A detailed example

virtual address space = 16 KB, page size = 64-byte

2^{14} ($2^4 \cdot 2^{10} = 2^{14}$), 2^6

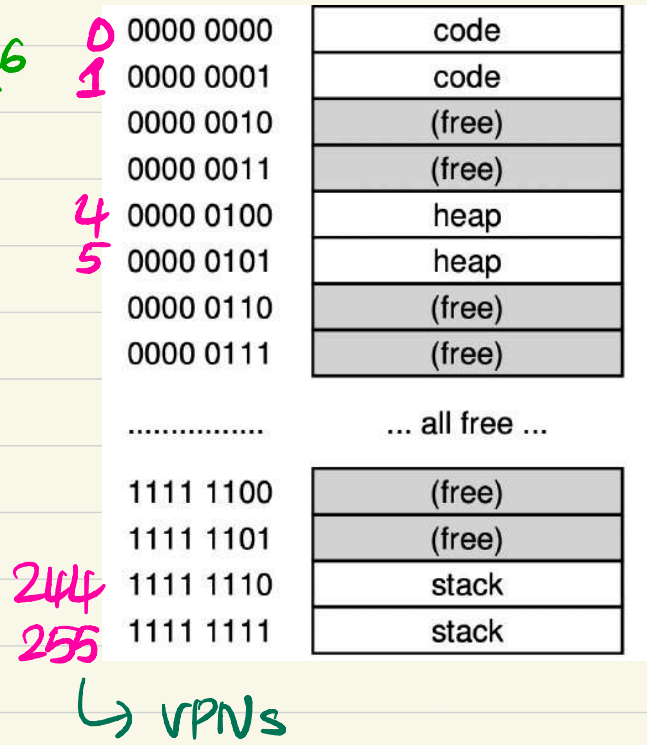
$$\#\text{ virtual pages} = \frac{2^{14}}{2^6} = 2^8.$$

$\Rightarrow$ # bits needed for VPN = 8

Size of a linear page table = 2^8 x (4 bytes)

(4 bytes: size of PTE)

$= 2^{10} = 1$ KB in size

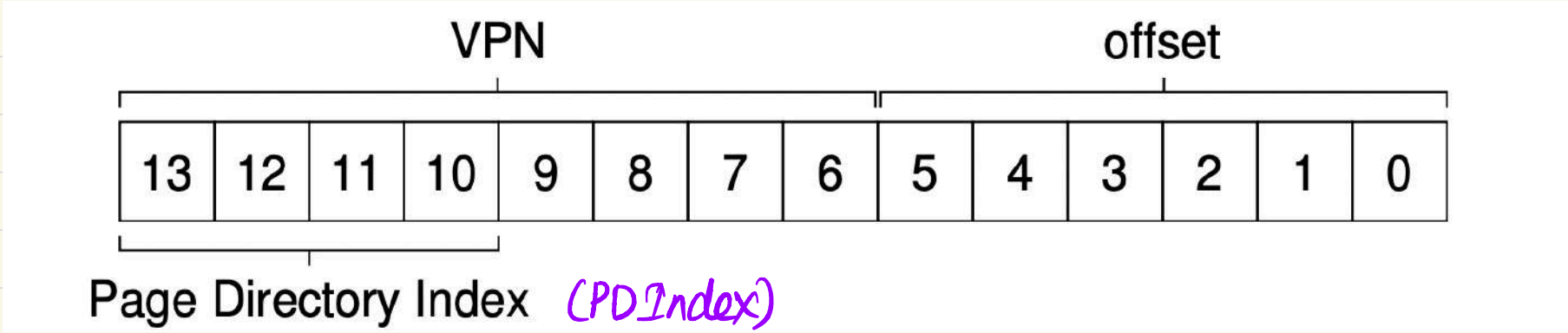


In multi-level table, we want to divide the above page table into pages (64-byte)

$\Rightarrow$ # pages needed to store page table = $\frac{2^{10}}{2^6} = 2^4 = 16$ pages

16 pages: # pages in the page table directory

$16 = 2^4$ $\Leftarrow$ 4-bits for page directory index.



How to extract PDE address (PDEAddr) from PDIndex?

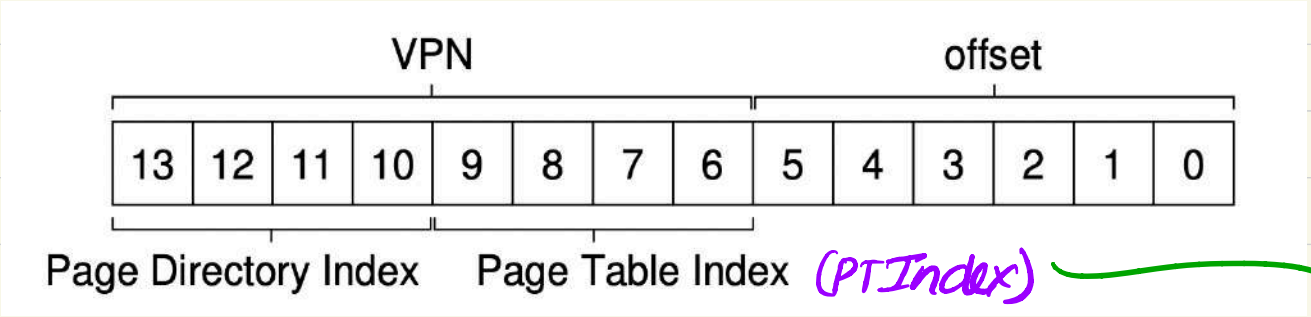
$$PDEAddr = PageDirBase + (PDIndex * sizeof(PDE))$$

If PDE is invalid → raise an exception.
What if PDE is valid? need to fetch PTE. How?

$$\#PTEs \text{ in a single page} = \frac{64\text{-byte}}{4\text{-byte}} = \frac{2^6}{2^2} = 2^4 = 16$$

⇓

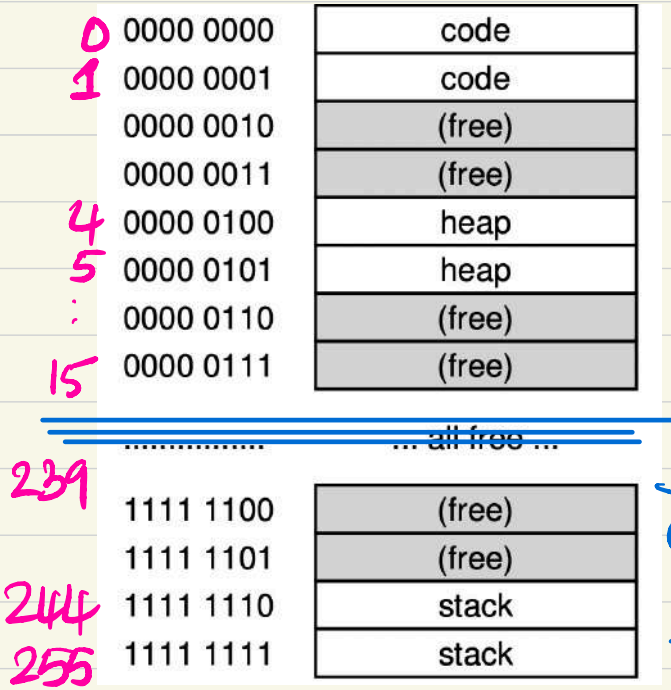
4-bits needed for PTIndex

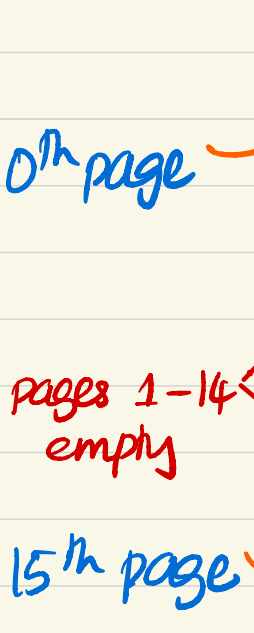


How to extract PTE address (PTEAddr) from page table Index?

$$PTEAddr = (PDE.PFN \ll SHIFT) + (PTIndex * sizeof(PTE))$$

(PFN at PDEAddr in page directory)





Page Directory		Page of PT (@PFN:100)			Page of PT (@PFN:101)		
PFN	valid?	PFN	valid	prot	PFN	valid	prot
100	1	10	1	r-x	—	0	—
—	0	23	1	r-x	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	80	1	rw-	—	0	—
—	0	59	1	rw-	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	55	1	rw-
101	1	—	0	—	45	1	rw-

```
1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True)    // TLB Hit
4    if (CanAccess(TlbEntry.ProtectBits) == True)
5      Offset   = VirtualAddress & OFFSET_MASK
6      PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7      Register = AccessMemory(PhysAddr)
8    else
9      RaiseException(PROTECTION_FAULT)
10 else                    // TLB Miss
11   // first, get page directory entry
12   PDIndex = (VPN & PD_MASK) >> PD_SHIFT
13   PDEAddr = PDBR + (PDIndex * sizeof(PDE))
14   PDE     = AccessMemory(PDEAddr)
15   if (PDE.Valid == False)
16     RaiseException(SEGMENTATION_FAULT)
17   else
18     // PDE is valid: now fetch PTE from page table
19     PTIndex = (VPN & PT_MASK) >> PT_SHIFT
20     PTEAddr = (PDE.PFN<<SHIFT) + (PTIndex*sizeof(PTE))
21     PTE     = AccessMemory(PTEAddr)
22     if (PTE.Valid == False)
23       RaiseException(SEGMENTATION_FAULT)
24     else if (CanAccess(PTE.ProtectBits) == False)
25       RaiseException(PROTECTION_FAULT)
26     else
27       TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
28       RetryInstruction()
```

Figure 20.6: **Multi-level Page Table Control Flow**

Multi-level page tables advantages

- allocated page table space is proportional to the used address space
- easy to manage the memory.

because, in contrast to linear page tables (that appear contiguously in memory) the pages of a multi-level page tables need not be together.

⇒ more flexible to allocate

Cost of multi-level page tables?

On a TLB miss → linear page table → 1 memory reference to access PTE

→ multi-level page table → 2 memory reference needed (first to PDE → then to PTE)

Remember: smaller tables comes with a cost!

~ time-space trade off

more complex page table lookup

(but, we are willing to increase the complexity in order to improve performance and reduce overheads)

Page Directory		Page of PT (@PFN:100)			Page of PT (@PFN:101)		
PFN	valid?	PFN	valid	prot	PFN	valid	prot
100	1	10	1	r-x	—	0	—
—	0	23	1	r-x	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	80	1	rw-	—	0	—
—	0	59	1	rw-	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	55	1	rw-
101	1	—	0	—	45	1	rw-

Example of a translation. What is the physical address of 0th byte of VPN 254:

(an extension of the previous example

i.e., 11 1111 1000 0000 in binary

PDInd PTIndex offset

PDIndex = 1111 = 15 ⇒ PFN of the 15th page = 101

(accessed using PDEAddr)

Go to PFN = 101, the PTE we need is present in this frame.

PTIndex = 1110 = 14 $\Rightarrow$ 14th entry of the page of the pagetable in PFN=101 is the desired PTE

14th entry points to the physical page 55 = 0011011 1

since offset = 000000, the desired physical address
= 001101111000000

More than two levels

Suppose virtual address space = 30-bit, page size = 512 bytes = 2^9 bytes

we need 9-bit offset

Total # virtual pages = $\frac{2^{30}}{2^9} = 2^{21}$

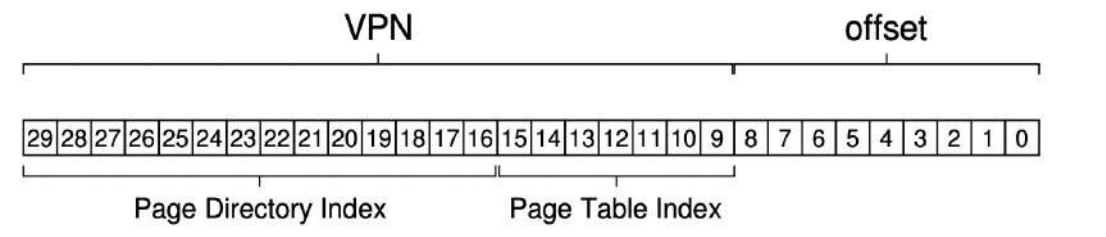
$\Rightarrow$ # page table entries = 2^{21}

page size

since entry needs 4 bytes,

$\Rightarrow$ # pages for page table = $\frac{2^{21} \times 2^2}{2^9} = 2^{14}$

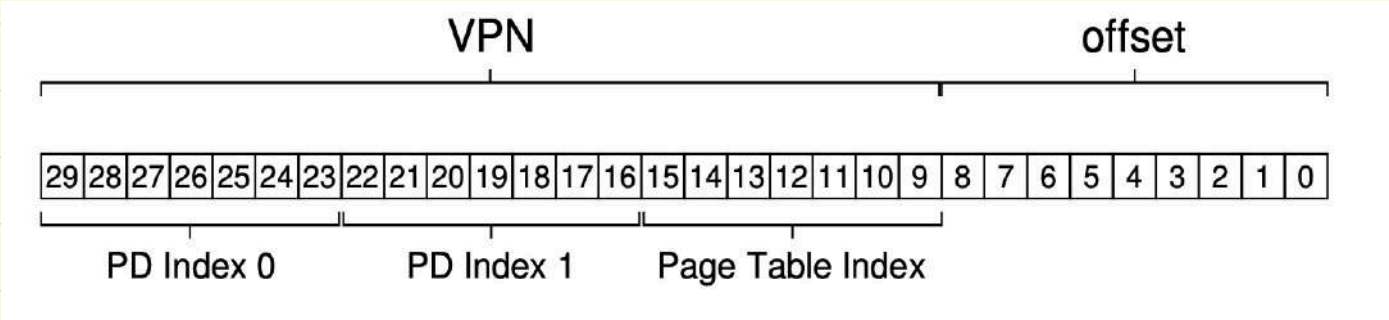
entries in page directory = 2^{14}



Suppose each PDE is of 4-bytes, then 1 page of the directory can contains only $\frac{2^9}{2^2} = 2^7$ entries

⇒ if we want to maintain only 1 directory, it spans over multiple pages that need to be placed consecutively, which is again bad! (2^7 pages)

How to overcome this? split the page directory into multiple pages, and add another directory on top to point to these pages.



Inverted Page tables

Recall that, in normal page table → one entry per virtual page and there is a page table corresponding to each process.

Inverted page table → one entry per physical frame
(IPT) (one single table for all the processes)

each entry stores
- virtual page number (VPN)
- process identifier (ASID or PID)
- other bits (protection bit, valid bit, etc..)

Problem: Look up is reversed! Given a VPN, how do we quickly find which physical frame it maps to?

↳ a linear scan of (VPN, PID/ASID) could work, but it is expensive!

How to speed up the look ups?

→ use hash tables! (eg:- PowerPC architecture use this)

Role of hashing:

* extract VPN from the address
* key = (ASID/PID, VPN)
* compute hash = h(ASID/PID, VPN)
  ↳ hash function (eg:- h(ASID, VPN) = (ASID ⊕ VPN) mod Hash-table size)
* Go to hash_table[hash]
* That entry points to the corresponding entry in the Inverted page table

If there's a collision (two keys hash to the same slot), a linked list/chain of entries is used. The OS walks the chain, checking (PID/ASID, VPN) until it finds the right one.

Trade offs:

- Hashing Pros: Fast lookup, efficient for large address space
- Cons: extra space for Hash table + chain, handling collisions, more complex than a linear scan
- but a linear search is only reasonable for very small systems (few frames)

Multi-level Page tables

- Very common today (X86, ARM, RISC-V etc.)
- Each process has its own page table
- Hardware page walker walks the levels on a TLB miss, works well with sparse address spaces.

Inverted Page tables

- Used in some systems (IBM POWER, early HP-UX, some SPARC)
- Keeps one global table
- Memory overhead is lower for very large address spaces with many processes, but look up is slower, and doesn't play nicely with hardware managed TLBs.

Which is more common?

Multi-level page tables are far more common in modern CPUs and OSs.

Why? - Hardware support: every mainstream ISA (X86, ARM, RISC-V) defines MLPT.

- performance: MLPT works seemlessly with hardware managed TLBs
  - faster look ups
- flexibility; etc...

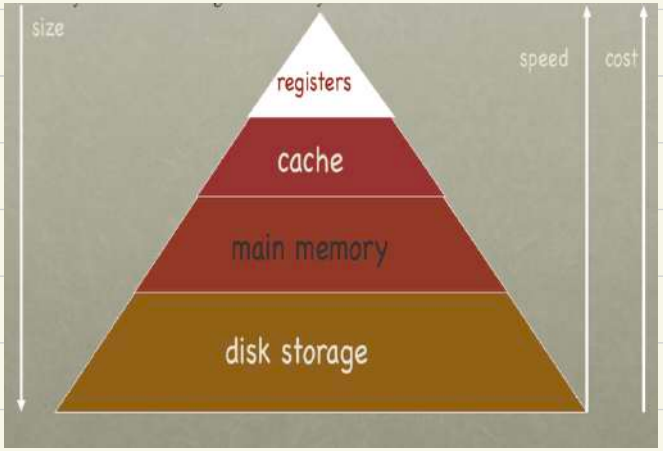
(Note that IPT makes more sense in software managed TLB architectures as it involves complicated data structures)

Beyond Physical memory: Mechanisms

an assumption so far: every address space of every running process fits into memory.

Question: How to run process when we don't have enough physical memory?

↳ OS needs a place to stash away portions of address spaces that currently aren't in great demand.



↳ this is served by hard disk drive.

Memory hierarchy

Older systems → memory overlays → programmers have to manually move pieces of code or data in and out of the memory

Modern systems → OS provides a powerful illusion that every process has all memory to itself!

Question: How to move the pages back and forth (RAM ⟷ DISK)

swap space → a reserved portion of disk that the OS uses as an extension of physical memory (RAM)

- when RAM is full, the OS can move (swap out) some pages from RAM to swap space, freeing RAM for active data.

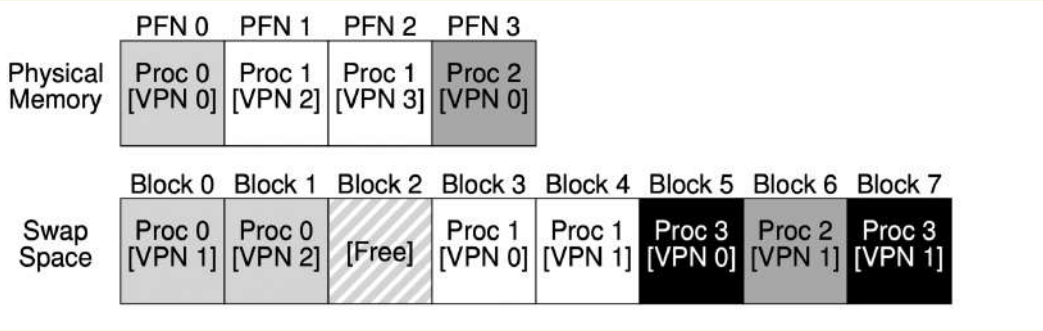
- later, if those swapped-out pages are needed again, the OS swaps them back in.

How swap space differs from the rest of the disk?

- Normal disk space (eg:- where our files and applications live) is managed by file system, which is further organized into files and directories
  ↳ will learn in detail later!
- swap space is managed directly by OS memory manager.
  It is organized into fixed-sized blocks = pages (same size as RAM pages)
  ↳ no files or folders → just raw space for pages evicted from RAM.

Note that each swap page is uniquely numbered → OS keeps a swap map (table) to track which RAM page is stored where on disk.

Example: 4-page physical memory
8-page swap space



3-processes actively sharing the memory (only some valid pages in RAM)

Proc 3 → has all pages swapped out to disk (⇒ currently not running)

Two possible disk locations for "swapping-traffic"

Swap-space
- used for pages whose data is not backed by any file (eg:- stack, heap, etc)
- they exist only in memory → if they are evicted, OS must write them to swap SO they aren't lost!

original files on disk
- used for file-backed pages (pages that came from executable, shared library, memory maped file etc)
- these already exist in disk → OS does not need to write them to swap when evicting them
- if needed, the OS just reloads from their original location on disk. eg:- when you run ls, its binary lives in /bin/ls on disk.