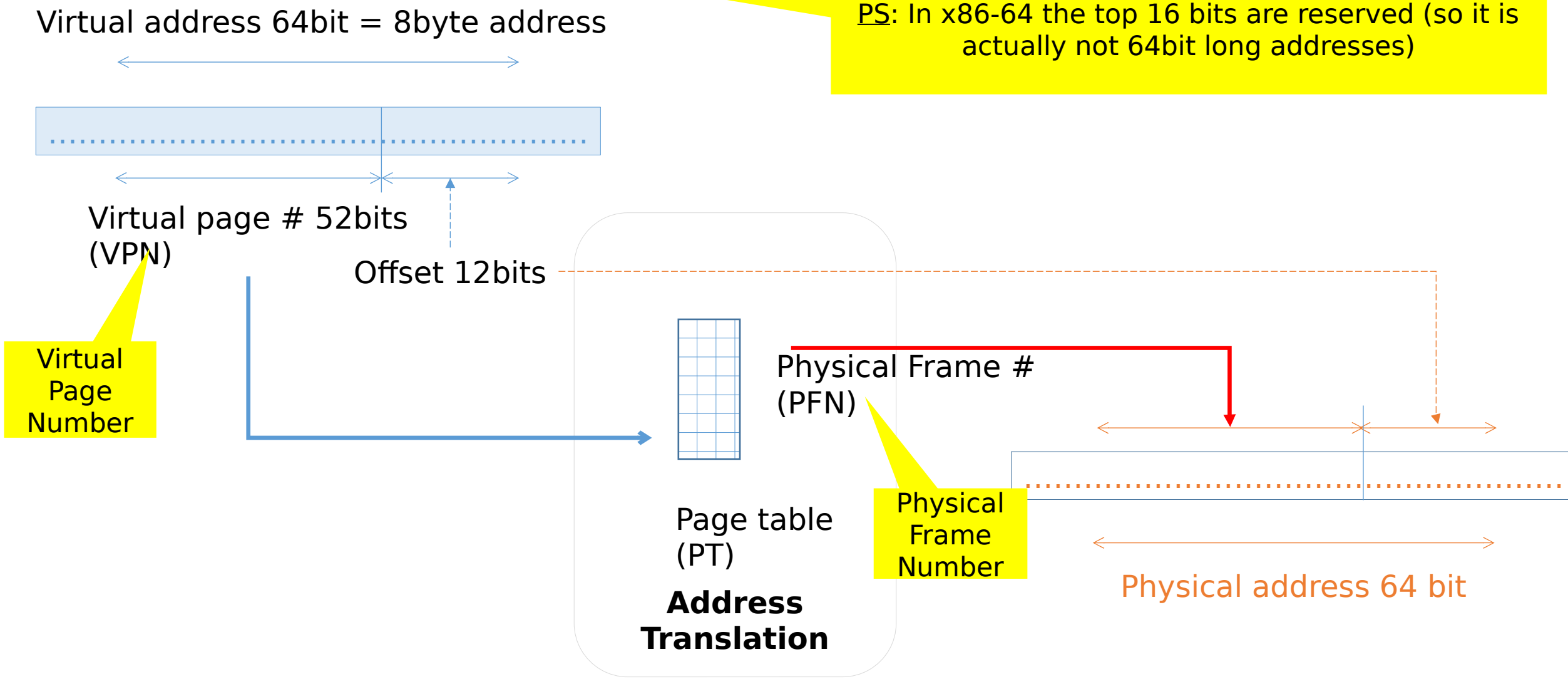
Mapping pages in address space to RAM page frames

Internal fragmentation is unavoidable

- When an executable is loaded it's segments/regions are each broken into units equal in size to a frame, i.e., the address space is split into (**virtual**) **pages**. And each page is allocated a **page frame** in the RAM.
  - (Virtual) Page → Page Frame …. Each page has its own "base address"=Frame No+…
- This eliminates the external fragmentation problem.
- The mapping or translation information is preferably done in the MMU. Where should the page table be?
- A simplified translation scheme is to use the lower order bits as offset in the (virtual) page and the higher order bits as (virtual) page number in the address space, the next slide illustrates. (recall seg)
- Question: Are large pages good or small ones?

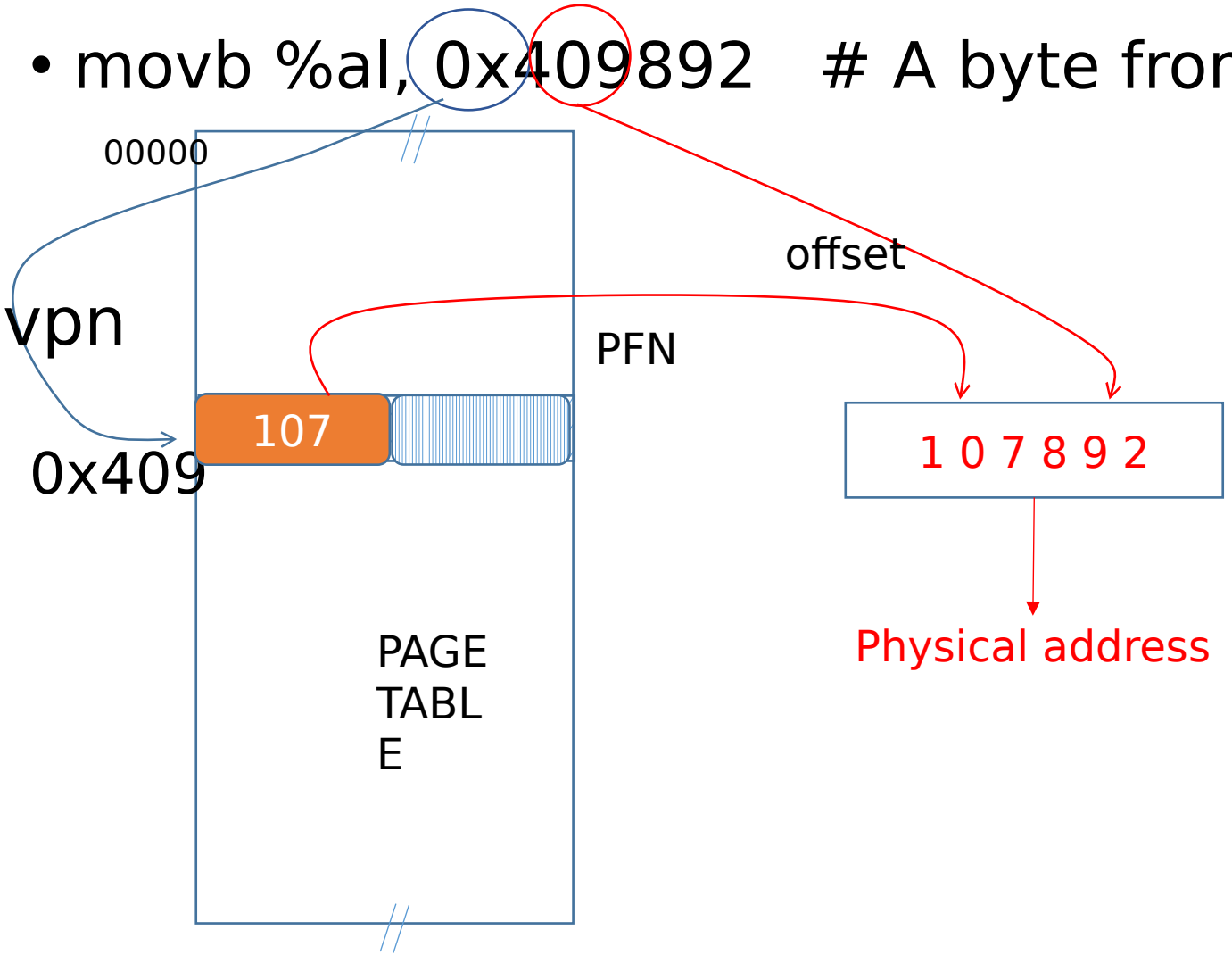
Address translation with frames

PS: In x86-64 the top 16 bits are reserved (so it is actually not 64bit long addresses)



An example translation – VPN and PFN

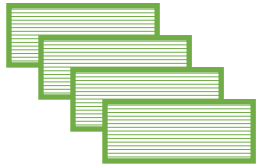
- movb %al, 0x409892 # A byte from al to 0x409892 in RAM



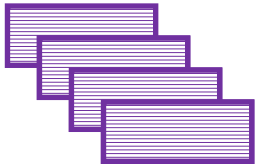
Sample view of frames allocated to processes

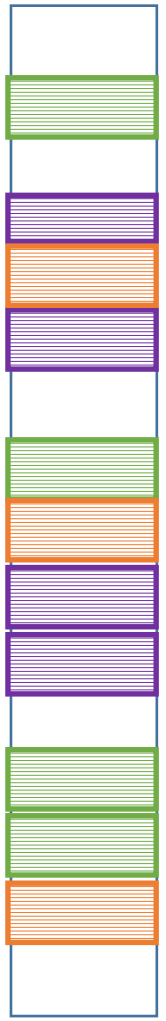
P1

P2



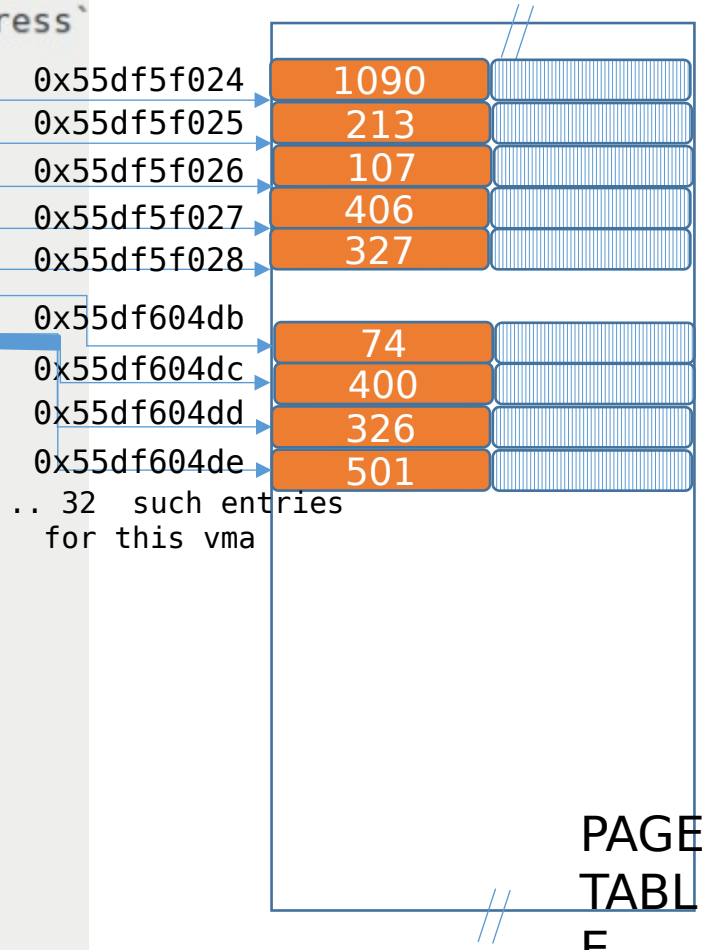
P3

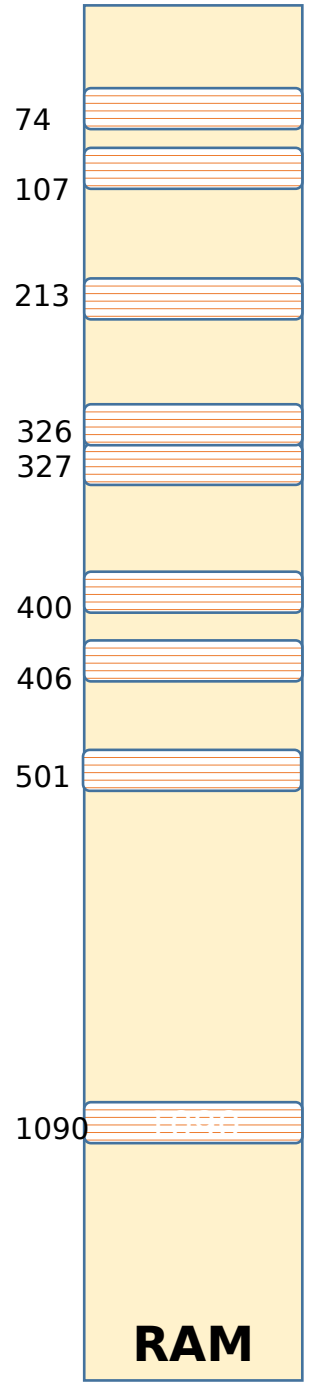




Example partial mapping of our example virtual address space

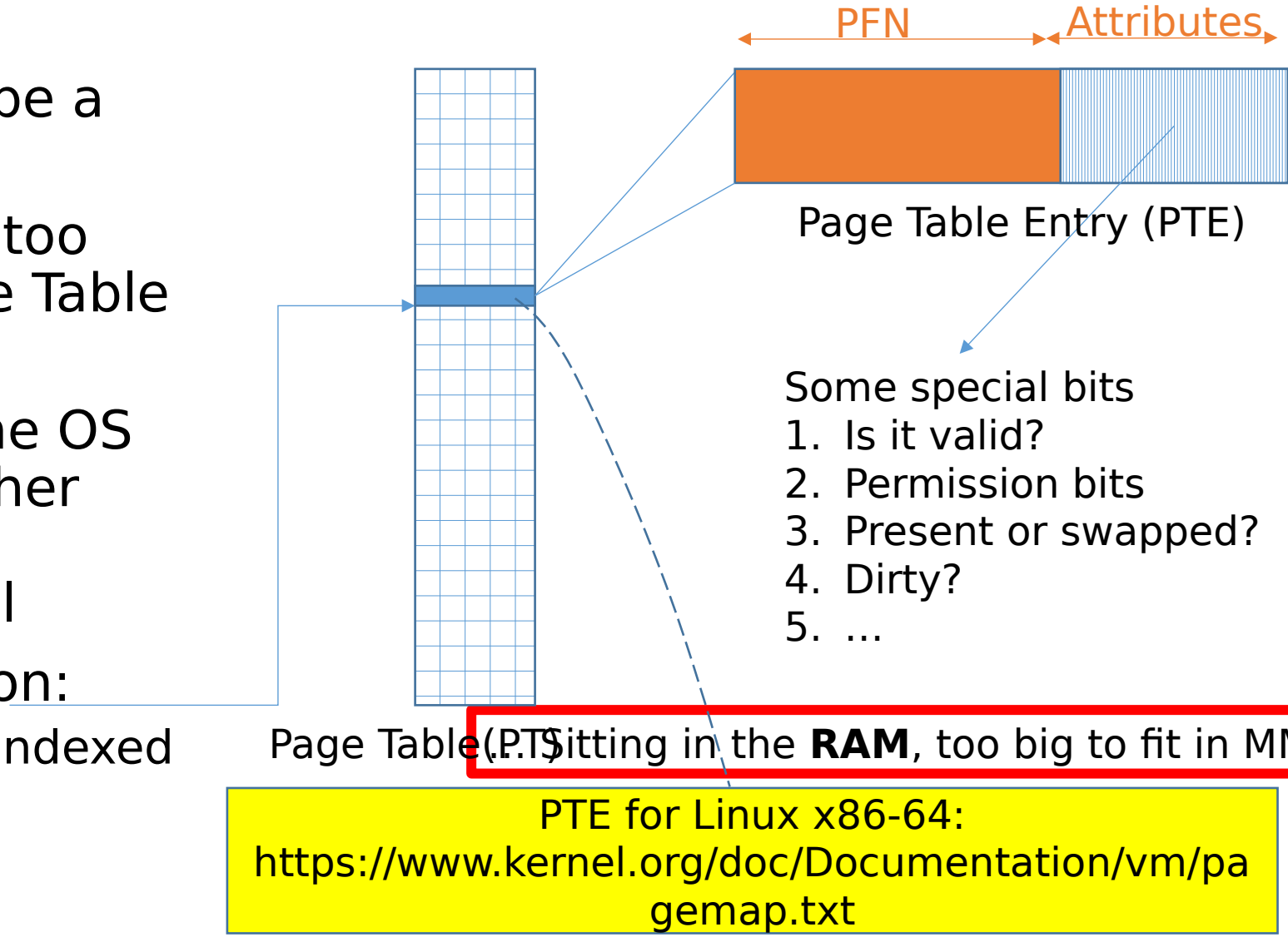
```
iiitb@iiitb-Vm:~/courses/zzz$ pmap `pidof address`
72963:   ./address
000055df5f024000      4K r---- address
000055df5f025000      4K r-x-- address
000055df5f026000      4K r---- address
000055df5f027000      4K r---- address
000055df5f028000      4K rw--- address
000055df604db000      4K rw---   [ anon ]
000055df604dc000    128K rw---   [ anon ]
00007fcbb3c2f000    136K r---- libc-2.31.so
00007fcbb3c51000   1504K r-x-- libc-2.31.so
00007fcbb3dc9000    312K r---- libc-2.31.so
00007fcbb3e17000     16K r---- libc-2.31.so
00007fcbb3e1b000      8K rw--- libc-2.31.so
00007fcbb3e1d000     24K rw---   [ anon ]
00007fcbb3e34000      4K r---- ld-2.31.so
00007fcbb3e35000    140K r-x-- ld-2.31.so
00007fcbb3e58000     32K r---- ld-2.31.so
00007fcbb3e61000      4K r---- ld-2.31.so
00007fcbb3e62000      4K rw--- ld-2.31.so
00007fcbb3e63000      4K rw---   [ anon ]
00007ffc58c6e000    132K rw---   [ stack ]
00007ffc58de5000     16K r----   [ anon ]
00007ffc58de9000      8K r-x--   [ anon ]
ffffffffff600000      4K --x--   [ anon ]
 total             2500K
```





Page table – where ? What does it contain?

- Clearly the MMU would be a good place
- However, there are just too many(howvmany?) Page Table Entries (PTEs).
- So we often keep it in the OS in the RAM like many other pieces of per process information in the kernel
- PT Structure & Translation:
  - A simple scheme: PT is indexed by the VPN
  - “Linear Page Table”



Invalid pages

- Note that not all pages (ie VPNs) are used by a processes
- In our mapping example (using pmap) we saw a small fraction of the possible pages of the address spaces is actually used.

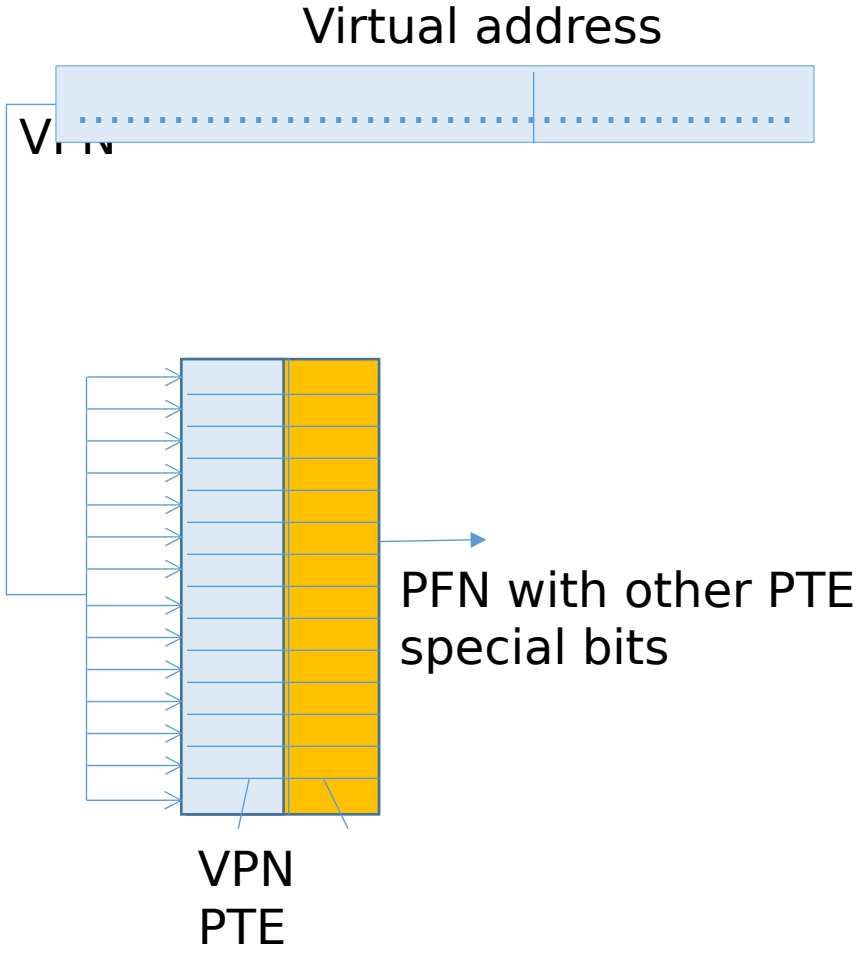
- The PTE tracks this by using an **invalid bit** for the corresponding VPN.
- Accessing an invalid page causes an illegal address exception

Translation with paging can be slow

- Since the PT is somewhere in the RAM, each translation requires somehow indexing it, then pulling out the PTE, and then doing the appropriate checks, followed by creating the new address, which finally goes out on the memory bus.
- This is to be done for each instruction.
- Latency – to access one memory location (say some variable in the program x) we need to access another memory table called the PTE, in that we need to pickup the element at a certain index, etc.
- That would be exceedingly slow, right?
- This is where TLBs (*Translation Lookaside Buffers*) come in handy!

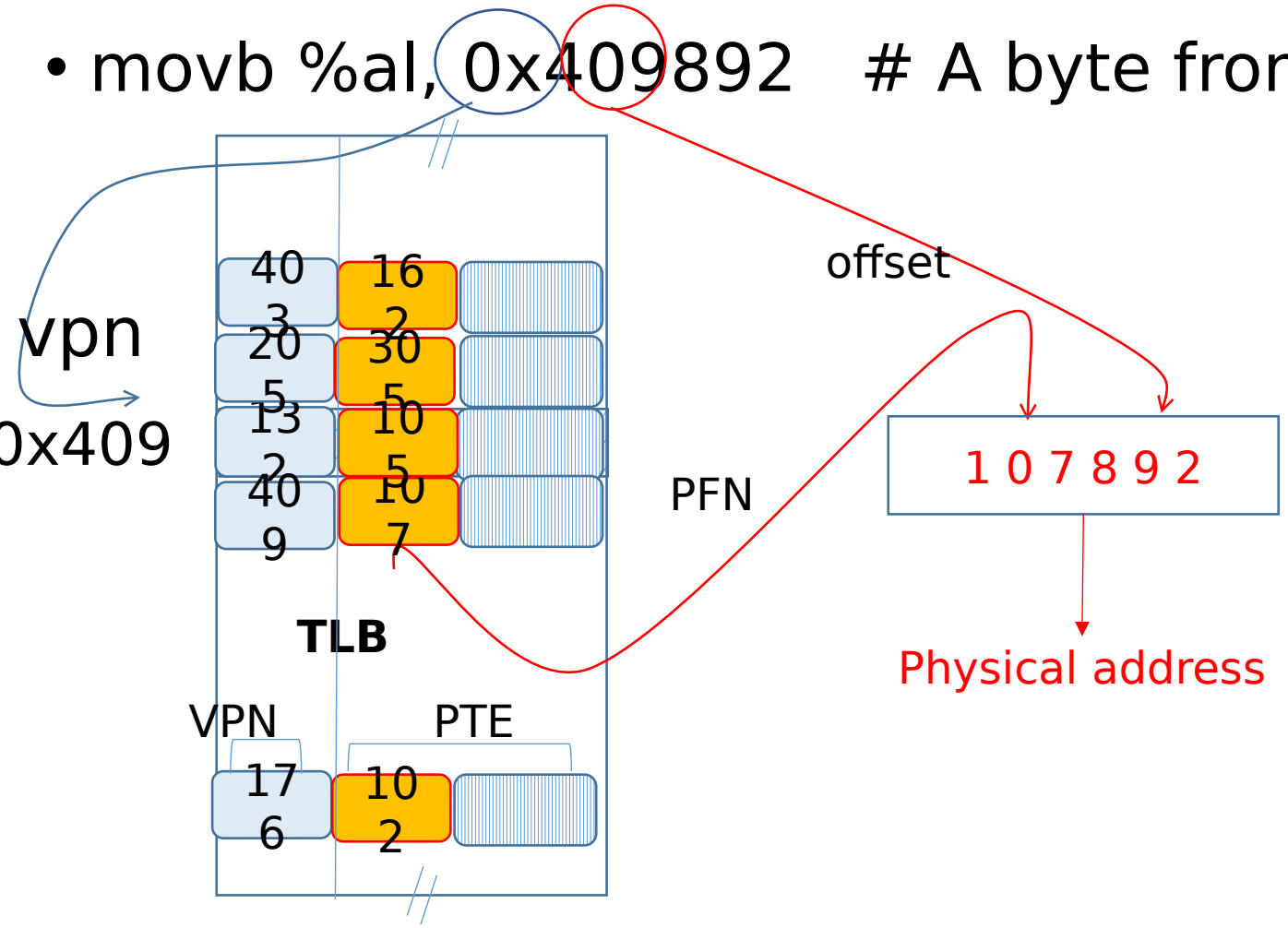
TLB’s to the rescue

- The idea of the MMU was to be able to do translation in hardware immediately.
- We can do that if we have the just the right PTEs in the MMU a cache for PTEs.
- The hardware support for that in the MMU – “Translation Look-aside buffer”.
- TLBs are crucial to making paging work.
- Why not have all the PTEs in the TLB?
- How does the TLB work?...



An example translation – VPN and PFN **from TLB**

- movb %al, 0x409892 # A byte from al to 0x409892 in RAM



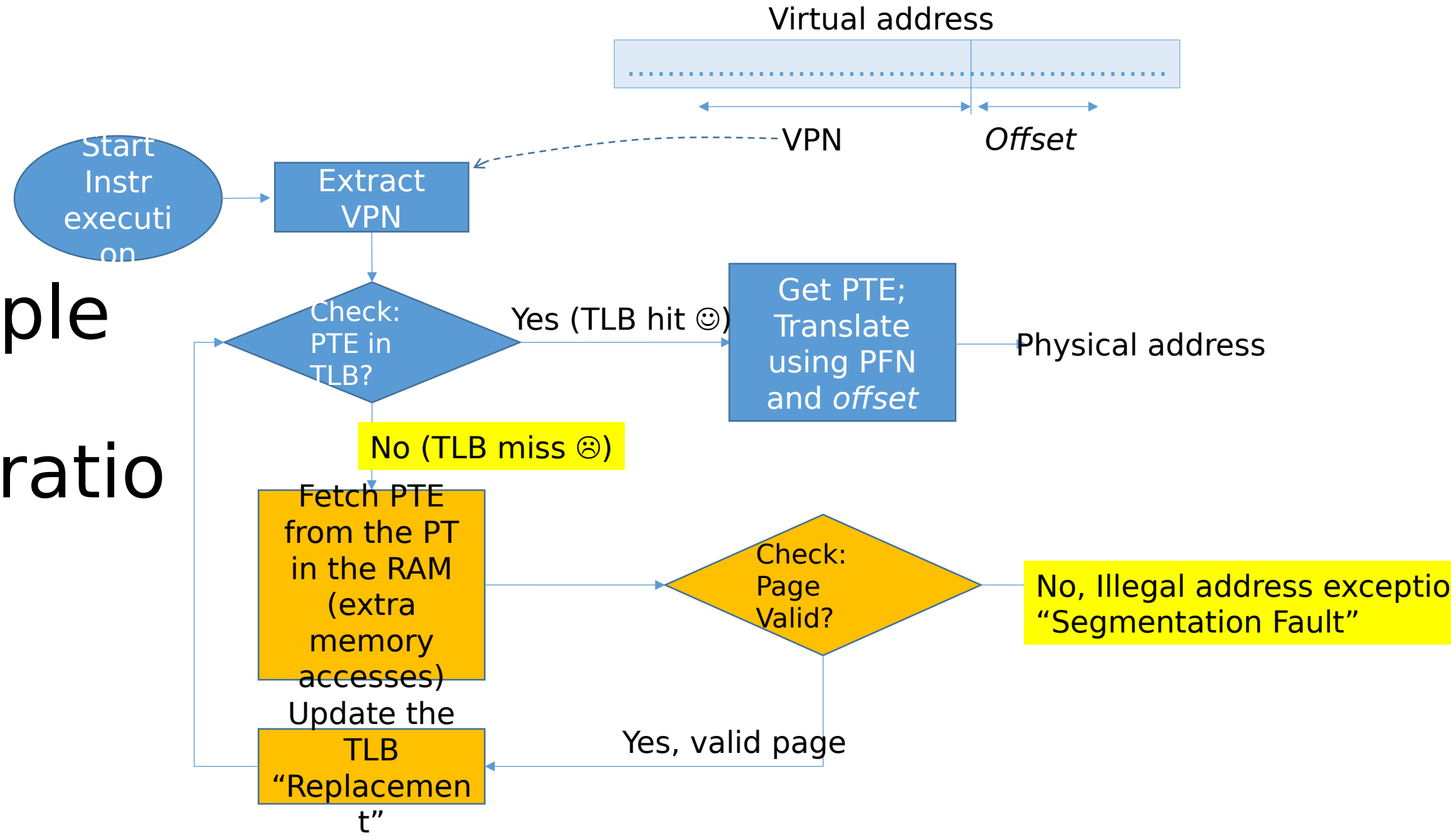
What does a TLB entry look like?

- A TLB entry has **both** the VPN and the PFN in it in addition to other bits in the PTE
- This is because each PTE can be in any position in the TLB.
- TLB entries are **not** address accessed
- TLB entries are entries are all searched at the same time for a given VPN and the appropriate PFN is got if available
  - Special hardware –**fully associative** caches, content addressable memory.
- Since TLB is a cache it is small, TLB misses are not uncommon
- Replacement policy kicks in when the TLB is full
  - LRU? Random ? …

TLB Misses are costly!

- It only works if TLB misses (can they be avoided?) are few and far in between
- Works where there is
  - Temporal locality (the same variables are reused over time)
  - Spatial locality (variables physically close by are accessed together)
- It would help to layout variables so that there is spatial and temporal locality… job of the compiler and linker.
- TLB misses are **a special exception** to be handled like other exceptions. The *handler* makes sure to populate the right PTE in the TLB.
- Some architectures allow the hardware to do that by putting the starting point of the page table in a special **page table base** register.

Simple TLB operation

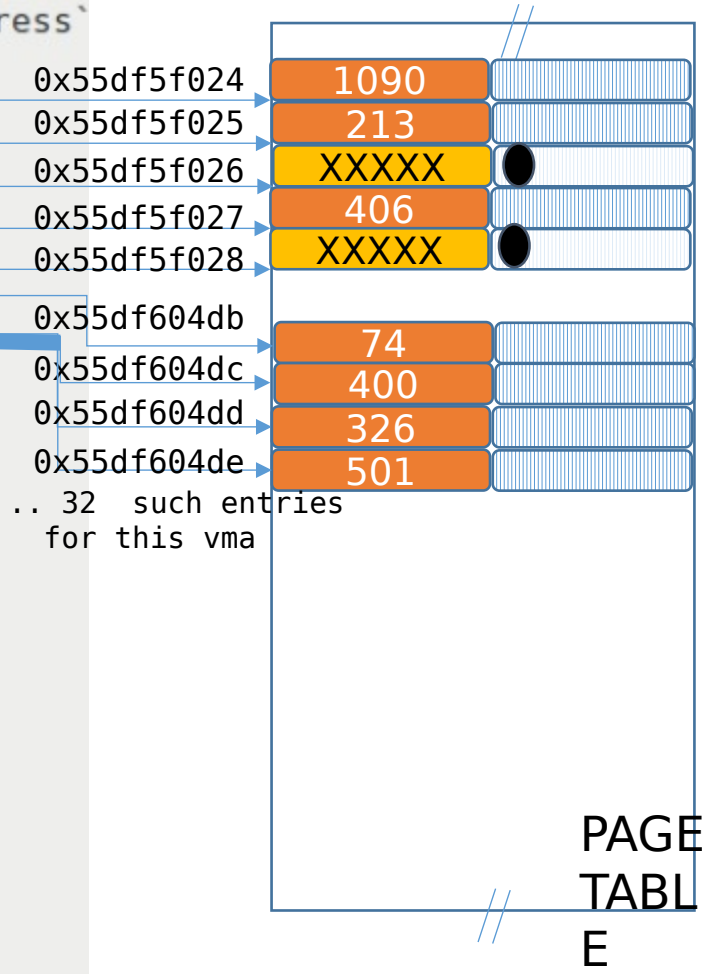


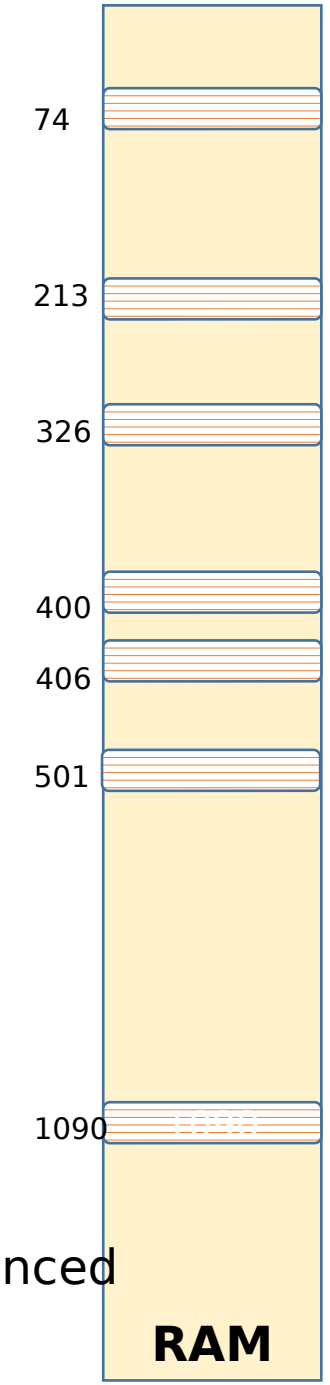
Demand paging

- Most modern OS’s don’t bring all the pages of the process into the memory at the beginning.
  - This means some of the VMAs may not be in the RAM
- They are brought in only when there is a need i.e., when the page is referenced
  - This is called **demand paging**
- This idea allows more processes to have useful pages in the RAM at the same time.
- It also means sometimes we wont find pages of a process in the RAM
  - So the PTE will also track this using a ‘present’ bit

Page Table with demand paging showing two unloaded pages

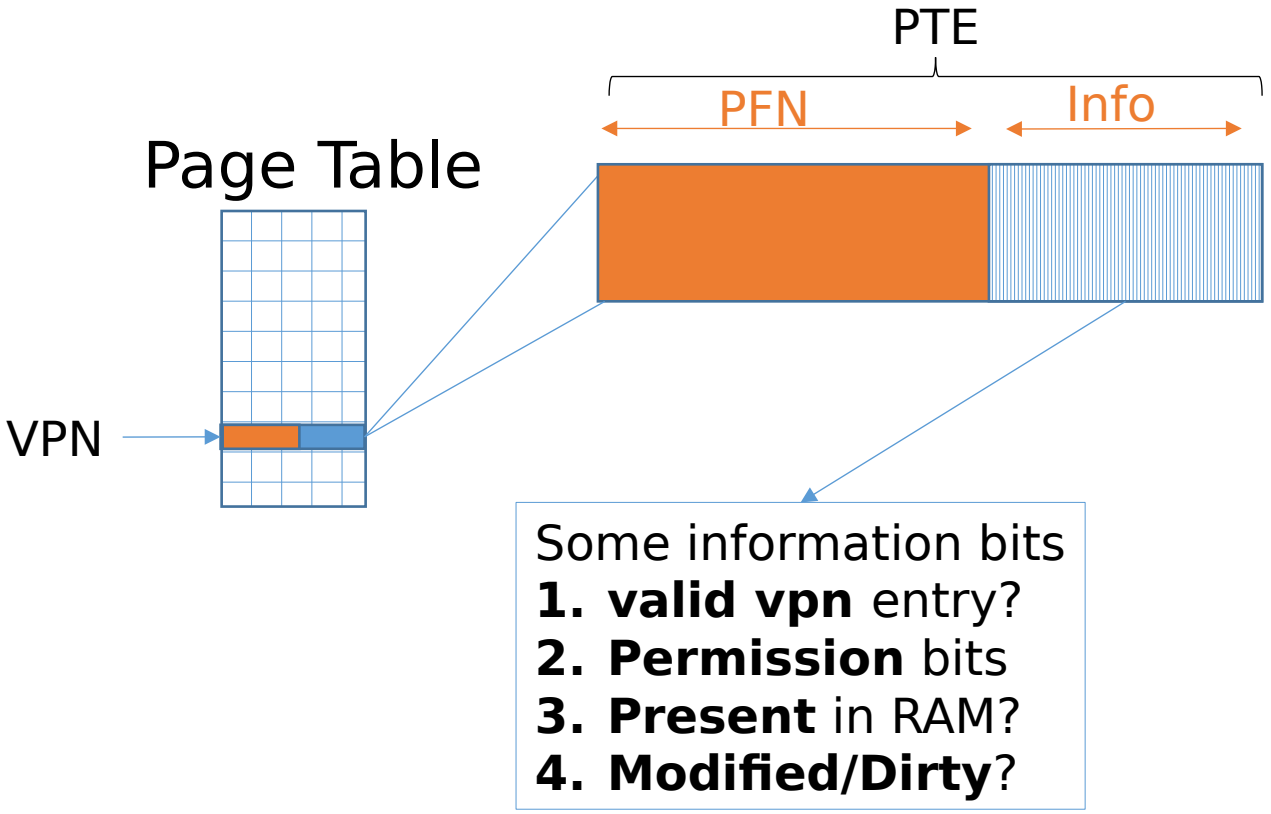
```
iiitb@iiitb-Vm:~/courses/zzz$ pmap `pidof address`
72963:   ./address
000055df5f024000      4K r---- address
000055df5f025000      4K r-x-- address
000055df5f026000      4K r---- address
000055df5f027000      4K r---- address
000055df5f028000      4K rw--- address
000055df604db000      4K rw---   [ anon ]
000055df604dc000    128K rw---   [ anon ]
00007fcbb3c2f000    136K r---- libc-2.31.so
00007fcbb3c51000   1504K r-x-- libc-2.31.so
00007fcbb3dc9000    312K r---- libc-2.31.so
00007fcbb3e17000     16K r---- libc-2.31.so
00007fcbb3e1b000      8K rw--- libc-2.31.so
00007fcbb3e1d000     24K rw---   [ anon ]
00007fcbb3e34000      4K r---- ld-2.31.so
00007fcbb3e35000    140K r-x-- ld-2.31.so
00007fcbb3e58000     32K r---- ld-2.31.so
00007fcbb3e61000      4K r---- ld-2.31.so
00007fcbb3e62000      4K rw--- ld-2.31.so
00007fcbb3e63000      4K rw---   [ anon ]
00007ffc58c6e000    132K rw---   [ stack ]
00007ffc58de5000     16K r----   [ anon ]
00007ffc58de9000      8K r-x--   [ anon ]
ffffffffff600000      4K --x--   [ anon ]
 total             2500K
```





These two pages are somewhere on the disk and will be loaded when referenced
Access to pages which are valid but not in RAM causes a page fault

PTE **attributes/ information bits** and **Page faults**



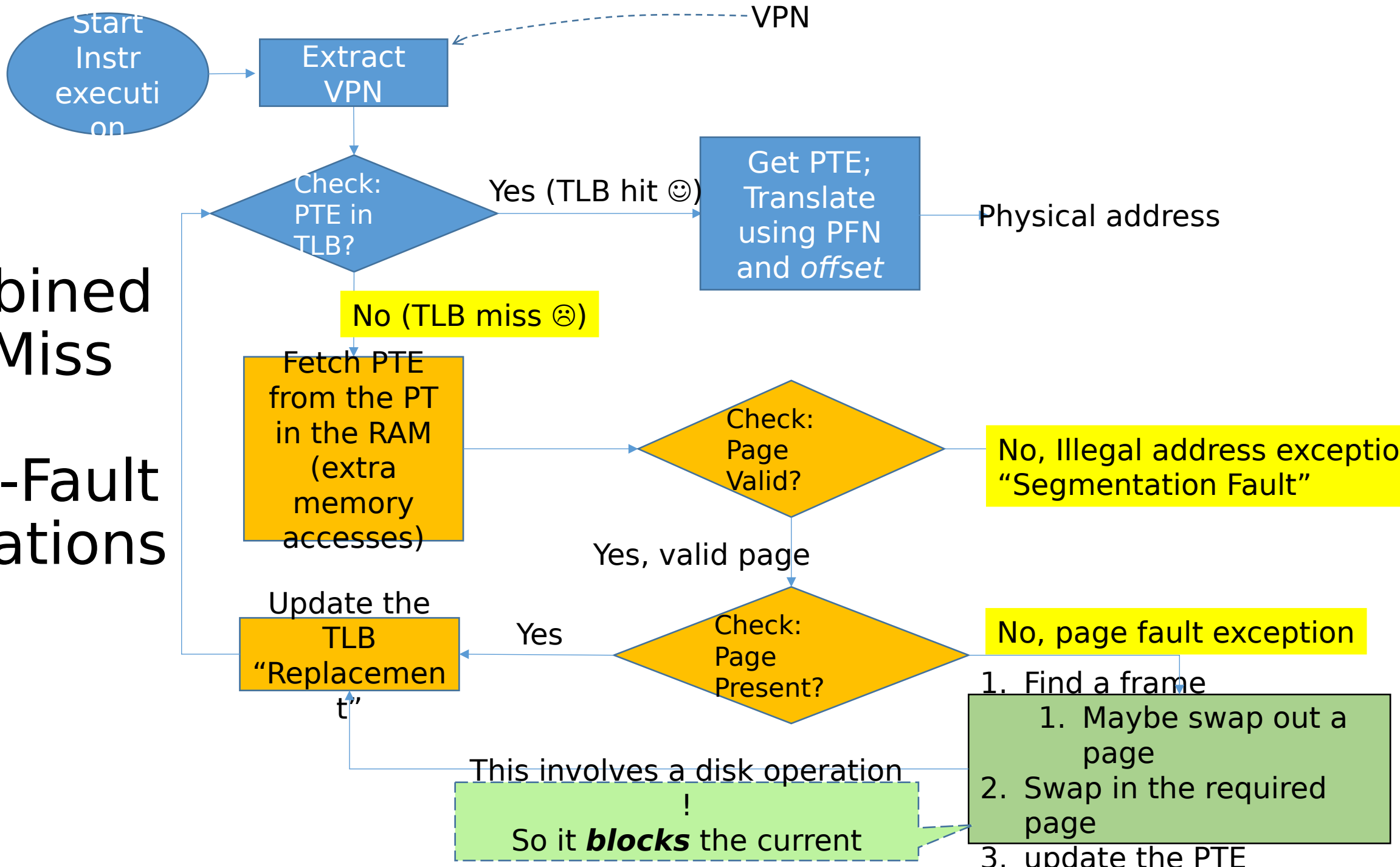
TLB-Miss Handler Operation:

- If PTE is **invalid**:
  - ***An Illegal address exception,***
- What if it is **valid**, but **not present** ?
  - Means the page not in RAM
    - Must be on the disk!
  - A ***page fault exception***

Page Fault Handler Operation:

- Finds a free frame (frame allocation)
- Loads the frame with the page from the disk
- What if there are no free frames?
  - Free up a frame (frame replacement)
    - Puts its contents on disk - “swap out”

Combined TLB-Miss and Page-Fault operations

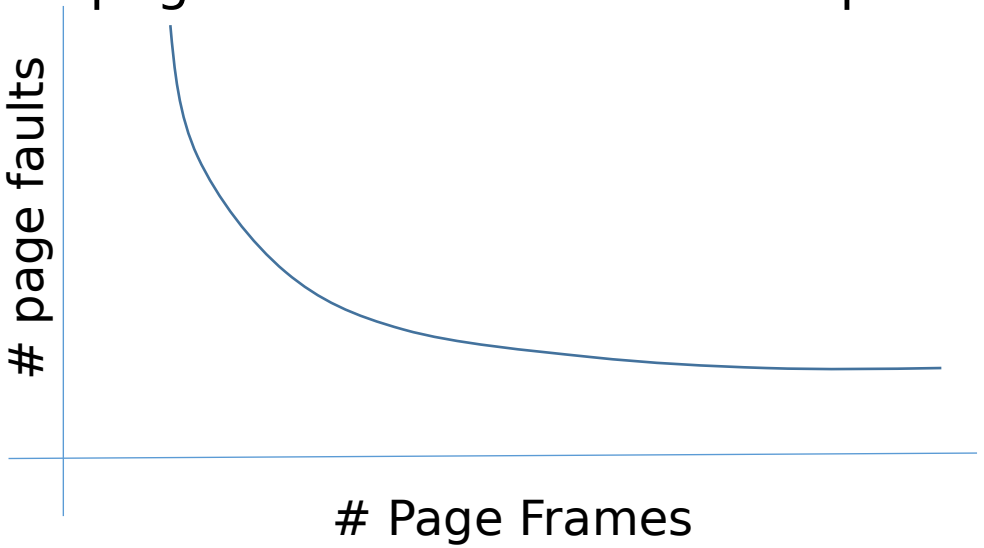


1/3 Find a frame for a accessed page

- Pick a frame from the list of free frames (all frames are equal)
- If all the frames are full…
  - You need to replace a frame holding a page – a “victim page”
- Algorithms for picking the “victim page”:
  - Random
  - FIFO
  - LRU and its variants…. With some hardware support

Laszlo Belady's anomaly

- More RAM is usually a good thing, would cause fewer page faults. So we would expect -



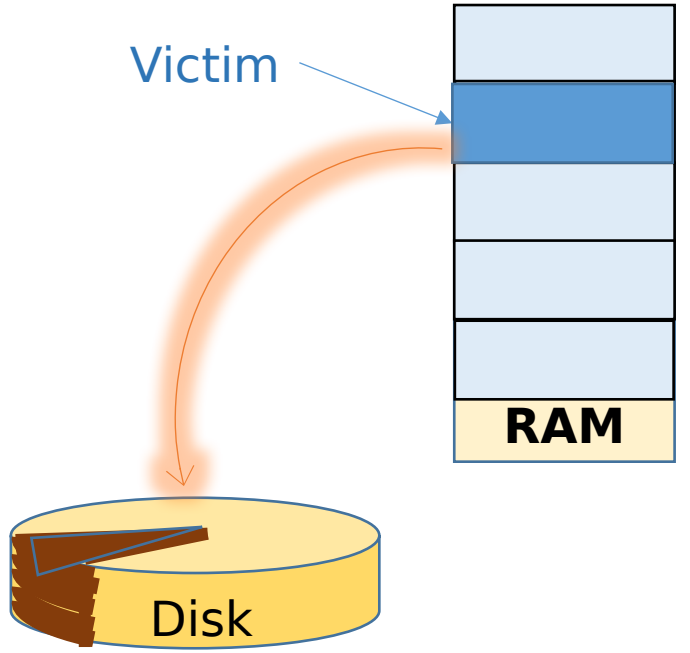
- Belady showed that is not always the case
  - It depended on
    1. The access pattern of the frames
    2. The replacement algorithm
  - In particular FIFO could create a problem

The anomaly is that for some algorithms and some access patterns it is *not* a simple trend:

- Increasing frames may increase faults!
- An example page access sequence where **FIFO** displays this anomaly:
  - 3 2 1 0 3 2 4 3 2 1 0 4
  - Note: This is with just one process
  - Try with #frames=3 vs #frames=4
- Good news: **Does not** happen with **LRU**

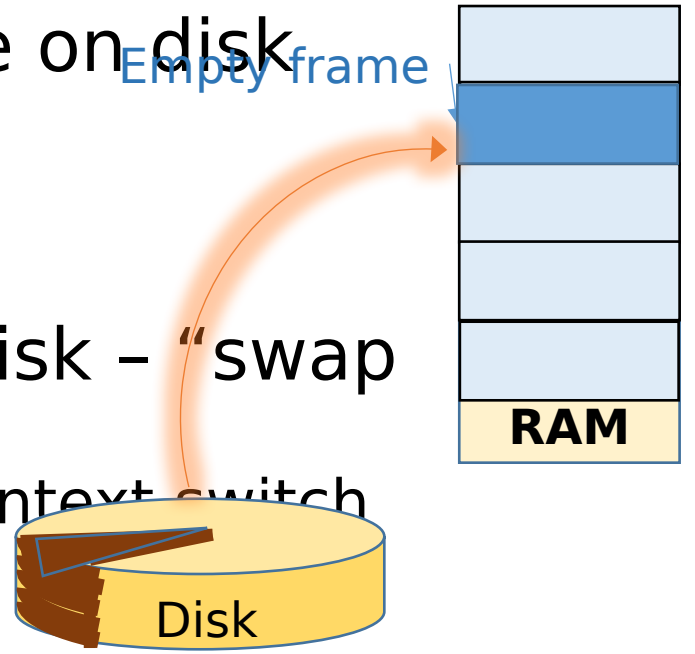
1.1/3 Swap-out the “victim page”

- The “victim” page may be needed later on, so save it on disk (“swap out”) if needed
  - usually in an area called ‘swap area’.
  - On Linux see: `swapon --show`
  - This is a disk write – so will be slow.
    - This means it causes blocking
    - Thus an opportunity to context switch.
- The corresponding PTE has to be updated
  - Not ‘present’ anymore
  - Must save information on where to find the swapped out page



2+3/3 Swap in the accessed page from the disk

- We have an empty frame now.
- The accessed (“faulted”) page is somewhere on disk
  - Usually in the `swap area’
  - Sometimes in a normal file.
  - The PTE has this information
- Read in the contents of this page from the disk – “swap in”
  - Again a slow operation, and an opportunity to context switch
- Update the PTE
  - The page is now present and has a new PFN



The PTE and frames information on Linux:

- Run a program or choose one that is already running, like `bash`
- pmap PID shows you the vma map
- **`/proc/PID/pagemaps`** file has the pagetable as if it were a linear page table.
  - The data is not in ASCII, but in binary
  - Each PTE indexed by the VPN is 8 bytes long (64 bits)
  - It is possible to read it if you are root
    - See http://fivelinesofcode.blogspot.com/2014/03/how-to-translate-virtual-to-physical.html for one code example of how to extract PFN.
    - Remember to run it with **sudo** else reading will fail.
- You can also read more about the page frames themselves by looking at **`/proc/kpageflags`** which is indexed by the PFN
- See example on github: `memory/frameit.c`

These /proc/PID files are documented here:
https://www.kernel.org/doc/Documentation/vm/pagemap.txt

Reducing the Swap-out time – “Dirty bit”

- Swapping out is
  - A time consuming process (involves disk IO)
- Sometimes there is no need to swap out
  - Maybe the page is as on the disk – i.e., it was not modified at all, only read
- Therefore the PTE has a “Modified” bit, also called “dirty” bit.
  - This is ‘clean’ when a page is loaded from the disk
  - This is set whenever a page is written to.
- Swap-out only happens when the “Dirty” bit is set.
- **Thrashing** is when you have too much swap-in swap-out. Process makes very slow progress.
- Aside - Why it makes sense to match page size with disk block size.

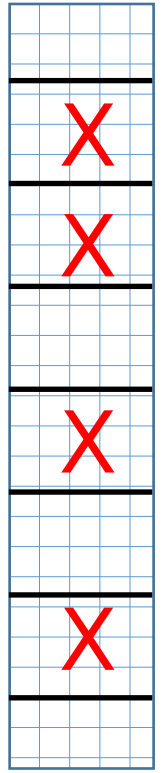
Dealing with large Page Tables in RAM

- Our Linear page tables are large
  - 52 bits or 36 bits means a lot of entries
  - $2^{36} \times 2^{3}$ bytes assuming PTE size of 8 bytes giving a total size of 512 GB
  - Infeasible by any practical standard!
- Also often the page table is sparsely populated
  - We only want PTEs for a process for which there are VPNs

- It therefore makes sense to think of alternate Page Table organization
- One idea is to note that lots of contiguous VPNs are likely to be invalid. We can make use of this fact.

How to deal with large page tables - Directory

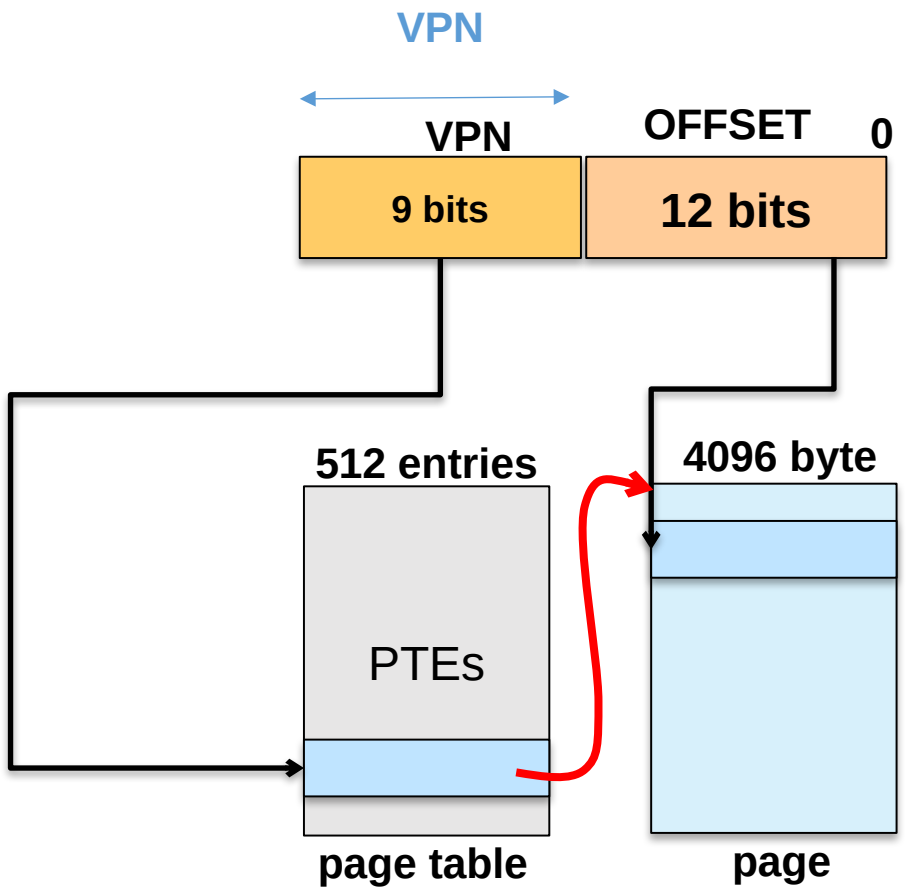
- Recall that the Page Table is in the RAM
- So it occupies a number of frames.
- If a whole frame worth of PTEs are invalid, we just don’t have to keep a frame reserved for that.
- We can use the address splitting idea to split the VPN for this purpose
  - And our PTEs will only be contiguous within a frame.



Page Table in the RAM is spread over multiple frames

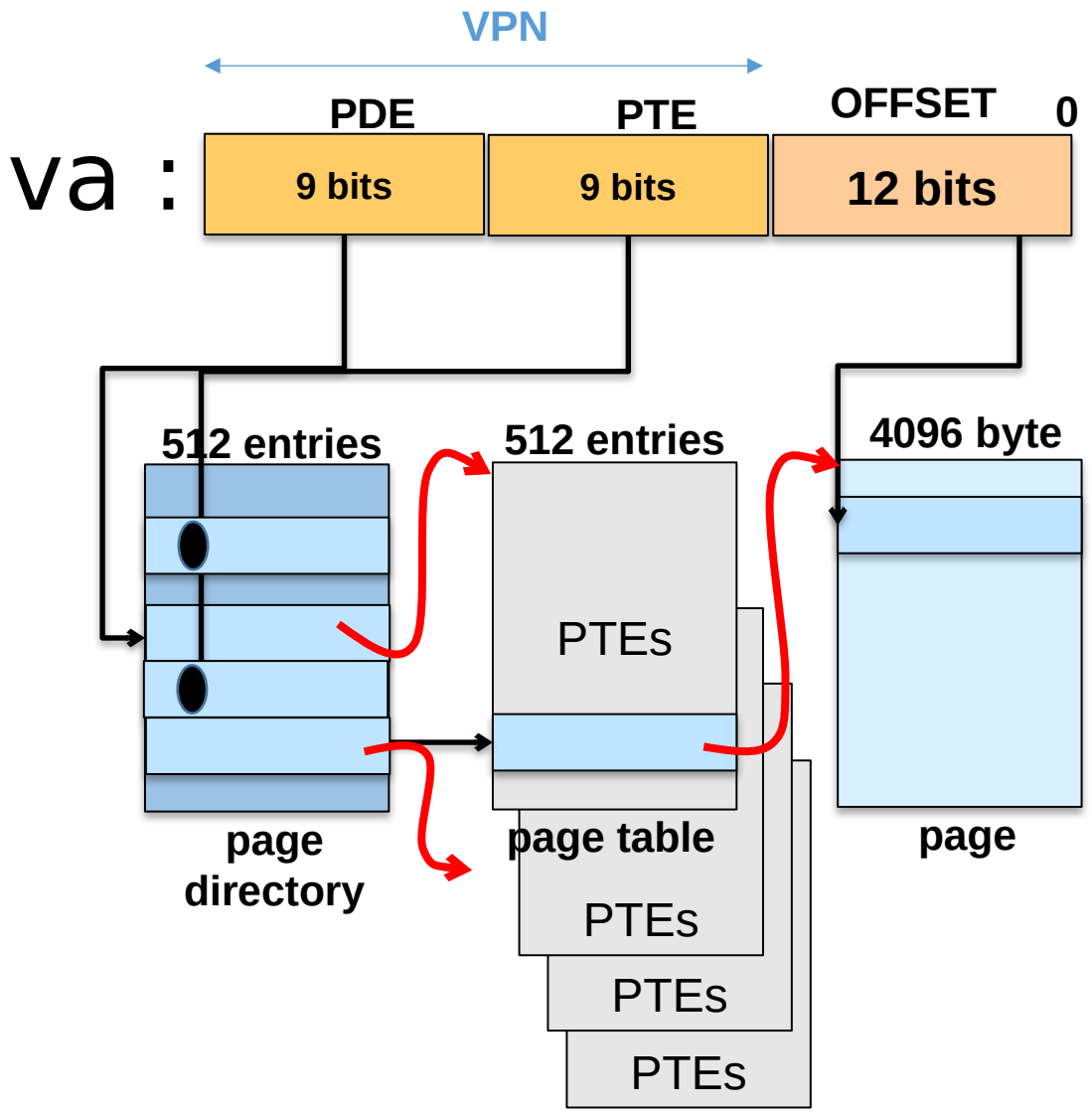
Towards Page Table Hierarchy on x86-64

- In a small world we just have a PTE which is one page big
- Happens if our VFN is small, say 9 +12=21 bits.
- The 9 bits then index into the PTE as shown
- In reality we have a lot more than 9 bits in our VPN



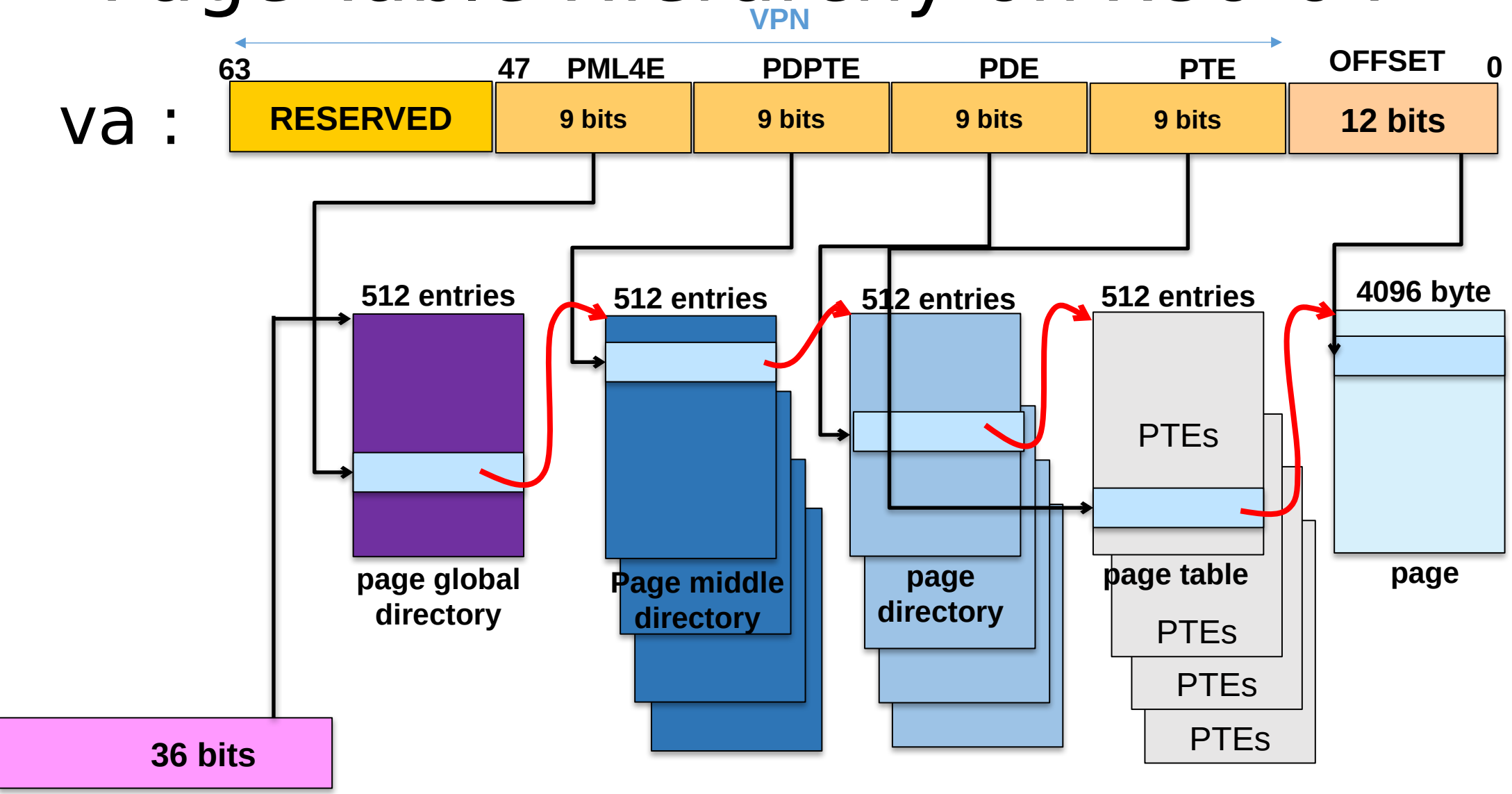
Towards Page Table Hierarchy on x86-64

- We could use the next 9 bits as an index to a Page holding pointers to a page worth of PTEs
- We could thus point to multiple pages of PTEs.
- This definitely appears as a complicated way to do a linear table.
- However, we could play a trick: if a page worth of PTEs is going to be invalid, just dont allocate a page!
- We can further extend this idea…



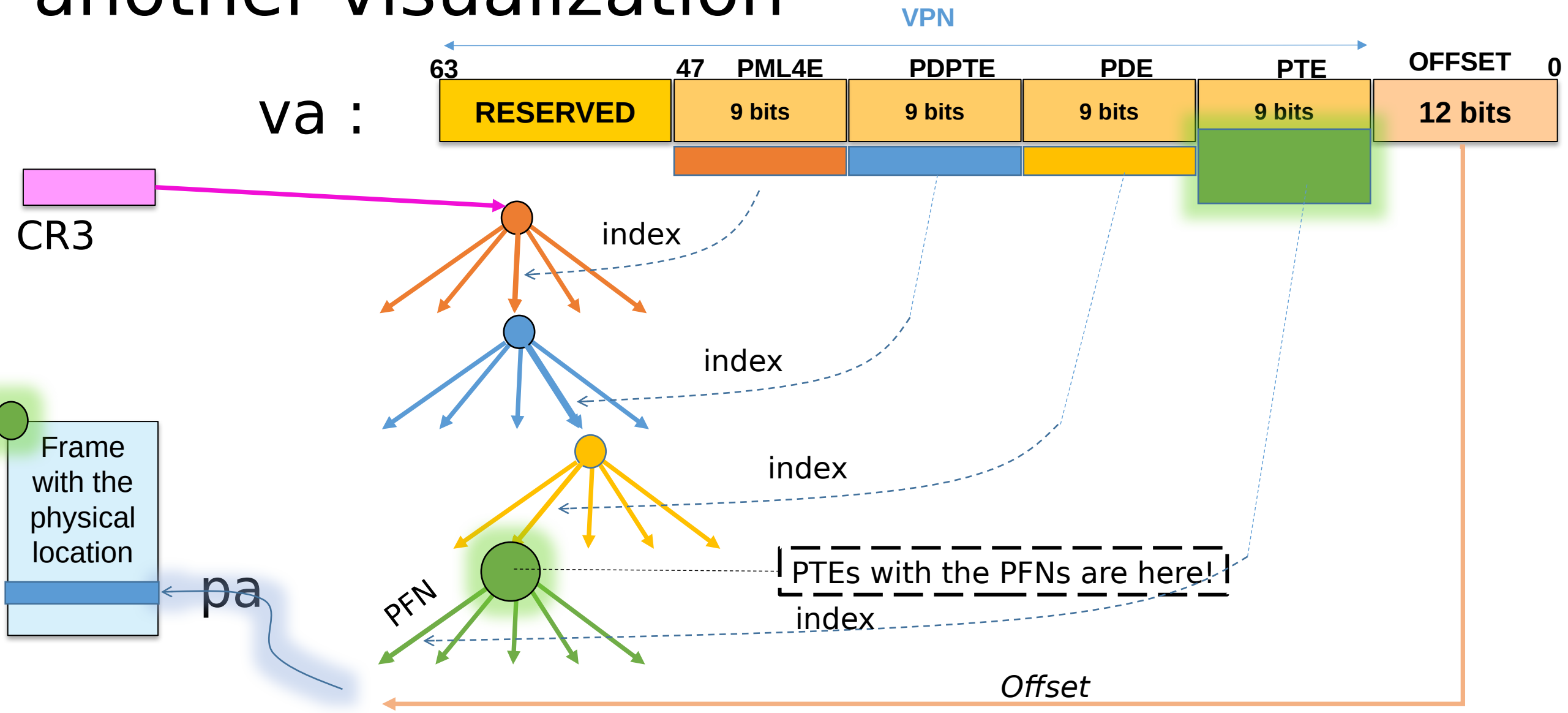
Page Table Hierarchy on x86-64

va :



X86-64 understands this hierarchy and 'walks' through it (as a hardware operation) when a TLB miss occurs!

Page Table Hierarchy on x86-64 – another visualization



As an aside

- Most modern Intel processors support something called
  - Program Context ID (also called address space ID)
- This is inside CR3
  - It may be used to allow multiple processes' PTEs to exist in the TLB at the same time.
  - Linux uses this to support PTE’s from multiple processes to be in the TLB at the same time. Reduces TLB flushing.

Copy on write – Avoiding “duplicate pages”

- Some pages like the code for executables, especially libraries, are heavily shared among processes. This saves a lot of duplicate frames.
- Further when a process forks, not only the code, but the data pages also initially stay the same
- A copy is only made when it is actually written to.
- This is called **copy on write** (COW) page usage.

Memory allocation from a contiguous memory pool (part of address space)

- malloc() … usually gets a chunk of memory from the OS (in pages) and manages itself.
- How to track free memory pieces
  - (a) split (b) coalesce
  - Maintain a linked list of free pieces
- The buddy method
- In the kernel another trick is also used – slab allocation
  - Basically pre-create a number of typically used small structures and keep them available for immediate allocation, even pre-initialized.
  - See /proc/slabinfo

Summary – Memory Management

- AIM: Support(TEP) multiple programs to be resident at the same time
- Translation is the key
- Base and bound is a core idea – with HW Support – MMU
- Segmentation provides more protection, more flexibility
- Introducing page frames: Paging - removes external fragmentation, Page Tables in RAM
- Paging is slow, because it needs too many “PTEs” in the MMU – wont happen.
  - TLBs are to our rescue
  - Caches a small set of PTEs in the MMU
- Dealing with TLB-Miss and Page-fault and the role of swapping
- But the PT can become large too
  - Hierarchical PTs
- We saw a couple of tools along the way to explore how this happens in Linux/x86-64

Ends the section on Memory Management