16. Advanced page Tables

page tables are too big => consume too much memory.

HOW to make Page Tables Smaller?

Simple array-based page tables (usually called linear page tables) are too big, taking up far too much memory on typical systems.

Simple Solution: Bigger Pages

=> less entry number => reduce the size of the page table

internal fragmentation

allocating pages but only using little pieces of each
=> memory quickly fills up with these overly-large pages

==> most systems use relatively small page sizes in the common case: 4KB (as in x86) or 8KB

Multiple Page Sizes

many architectures support multiple page sizes.

Usually, a small (4 or 8KB) page size is used.

However, if a "smart" application requests a single large page (ex. 4MB), It can be used for a specific portion of the address space, enabling such applications to place a frequently-used (and large) data structure in such a space while consuming only a single TLB entry.

Large page usage is common in database management systems & high-end commercial applications.

The main reason for multiple page size is

- not to save page table space
- to reduce pressure on the TLB, enabling a program to access more of its address space without suffering from too many TLB misses.

using multiple page sizes makes the OS virtual memory manager notably more complex
*Ex. number of bits for offset in virtual address should be change,
choosing what page size will app gonna use

=> large pages are sometimes most easily used by exporting a new interface to applications to request large pages directly.
ex) size of page it needs

hybrid

whenever you have 2 reasonable but different approaches to something in life, you should always examine the combination of the two to see if you can obtain the best of both worlds.

Hybrid Approach : Paging & Segments

to reduce the memory overhead of page tables

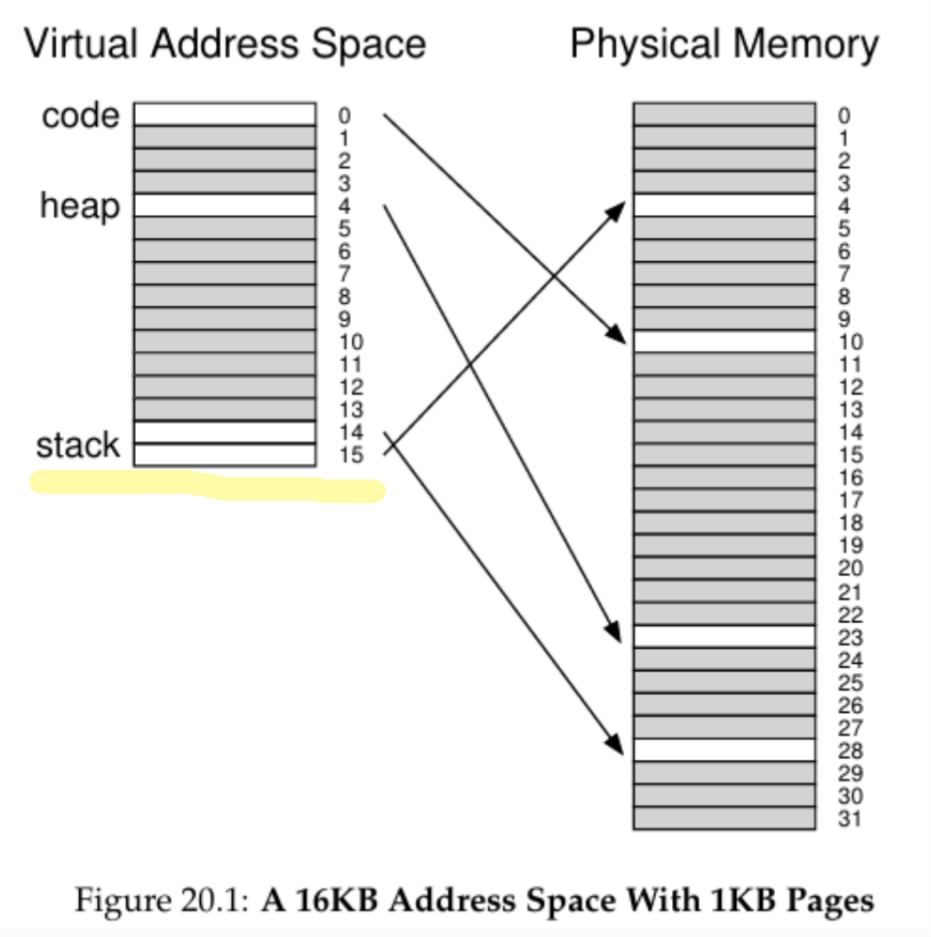
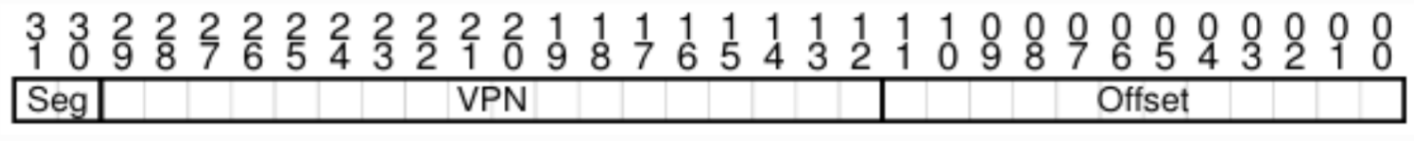


Figure 20.1: **A 16KB Address Space With 1KB Pages**

most of the page table is unused, full of invalid entries. WHAT A WASTE!

having page table /per process -> page table /per logical segment



have base & bound register (per segment) in the MMU

- base register : hold the physical address of **the linear page table** of that segment, not the physical address of that segment itself.
- bound register : indicate the end of the **page table** (how many valid pages it has)

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

- what about switching in segments : we don't need that. we got base & bound register pairs in MMU as many as the number of segments. we can just use it.

On a TLB miss (assuming a hardware-managed TLB), the hardware

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

1. uses the segment bits (SN) to determine which base & bounds pair to use.
2. takes the physical address therein
3. combines it with the VPN to form the address of the page table entry (PTE)

Unallocated pages between the stack & heap

no longer take up space in a page table just to mark them as not valid
don't have PTE for those. => page table size reduced.
we only save valid PTE in page table

Problems

1. **segmetation** : not flexible as we expect,
   as it assumes a certain usage pattern of the address space.
   - large but sparsely-used heap => can still end up with a lot of page table waste

2. **external fragmentation** : arbitrary page table size
   while most of memory is managed in page-sized units,
   page tables now can be vary (in multiples of PTEs) since we don't save invalid PTEs.
   => finding free space for them in memory is more complicated
   + More allocations : separate page tables per a segment, not process

Multi-level Page Tables

how to get rid of invalid regions in the page table instead of keeping them all in memory? : turns the linear page table into something like a tree

so effective that many modern systems employ it

Basic idea

1. chop up the page table into page-sized units
2. if an entire page of page-table entries (PTEs) is invalid, don't allocate that page of the page table at all.

Page directory

to track

- whether a page of the page table is valid
  : tell you if the entire page of the page table contains any valid pages or not
- if valid, where a page of the page table is

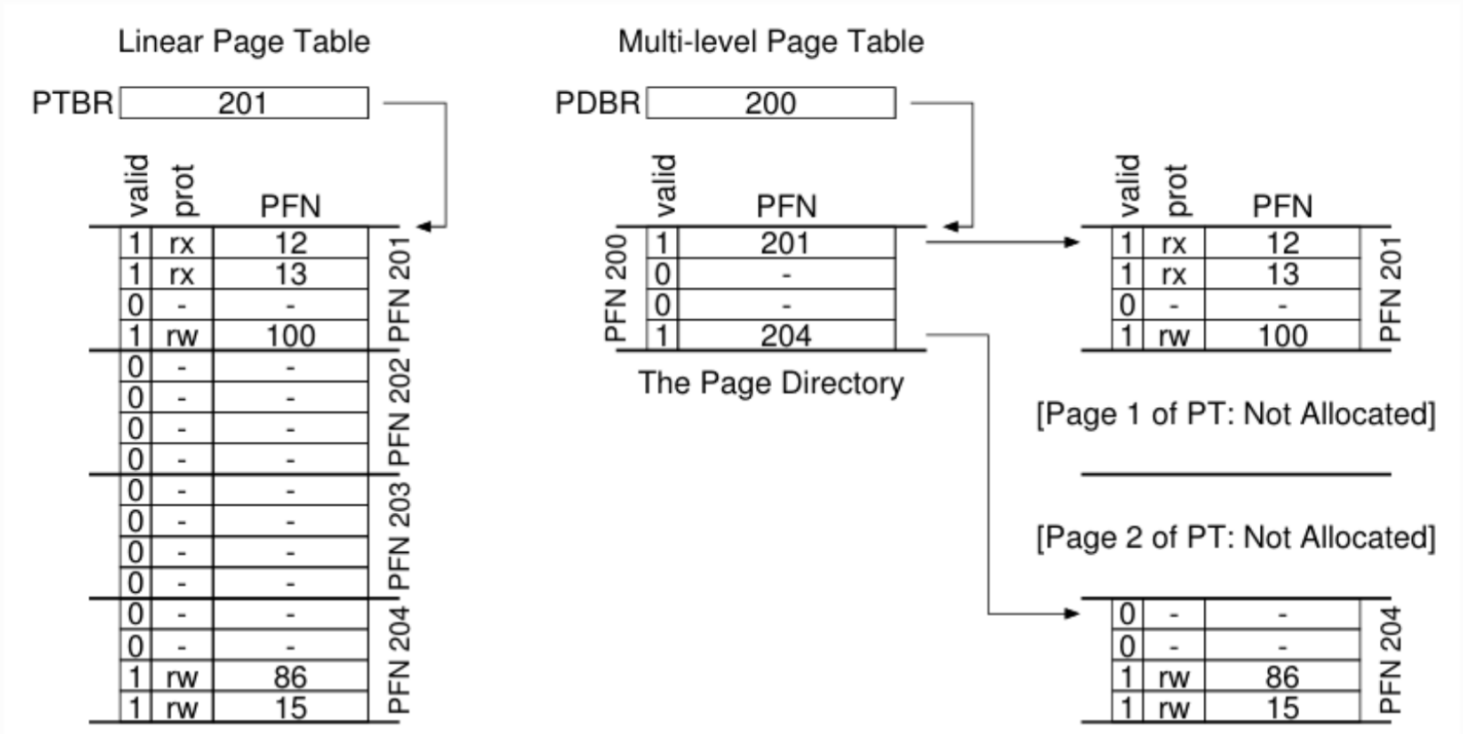


Figure 20.3: **Linear (Left) And Multi-Level (Right) Page Tables**

linear page table
even though most of the middle regions of the address space are not valid, we still require page-table space allocated for those regions (the middle 2 pages of the page table)

multi-level page table
the page directory marks just 2 pages of the page table as valid.
tracks which pages of the page table are allocated with the page directory

Page Directory
consists of page directory entries (PDE)

Page Directory Entries (PDE)
has a valid bit & a page frame number (PFN)

PDE vs. PTE is valid
PTE is valid : that page is actually allocated & mapped in physical memory
PDE is valid : that page of page table includes at least one vaild PTE

Multi-level Page tables :Advantages

1. greatly reduce page table size : memory saving
only allocates page-table space (in proportion to the amount of address space)* you are using
=> compact & supports sparse address spaces.
*it saves a whole PTE of the PDE even if there is only 1 valid PTE within that PDE. not strictly save valid PTE only.

2. Easy to manage memory : size is unified
each portion of the page table fits neatly within a page, making it easier to manage memory
the OS can simply grab the next free page when it needs to allocate or grow a page table

3. contiguous vs. seperate
- in linear page table , page table **[must be contiguous]**
  the entire linear page table must reside contiguously in physical memory
  => For a large page table, finding such a large chunk of unused contiguous free physical memory can be quite a challenge.

- in multi-level page table, page table **[can be seperated by PDE unit]**
  add a level of indirection through use of the page directory, which points to pieces of the page table.
  => allows us to place page-table pages wherever we would like in physical memory
  *however in PDE, PTEs should be contiguous

Multi-level page tables : cost

takes more time when TLB miss occur
on a TLB miss, 2 loads from memory (page directory/PTE itself) are required to get the right translation information from the page table
=> **time-space trade-off.**
although in the common case (TLB hit), performance is obviously identical, a TLB miss suffers from a higher cost with this smaller table.

Complexity
handling the multi-level page-table lookup on a TLB miss is more involved than a simple linear page-table lookup.

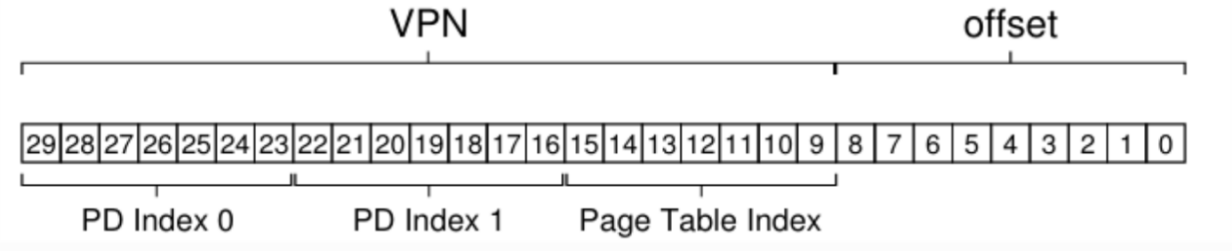
we are willing to increase complexity in order to improve performance or reduce overheads.
we make page-table lookups more complicated in order to save valuable memory.

More than Two Levels page table

goal in constructing a multi-level page table
: make each piece of the page table fit within a single page.
=> what if the page directory gets too big?

we build a further level of the tree,
by spliting the page directory itself into multiple pages,

and then adding another page directory on top of that,
to point to the pages of the page directory.



```
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**

More than Two Levels page table

goal in constructing a multi-level page table
: make each piece of the page table fit within a single page.
=> what if the page directory gets too big?

we build a further level of the tree, by spliting the page directory itself into multiple pages, and then adding another page directory on top of that, to point to the pages of the page directory.

Inverted Page Tables

even more extreme space saving

instead of having many page tables (/per process),
keep a single page table that has **an entry for each physical page** of the system (/per system, physical-logical page 1-1 mapping)

- page table size = **Number of physical frames** × Size of each PTE

entry tells us

- which process is using this page
- which virtual page of that process maps to this physical page

Finding the correct entry is now **a matter of searching** through this data structure. (cause is indexed with physical page number, not logical)
=> a linear scan would be expensive => hash table / to speed up lookups.

page tables are just data structures.
: you can freely do anything with data structures,
making them smaller / bigger, making them slower / faster.

Swapping the Page Tables to Disk

Even with our many tricks to reduce the size of page tables,
Page Tables may be too **big** to fit into kernel-owned **physical memory** all at once.

=> some systems place such page tables in kernel **virtual** memory, thereby allowing the system to swap some of these page tables to **disk** when memory pressure gets a little tight.

the bigger the table, the faster a TLB miss can be serviced.

with a linear page table, we need a single register (PTBR).

Q. should we need to save page table size?
we do not necessarily need to save page table size because we anyway have PTE for each every possible virtual address whether is valid or not.

Q. assuming that hardware does the lookup upon a TLB miss. How many registers do you need to locate a two-level page table? A three-level table?
: only one.
because either way the start address for each PDE or PTE will save in prior level page table

- is TLB per a process? : no. it's per CPU since it's in MMU units.
- should TLB save the process number? : yes. with ASID
- what TLB saves? : VPN - PFN, ASID
- should TLB reset in context switch? : No since it saves ASID together