# Chapter 20 Paging: Smaller Tables

Problem with paging: various overheads

- Storage overhead for the page tables
- Page tables are stored in memory
- Requires extra memory lookup for each virtual address
  - Going to memory for translation information for every instruction fetch and for every load/store instruction
  - Prohibitively slow
  - In chapter 19, we introduced the concept of an *address translation cache*, or TLB, to solve the latency problem.

**Example**. 32-bit address space ($2^{32}$ bytes = 4 GB)

- 4 KB ($2^{12}$ byte) pages
- 4 byte page-table entry

One address space has one million virtual pages ($2^{32}$ / $2^{12}$) = $2^{20}$.

Multiply this with the size of the page table entry: 4 x $2^{20}$ = 4 MB.

One page table ____

With 100 active ____

Q: How can we make page tables smaller?

- Avoid linear page tables (arrays) — they are too big
- Key idea: new data structures
- Challenge: other inefficiencies with new data structures

### 20.1 Simple Solution: Bigger Pages

Example with 32-bit address space again

But now let's consider 16 KB pages or $2^{14}$ bytes.

-

-

$2^{18}$ = 256k entires

Problem with approach:

-

  -

Another alternative:

-

-

-

  -

## 20.2 Hybrid Approach: Paging and Segments

**Def. Hybrid**: *two (or more) good ideas combined to achieve the best of both worlds.*

**Example**. Linear page table

-

-

-

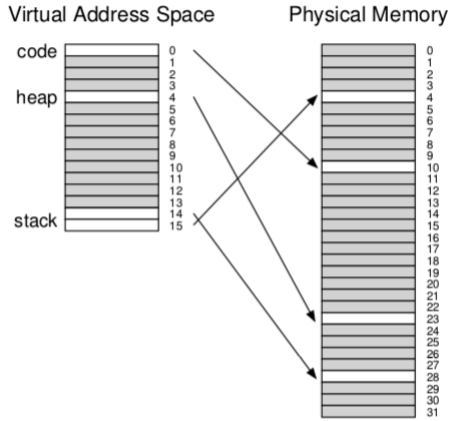



Figure 20.1: **A 16KB Address Space With 1KB Pages**

Fig. 20.2 shows that most PTEs are invalid (unused)

| 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 | — | - | - |
| 28 | 1 | rw- | 1 | 1 |
| 4 | 1 | rw- | 1 | 1 |

Figure 20.2: **A Page Table For 16KB Address Space**

Q: Why not have ____ ?

-

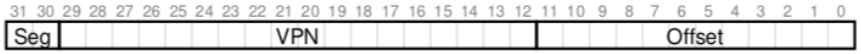
  -

Use base and bounds, but now:

- **Base** points to physical address of the page table for the segment
  - Code, stack, heap
- **Bounds** indicate end of page table
  - How many valid pages it has

**Example**. 32-bit Virtual Address Space

- 2 bit segment
- 18 bits for VPN
- 12 bits for offset (4 KB pages)
- HW: 3 base/bounds pairs of registers




On context switch:

-

On TLB miss:

- HW
  -
- HW takes
  -
  -

```
SN            = (VirtualAddress & SEG_MASK) >> SN_SHIFT
VPN           = (VirtualAddress & VPN_MASK) >> VPN_SHIFT
AddressOfPTE = Base[SN] + (VPN * sizeof(PTE))
```

Only difference from linear page tables:

-

-

**Example**. If *code* segment consumes three pages (0, 1, 2):

- Page table will have only three entires
- Bounds register will be set to 3
  - Memory accesses beyond 3 leads to exception — process termination

Benefits with hybrid approach:

-
-
  -

Challenges with hybrid approach:

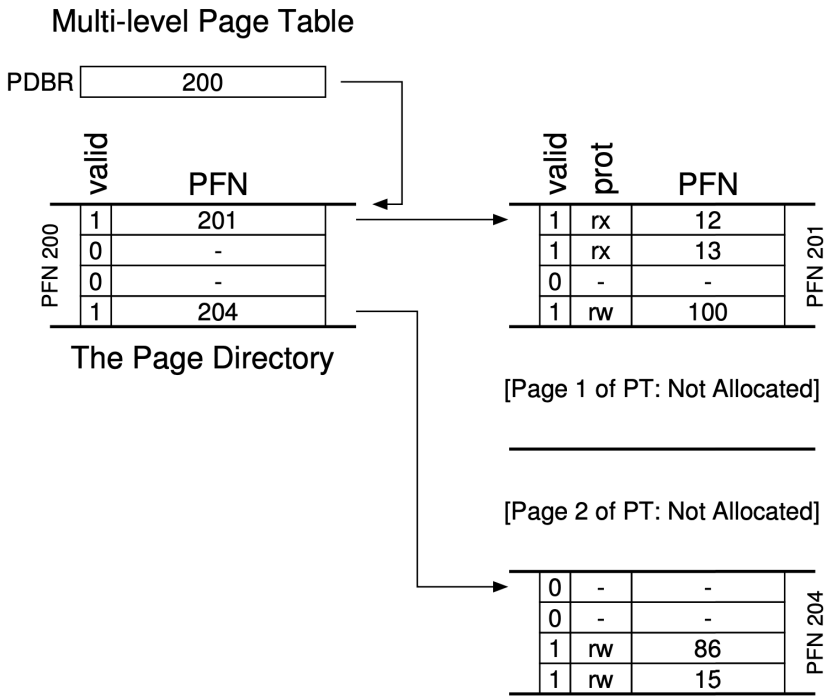
-
  -
  -
-
-

### 20.3 Multi-level Page Tables

Tree of linear page tables.

Basic idea:

- Chop Linear Page Table into page-sized units
- If an entire page of PTEs is invalid — don't allocate page at all
- **Page directory**: data structure to track
  - If a page in PT is valid and
  - Location in memory if valid

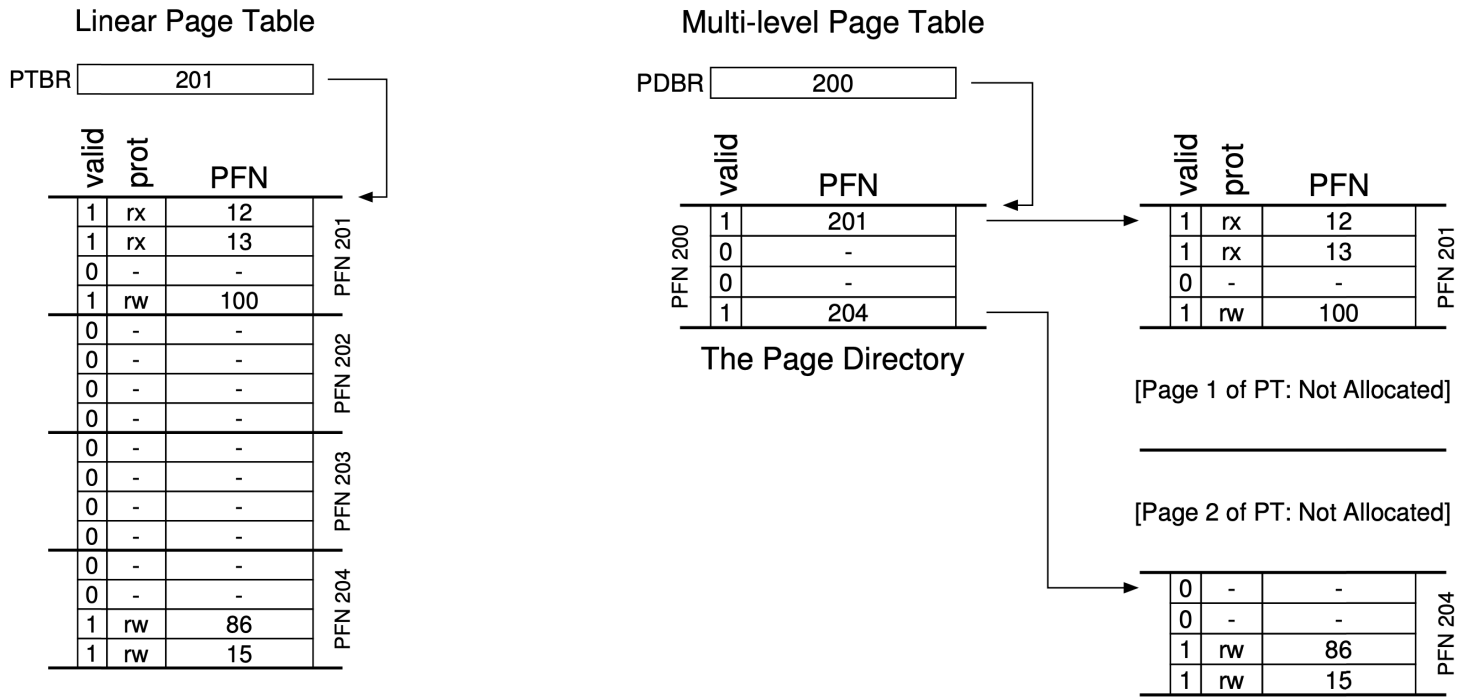


Linear Page Table:

- Two middle pages (PFN202 and PFN203) are not used
  - They have no mappings… but they still consume space in the linear PT.

Multi-level Page Table:

- Page Directory points to PFNs containing PFNs.
  - There are no pages in the middle two entries in the PD. Hence not allocated.



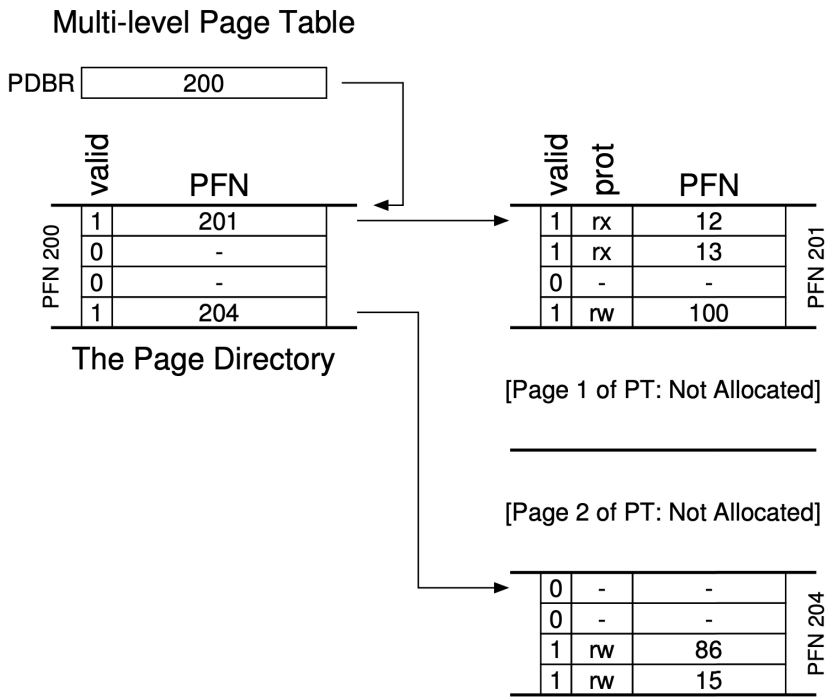
Query Page Directory to find out

- Where a page of the PT is
- Or the entire page of the PT contains no valid pages

Page Directory contains **Page Directory Entires** (PDEs) — similar to PTE.

Valid bit in the PDE:

- 1 means at least one PTE exists for page
- 0 means none of the PTEs in the PDEs range is valid



**Advantages of Multi-level Page Tables**

- Allocate only PT space for the addresses being used
  - Compact and supports sparse address spaces
- Page of PT fits on a page
  - Easier to manage memory
  - OS: Get next free page to grow the PT
  - (Not possible to grow a linear page table)
- We add a **level of indirection** by use of Page Directory.
  - Allow placing PT pages wherever in physical memory
- On TLB hit, fast, no performance penalty.

**Drawbacks with Multi-level Page Tables**

- On TLB miss:
  - Two loads from memory required to get translation
    - One for the PD and one for the PTE itself
- Linear Page table: just one load from memory
- Complexity: more involved than simple Linear PT lookup!

Time-space tradeoff: give some time to save some space

We make PT lookups more complicated to save space.

**Example.** Utilization in Linear Page Table

Small AS (16 KB), 64-byte pages

14-bit virtual AS:

- 8 bits for VPN
- 6 bits for offset
- Linear PT: $2^8$ = 256 entires
  - VP 0, 1: code
  - VP 4, 5: heap
  - VP 254, 255: stack
  - Rest unused

Utilization = (2+2+2)/256= 2,3 %

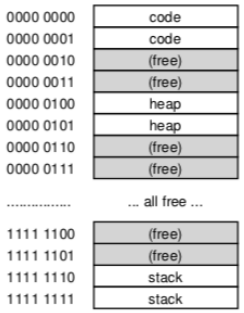


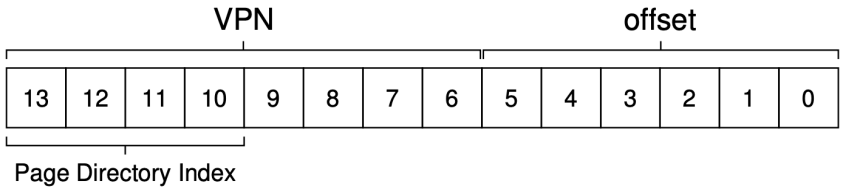

Figure 20.4: **A 16KB Address Space With 64-byte Pages**

**Example**. Build two-level Page Table

- Linear PT: 256 entires, PTE = 4 bytes
  - PT size = 256*4 bytes = 1 KB
- Break Linear PT up into page-sized units: 64 bytes
  - 1 KB Page Table divided into 16 64-byte pages
    - Each page can hold 64/4 = 16 PTEs

**Page-Directory Index** (PDIndex)

- 256 entries, spread over 16 pages
- PD: need one entry per page: 16 entires (or 4 bits of the VPN)
- PDIndex is used to compute address of the PDE
  - PDEAddr = PageDirBase + (PDIndex * sizeof(PDE))

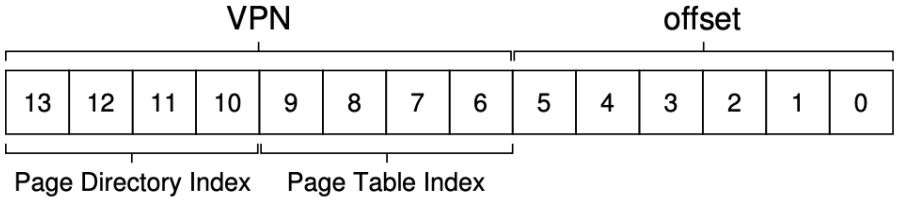



- Load PDE from the PDEAddr
  - If PDE invalid: raise exception
  - Else PDE valid

Next: *Load PTE from PT pointed to by PDE*

**Page-Table Index** (PTIndex)

- Index into PT itself
  - Gives us the address of our PTE
- PTIndex is the remaining 4 bits of the VPN
  - PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
  - Note: the PDE's PFN must be left shifted into place before combining with the PTIndex



**Example**. Multi-level Page Table in Fig. 20.5

**Page Directory** (left)

- Use only two regions of address space
  - (2 of 16 entries)
- In-between space is unused

**Page Table for PFN 100** (middle)

- Use four pages (0, 1, 4, 5) of address space
  - (4 of 16 entires)
- Code pages: 0, 1 ; Heap pages: 4, 5

| 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- |

Figure 20.5: **A Page Directory, And Pieces Of Page Table**

**Page Table for PFN 101** (right)

- Use two pages (14, 15) of address space
  - (2 of 16 entries)
- Stack pages: 254, 255

Utilization: 2+4+2 / (3*16) = 8 / 48 = 16,7 % (of the three pages)

But more important metric:
We now only use 3 pages instead of 16 pages
for storing the PT: (16-3) / 16 = 81 % improvement

*Savings for large addresses spaces are much greater!*

**Translation example**:

Virtual address 0x3F80 = 1111_1110_000000

- PDIndex = 1111 (15 th in Page Directory) —> 101
- PTIndex = 1110 (14 th entry in Page of PT (@PFN 101))
  - -> PFN = 55 = 0011_0111
  - Offset = 000000
  - SHIFT = 6 (since offset is 6 bits)

PhysAddr = (PTE.PFN << SHIFT) + Offset

= 0011_0111_000000

= 0x0DC0

| 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- |

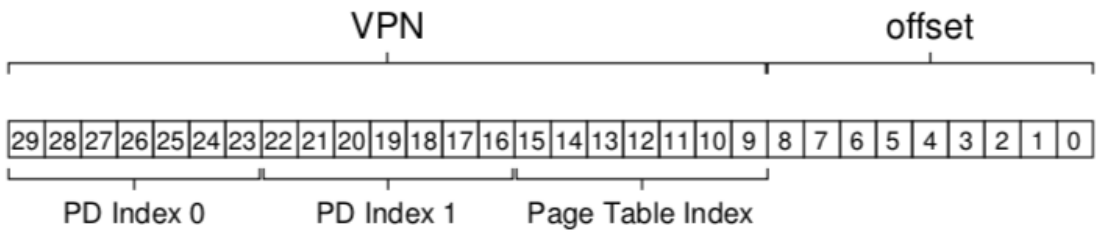
Figure 20.5: **A Page Directory, And Pieces Of Page Table**

**More than Two Levels**

For large address spaces, we need deeper multi-level page tables.

Idea (not completely accurate):

- Lookup using PD Index 0 (high order bits) to find *index* into second-level PD
- Combine this *index* with PD Index 1 to find *index2* into PT
- Combine this *index2* with PT Index and offset to find physical address



**The Translation Process (now with TLB)**

TLB hit: all good, no performance penalty for multi-level page table lookup

TLB miss: HW must do full multi-level lookup

- For *k*-level page table:
  - We get *k* additional memory accesses versus
    just one to get the actual data/instruction we are interested in.

```
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**

### 20.6 Summary

Tradeoffs:

- Time vs space: Sacrifice time to save space
- Complexity: Make PT lookups more complicated to save space for PTs