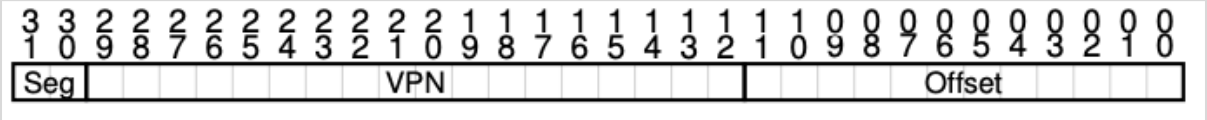
#uni/semester3/Betriebssysteme/chapter20/chapter-notes

➜ page tables are too big and we have one page for every process in the system.
➜ simple linear array based page tables are too big

1. Simple Solution: Bigger Pages
   ➜ by increasing the page size we would reduce the number of pages (less translations). The reduction mirrors the factor of increase in page size.
   ➜ this leads to a problem, namely that there is waste within each page, which is known as internal fragmentation.
   ➜ most systems use relatively small page sizes

2. Hybrid Approach: Paging and Segments
   ➜ many architectures now support multiple page sizes
   ➜ it still uses a normal page size, but a smart program request a single large page. It can use it to store a frequently used and large data structure, while consuming only a single TLB entry. The idea is to enable a program to access more of its address space, without suffering from too many TLB misses. Problem is that multiple page sizes makes the OS virtual memory management more complex.
   ➜ hybrid tries to combine segmentation with paging, in order to reduce the memory overhead of page tables.
   ➜ imagine an only paging system an address space of 16 pages, where only 4 are used (not contiguous). It is a waste of memory, and gets even bigger with 32-bit address spaces.
   ➜ hybrid approach tries to not have a single page table for an address space, but have a page table per logical segment. One for code, stack and one for heap.
   ➜ we still have the base and bounds registers in the MMU. Base is the physical address of the page table and bounds indicates how many VALID pages it has.
   ➜ the address space gets some extra bits, to decide which segment an address refers to



   ➜ in the hardware there are then 3 pairs of base/bounds. We have three segments for code, stack and heap.
   ➜ each process in the system has three page tables. These registers of base/bounds must be changed on context switch.
   ➜ assuming we have a hardware managed TLB, where the hardware handles the TLB misses. On a TLB miss the hardware uses the SN (segments) bits to determine which base

and bounds to use. Then takes the physical address of the page table and combines it with VPN to get the entry address.

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

(We don't have a page table base register anymore, but use base)

➜ the critical difference is that we have a bounds register per segment. This registers the value of the maximum valid page in the segment. If we have the code segment using its first three pages (0,1,2) the code segment page table will only have three entries and the bounds will be set to 3.

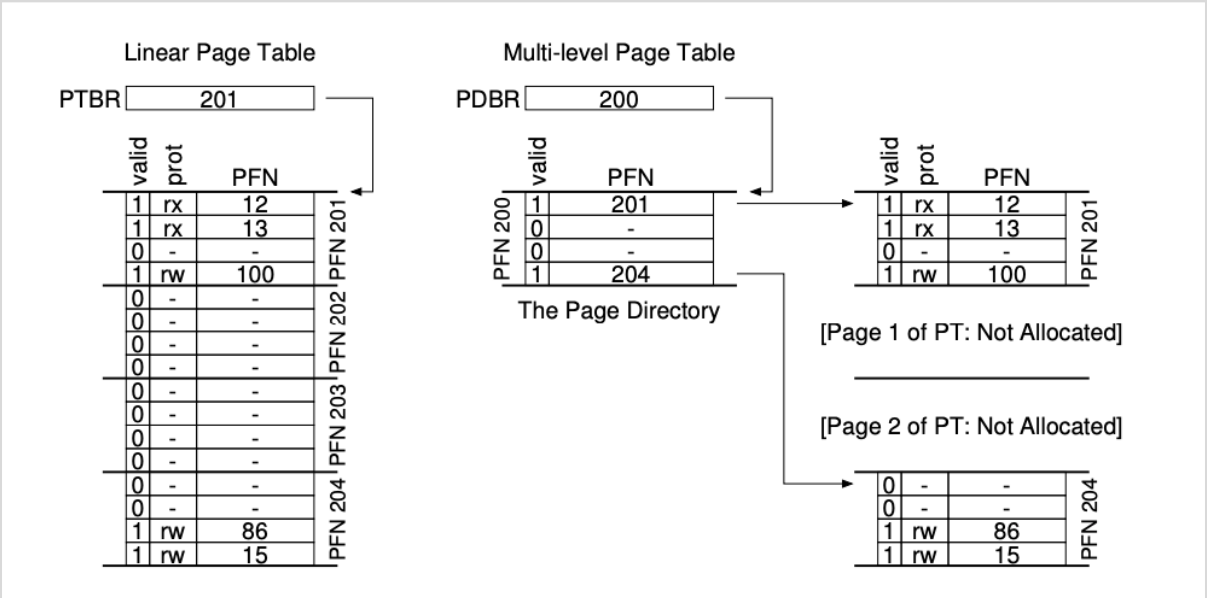
➜ Thus the hybrid approach reduces memory waste compared to the linear page table, because unallocated pages between the stack and heap won't take place in a page table anymore, just to mark them as invalid.

➜ Hybrid approach still has problems, because segmentation is not quite flexible. Large sparsely used heap also ends up with a lot of page table waste. At last it because pages can have variable sizes, it leads to external fragmentation.

3. Multi-level Page Tables

➜ multi level page table also wants to to get rid of all those invalid regions in the page table. This is done by turning the linear page table into a tree.

➜ it chops the page table into page-sized units. To track whether a page of the page table is valid, we use a page directory. The page directory tells you where a page of the page table is, or that the entire page of the page table contains no valid pages.



➜ PTBR stands for page table base register

➜ PDBR stands for page directory base register

➜ on the left is linear page table, where the middle regions are not valid but still allocated.

➜ on the right is a multi level page table. Only two pages of the page table are marked as

valid in the page directory. The two unused pages are marked as invalid, but still allocated in the page directory. On far right we see the entries of the four pages. First and last are allocated. Second and third are not allocated.

➜ Page directory has page directory entries (PDE) . A PDE has a valid bit and page frame number (PFN). If the PDE is valid, it means that at least a an PTE of the page the PDE points to, is marked as valid. If PDE is not valid, the rest of PDE is not defined

➜ Advantage is that multi level page table only allocates page table space in proportion to the used address space. Thus it supports sparse address spaces.

➜ Each portion of the page table goes in a page, thus the OS can manage memory easier. Simply grab the next free page on allocate or grow.

➜ In contrast with linear page table the PTEs don't have to lay contiguously to one another. In the page directory yes, but the PDEs can point to pages that can be placed anywhere in physical memory. Page directory adds a level of indirection

➜ Disadvantage is that on TLB miss, we need two loads from memory to get the translation. One for the page directory and one for the PTE. Thus, multi level page table is an example of time-space trade-off. On TLB hit performance is same as linear page table.

➜ Another disadvantage is complexity.

3.1 A Detailed Multi-Level Example

➜ address space of 16KB with pages of size 64 Byte

➜ VA is 14 bit, with 8 bit for VPN and 6 bit for offset.

➜ Thus a linear page table would have 2^8 (256) entries, even if only a very small portion of the address space would be in use. Each PTE has a size of 4 Byte. Thus, page table has 1KB (256 * 4)

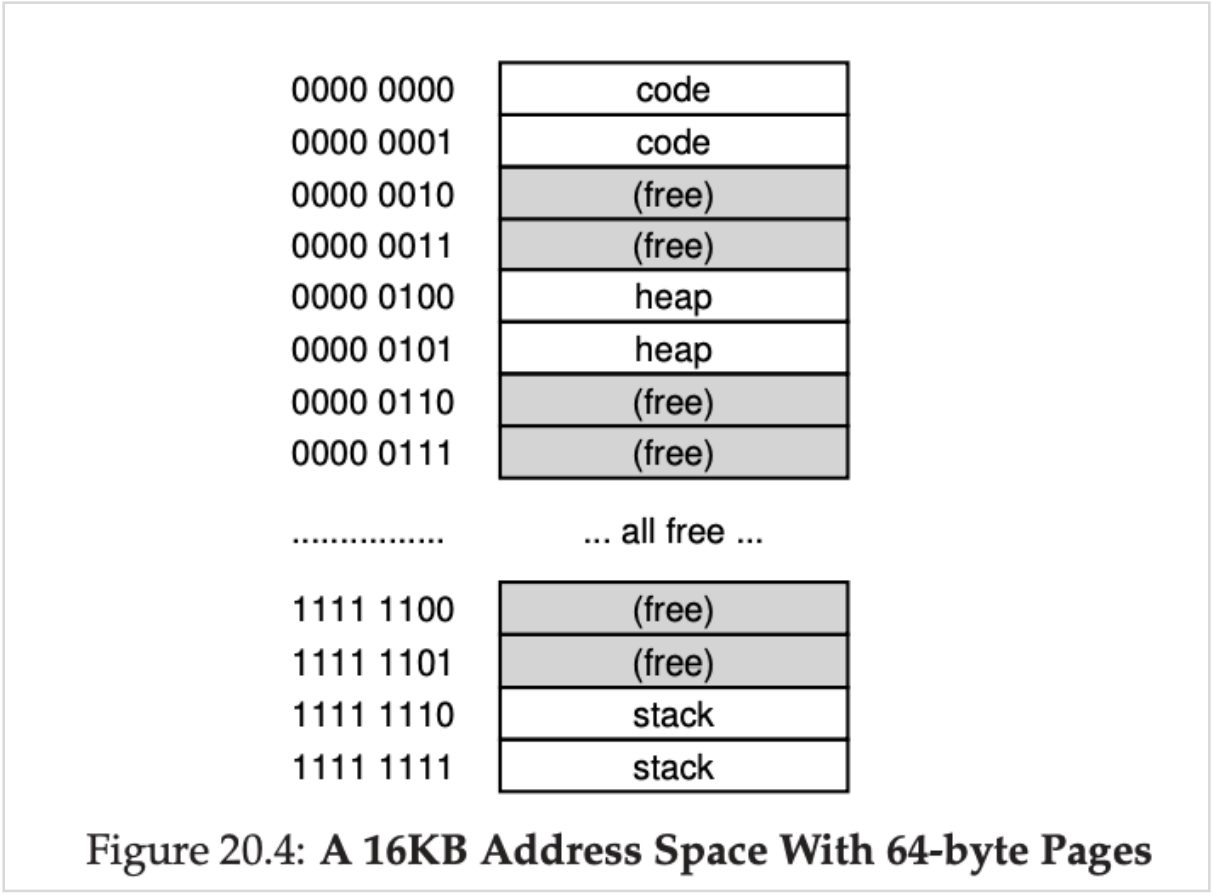


Figure 20.4: **A 16KB Address Space With 64-byte Pages**

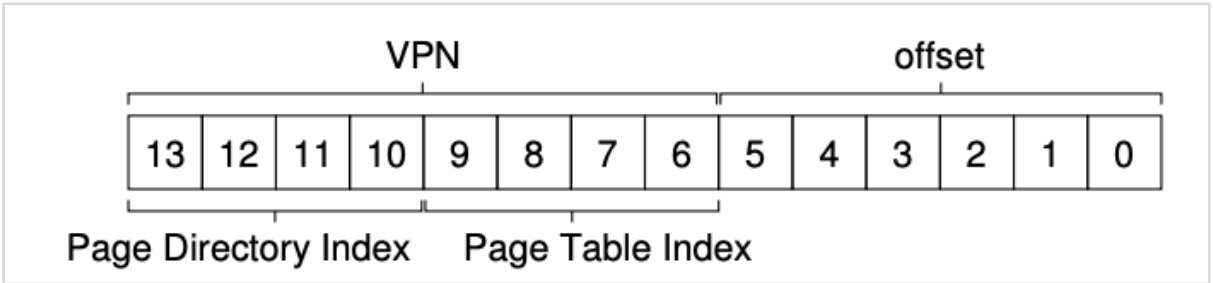
➜ 1KB can be divided into 16 pages, (page size = 64 Byte). We need in total 256 entries so 256 / 16 = 16. Thus 16 PTEs per page. (Or page size divided by PTE size)

➜ Now we need to take the VPN out of the VA, and use it to index first into page directory and then into the page of the page table.

➜ In the page directory we have 16 entries, so we need 4 bits for this. We take the to 4 bits from the VPN. We call this page directory index (PDIndex). With it we can calculate the PDE address, as follows:

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

➜ If PDE is invalid, we raise an exception. Else we then need to find the PTE in the page the PDE is pointing to.



➜ In one page of the page table we have 16 entries, so we need 4 bits for this. We take the 4 lower bits from the VPN. (Or the remaining bits, after using the bits for page directory). We call this page table index (PTIndex). With it we can calculate the address of the PTE

➜ PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))

➜ the (PDE.PFN << SHIFT) is used to calculate the base address of the page

➜ Example:

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**

➜ the PFN in page directory is actually an address of the page

➜ First entry of page directory points to middle part

➜ Last entry of page directory points to right part

➜ Last of the entries in page directory are set as invalid, which saves us memory.

➜ In this example VPNs 254 and 255 have valid mappings. These are last two entries in a linear page table, but also last here, so in PFN 101.

➜ In this example, instead of allocating the full 16 pages for a linear page table, we only allocate three. One for the page directory, and two for the chunks of the page table that have valid mappings.

➜ VA 0x3F80 or 11 1111 1000 0000 refers to the Offset 0 (lower 6 bits) of VPN 254 (higher 8 bits)

➜ With the 4 higher bits in VPN we index in page directory. So for 1111 we get the 15th entry in the page directory. The PFN in the PDE points to address 101.

➜ With the 4 lower bits in VPN we index into the page of the page table. So for 1110 we get the 14th entry on the page, and tells us that page 254 of our virtual address is mapped at physical page 55.

➜ We then concate PFN=55 (110111) with offset=000000, we get the desired physical address

➜ PhysAddr = (PTE.PFN << SHIFT) + offset

= 00 1101 1100 0000 = 0x0DC0

➜ shift should be of number of bits for offset, so 6

3.2 More Than Two Levels

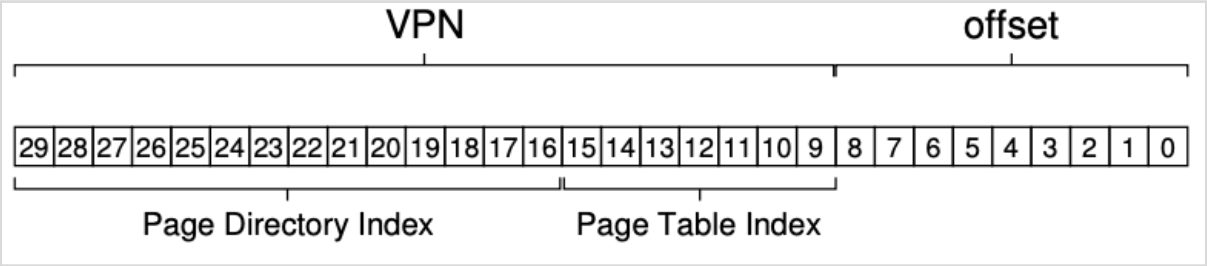
➜ last example we have one page directory and pieces of the page table. Sometimes we need more deeper levels.

➜ 30 Bit VA with page size of 512 Byte. Thus 21 bit for VPN and 9 bit for offset

➜ Goal of multi-level page table is to make each piece of the page table fit within a single page.

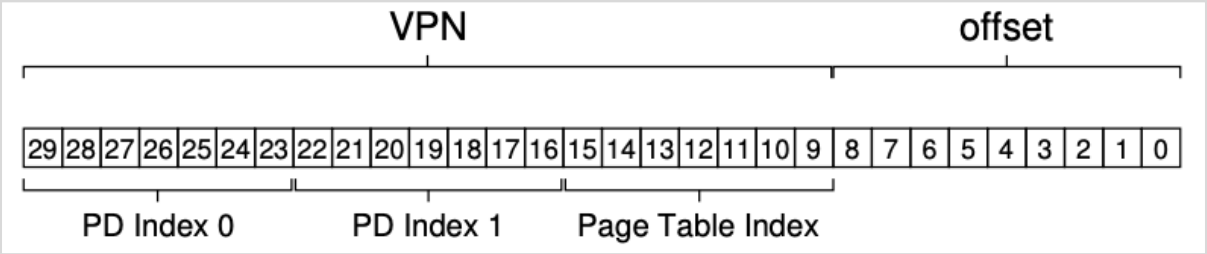
➜ What happens when the page directory gets too big?

➜ First step to determine how many levels are needed in a multi level page we need to know how many PTE fit within a page. Page size is 512 Byte and and the PTE size is 4 byte. Thus we can fit 128 PTEs on a single page. For this we will need 7 bits.



➜ As we can see the 14 bits left will be used for the page directory. The page directory will have 2^14 entries, and won't span one page but 128! Why because 2^14 / 128 PTE per page = 128

➜ To fix this we split the page directory into multiple pages and then add another page directory.



➜ We use PD Index 0 to fetch the PDE from the top level page directory. If the entry is valid we consult the second level of the page directory by combining the physical frame number from the top level PDE and the next part of the VPN (PD Index 1). Finally, if valid we combine Page table index combined with the PFN address from the second level PDE to get the PTE address.

3.3 The Translation Process: Remember the TLB

➜ Address translation for two level page table:

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)   // TLB Hit
  if (CanAccess(TlbEntry.ProtectBits) == True)
    Offset   = VirtualAddress & OFFSET_MASK
    PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
    Register = AccessMemory(PhysAddr)
  else
    RaiseException(PROTECTION_FAULT)
else                   // TLB Miss
  // first, get page directory entry
  PDIndex = (VPN & PD_MASK) >> PD_SHIFT
  PDEAddr = PDBR + (PDIndex * sizeof(PDE))
  PDE     = AccessMemory(PDEAddr)
  if (PDE.Valid == False)
    RaiseException(SEGMENTATION_FAULT)
  else
    // PDE is valid: now fetch PTE from page table
    PTIndex = (VPN & PT_MASK) >> PT_SHIFT
    PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
    PTE     = AccessMemory(PTEAddr)
    if (PTE.Valid == False)
      RaiseException(SEGMENTATION_FAULT)
    else if (CanAccess(PTE.ProtectBits) == False)
      RaiseException(PROTECTION_FAULT)
    else
      TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
      RetryInstruction()
```

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

➜ The figure shows what happens in hardware on a hardware managed TLB upon every memory reference.

➜ As normal the hardware checks first the TLB and on hit it doesn't access the page table. On TLB miss the hardware must do the multi level lookup. For the two level page table we get two additional memory accesses to look up a valid translation.

4. Inverted Page Tables

   ➜ Inverted page tables save even more space. Instead of having one page table per process we keep a single page table which has en entry for every physical page of the system. The entry tells which process is using the page and what virtual page of that processes uses this physical page.

   ➜ Finding the entry in this data structure is a matter of searching. A linear scan would be very expensive. Thus hash table is used to speed up lookups.

   ➜ Multi level and inverted page tables are just two examples how you can change the data structure to make them bigger, smaller, slower or faster.

   Multilevel Paging and Inverted Page Table - YouTube Watch it till the end, even if it is annoying

➜ multi level page table, says where to look, and the content is the PFN

➜ inverted page table, says what content to look for (PID and page number) and the index (where the content is) is our PFN

5. Swapping the Page Tables to Disk

➜ Till now assumption was that page tables are in kernel owned physical memory. Even now if our tricks, the memory still could not be enough. Thus, page tables will be placed in kernel virtual memory, allowing the system to swap some of these page tables to the disk.

6. Summary

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

Problem with 2 levels?

- Problem: page directories may not fit in a page
- Solution:
  - Split page directories into pieces
  - Use another page dir to refer to the page dir pieces

  - Exercise: How large is virtual address space with
    - 4 KB pages, 4 byte PTEs, each page table fits in page
    - given 1,2,3 levels

➜ Page size: 4KB

➜ Offset = 12 Bits (2^12 = 4096)

➜ PTE size: 4 Byte

➜ 4096 / 4 = 1024 entries

➜ Each page table, fits in one page. So a page table has 1024 entries!

➜ 1 level, means it has a linear page table?

  ➜ VPN should then have 10 bits (2^10 = 1024)

  ➜ virtual address space is therefore 22 bit

  ➜ 2^22 ist 4MB

➜ 2 level, means one page directory and many page tables

- ➜ Page directory fits on one page, so it should point to 1024 pages of page tables
- ➜ Virtual address space is (10+10+12) = 32 bit
- ➜ 2^32 ist 4GB
- ➜ 1K**1K**4K

➜ 3 level, same as 2 level, but one page directory more

- ➜ Virtual address space is 42 bit
- ➜ 2^42 ist 4TB