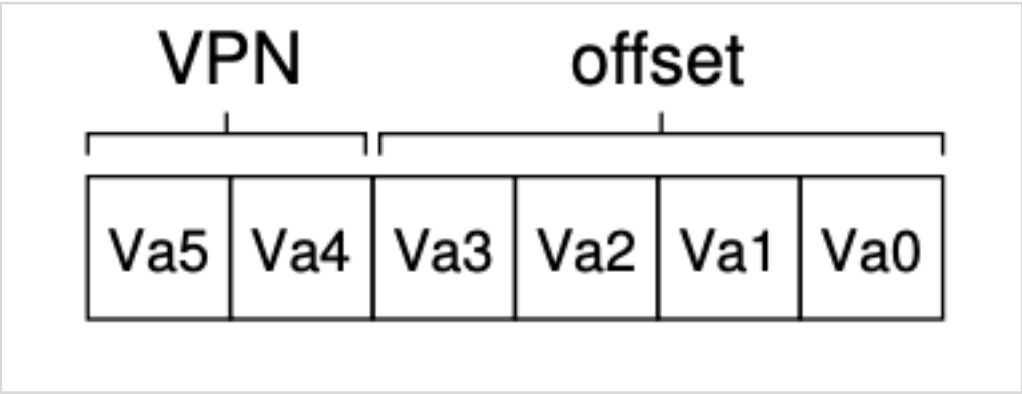
#uni/semester3/Betriebssysteme/chapter18/chapter-notes

➜ segmentation in virtual memory chops things in variable-sized pieces. This solution has difficulties, namely the space itself can become fragmented. Thus allocation becomes more challenging over time
➜ The second approach is to chop space into fixed-size pieces. This is called paging in virtual memory. The address space of a process gets splitted into fixed-sized units, which are called a page. Physical memory is viewed as an array of fixed-sized slots called page frames. Each of these frame can contain a single virtual-memory page.

1. A Simple Example And Overview
   ➜ Paging has many advantages. One particular is the flexibility. The system is able to support abstraction of an address space effectively, meaning we don't need to know how a process uses the address space.
   ➜ Another one is its simplicity. For example when we want to place a 64 Byte address space into physical memory, it simply finds four pages. Maybe the OS keeps these in a free list and just grabs the first four free pages.
   ➜ In order to record where each virtual page of address space is mapped in physical memory, the operating system keeps a per-process page table. Its role is to store address translations for each virtual page. It could look like this: Virtual Page 0 ➜ Physical Frame 3, VP1 ➜ PF 7, VP2 ➜ PF 5, etc...
   ➜ This page table is per-process, which means for another process the OS would have to manage a different page table for it. Most table structures are per-process, an exception is the inverted page table.
   ➜ To translate a virtual address, it must be first split into two components, the virtual page number (VPN) and the offset. For a 64 Byte address space we need 6 bits and out of these 6 bits we need for a page size of 16 byte, 4 bits for the offset.



   ➜ In the page table the physical frame number (PFN) or physical page number PPN is recorded. Thus, we can translate the virtual address by replacing VPN with PFN. The offset

remains the same. Example for virtual address 21

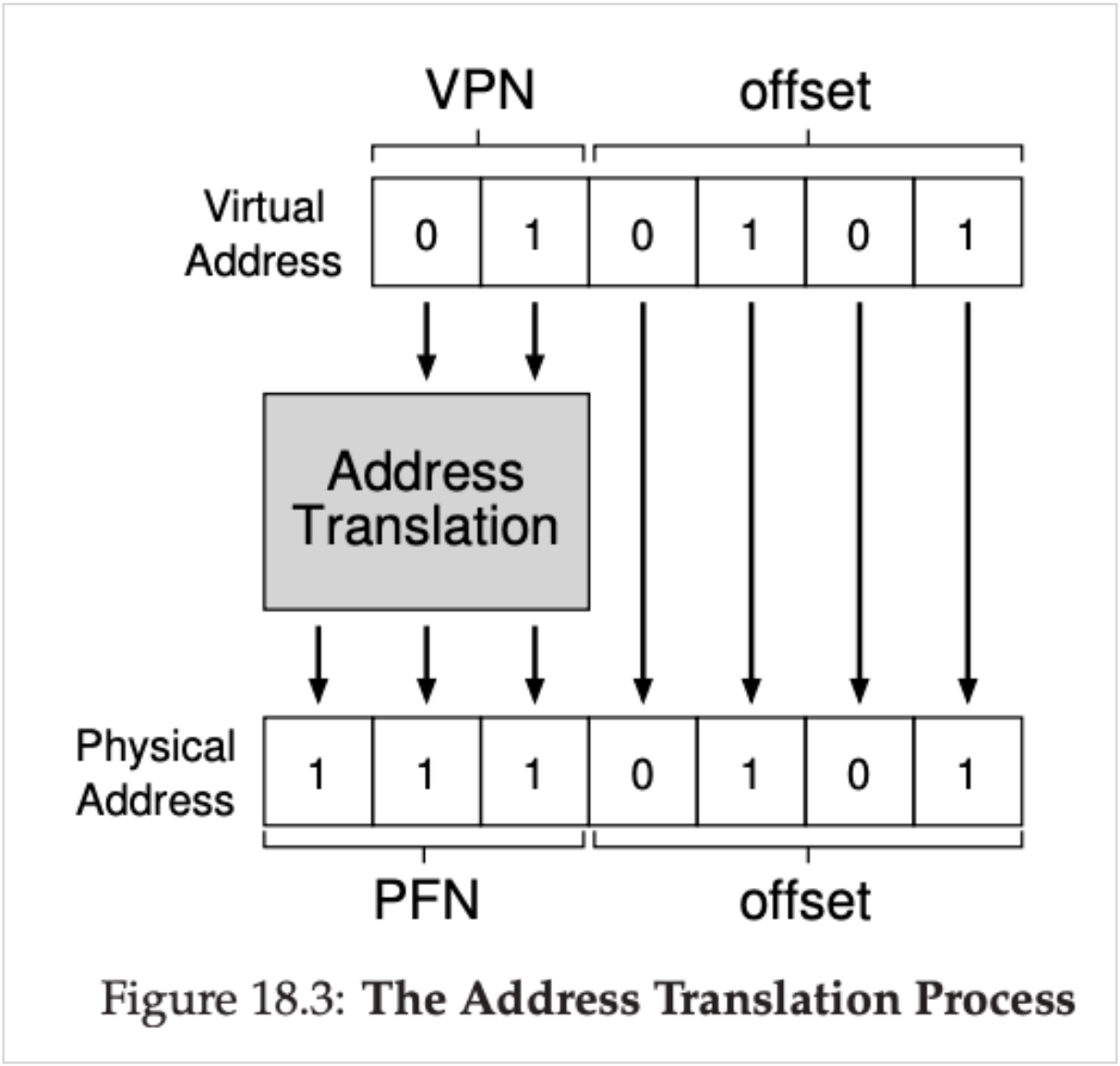


Figure 18.3: **The Address Translation Process**

2. Where Are Page Tables Stored?

   ➜ In comparison with small segment table or base/bounds pairs, the page tables can get terribly large. An 32-bit address space with pages of size 4kb, would need 2^20 translations (4kb = 2^12 ➜ 2^20 <=> 32 - 12 = 20). Assumption we need 4 bytes per page table entry (PTE) we need 4MB of memory for each table.

   ➜ Because a page table of the current running process is too big, it won't be stored in the MMU, instead each page table will be stored somewhere in memory. For now we can say, it is saved in the physical memory the OS manages. Later we will see that page tables can be saved in OS virtual memory and even swapped to the disk.

3. What's Actually In The Page Table?

   ➜ page table is a data structure used to map virtual page numbers to physical frame

numbers. The simplest form is called a linear page table, which is just an array. The OS indexes the array by the virtual page number (VPN) and the page table entry (PTE) at that index.

➜ PTE contains:

➜ a valid bit, which tells if a translation is valid. For example at the start we only have the heap and stack. The memory in between is unused and will be marked invalid. If the process tries to access such memory, it traps to the OS and gets terminated. By marking unused pages as invalid, later there won't be no need to allocate physical frames for those pages. By doing so, we save a lot of memory.

➜ protection bits (R/W), indicates of the page can be read, written to or executed from. It traps to OS if no permission

➜ present bit (P), indicates if the page is in physical memory or has been swapped out on disk. Swapping supports address spaces that are larger than physical memory.

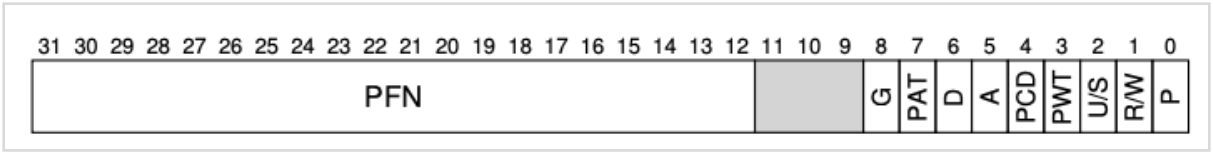
➜ dirty bit (D), indicates if page has been modified since it was brought in memory.

➜ accessed bit (A) (also reference bit) tracks if a page has been accessed, and is useful to determine which pages are popular and should be in physical memory. Such knowledge is critical during page replacement.

➜ a user / supervisor bit (U/S), which determines if user-mode processes can access the page

➜ (PWT, PCD, PAT, and G) that determine how hardware caching works for these pages

➜ and finally the page frame number PFN itself



➜ Valid bit is together with present bit. If it is set then it means page is valid and present. If not, it means that page may not be in memory (but is valid) or it may not be valid. An access to a page with P = 0 traps in the OS, which needs to determine whether the page is valid (and thus swapped back) or not (program is making illegal access).

4. Paging: Also Too Slow

➜ In order to fetch data from the virtual address, the system (hardware) must first fetch the proper page table entry from the process's page table, perform the translation, and then load the data from physical memory. To do so it needs to know where in memory the page table is, which for now will be stored in a page-table base register. To find the PTE the hardware will do this:

```
VPN      = (VirtualAddress & VPN_MASK) >> SHIFT
PTEAddr = PageTableBaseRegister + (VPN * sizeof(PTE))
```

➜ shift is always number of offset bits.

➜ Then the hardware can fetch the PTE from memory, extract the PFN and combine it with offset from virtual address.

```
offset   = VirtualAddress & OFFSET_MASK
PhysAddr = (PFN << SHIFT) | offset
```

➜ Hardware fetches the data from the address and it has succeeded in loading a value from memory. The protocol below:

```
1  // Extract the VPN from the virtual address
2  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
3
4  // Form the address of the page-table entry (PTE)
5  PTEAddr = PTBR + (VPN * sizeof(PTE))
6
7  // Fetch the PTE
8  PTE = AccessMemory(PTEAddr)
9
10 // Check if process can access the page
11 if (PTE.Valid == False)
12     RaiseException(SEGMENTATION_FAULT)
13 else if (CanAccess(PTE.ProtectBits) == False)
14     RaiseException(PROTECTION_FAULT)
15 else
16     // Access is OK: form physical address and fetch it
17     offset   = VirtualAddress & OFFSET_MASK
18     PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
19     Register = AccessMemory(PhysAddr)
```

➜ Paging requires us to perform one extra memory reference in order to first fetch the translation from the page table. Extra memory references are expensive and slow down the process by a factor of two or more. Paging causes the system to run slowly and use a lot of memory. These must be solved first.

5. A Memory Trace

   ➜ too complex to sumarize, look it up!

   http://pages.cs.wisc.edu/~remzi/OSTEP/vm-paging.pdf

```
int array[1000];
...
for (i = 0; i < 1000; i++)
    array[i] = 0;
```

```
1024 movl $0x0,(%edi,%eax,4)
1028 incl %eax
1032 cmpl $0x03e8,%eax
1036 jne  0x1024
```

Assumptions:

➜ we have a virtual address space of size 64KB

➜ a page size of 1KB => 64 pages

➜ linear (array-based) page table, which is located at physical address 1KB

➜ the first virtual page is where code lives on

➜ virtual address 1024 is on the second page of the virtual address space (VPN = 1). This maps to physical frame 4 (VPN ➜ PFN 4)

➜ The array itself, has the size of 4000 bytes (1000 integers * 4 Byte). This resides at virtual addresses 40000 through 44000 (not including the last byte). The virtual pages of this are VPN=39 ... VPN=42. These map to: (VPN 39 → PFN 7), (VPN 40 → PFN 8), (VPN 41 → PFN 9), (VPN 42 → PFN 10)

➜ Now we are ready to trace the memory references of the program

➜ When it runs, each instruction fetch will create two memory references.

➜ One to the page table to find the physical frame that instruction resides within. This is the PageTable[1] below

➜ And another one to the instruction itself to fetch it to the CPU for processing. This is lowest graph, the black points

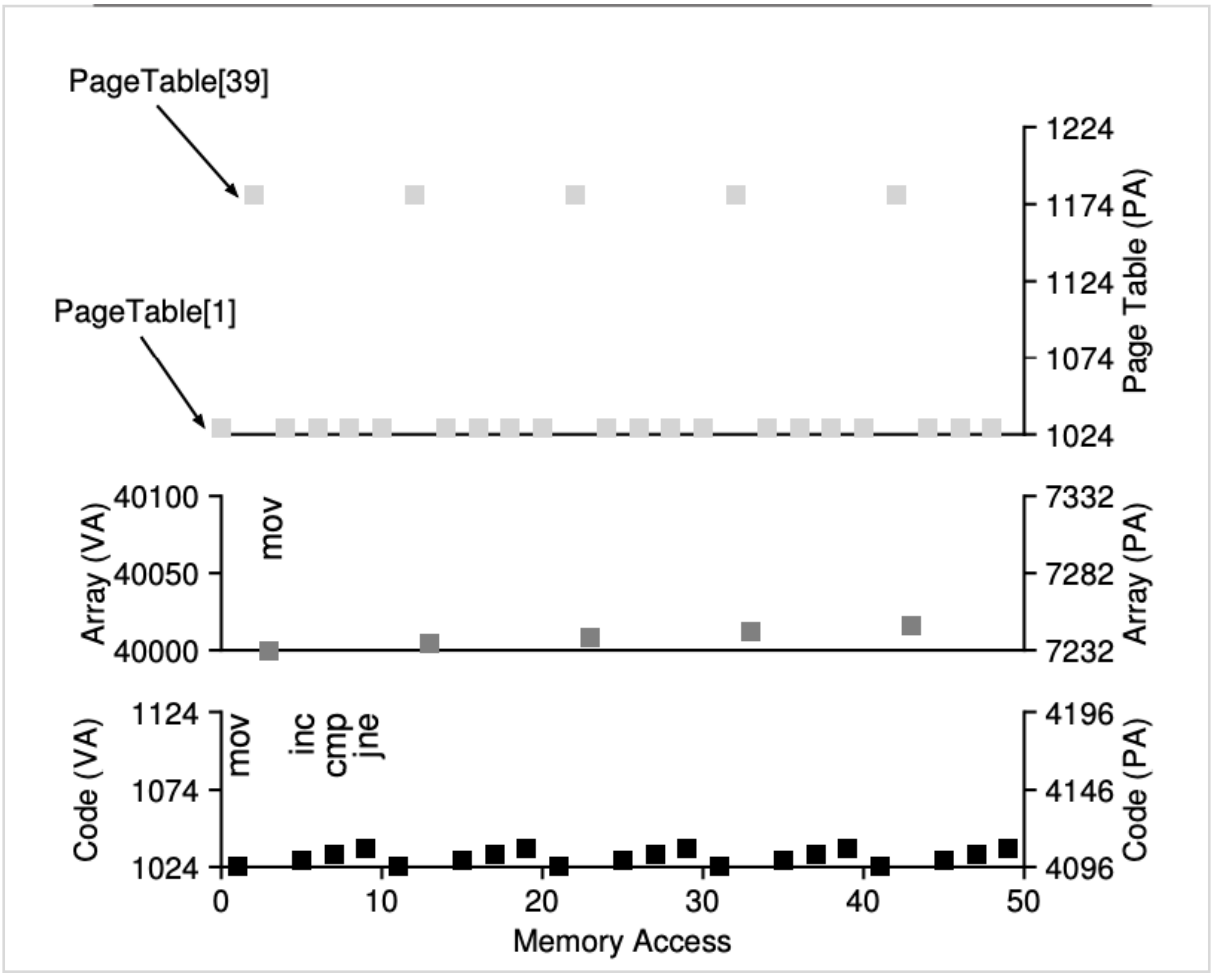
➜ instruction mov, has two extra memory references

➜ this adds another page table access first.(to translate the array virtual address to the correct physical one). This is PageTable[39].

➜ and then the array access itself. This is the middle graph, Gray points

➜ In total there are 10 memory accesses per loop. (4 Instructions + 1 Array + 4 PageTable[1] + 1 PageTable[39]) (which includes four instruction fetches, one explicit update of memory, and five page table accesses to translate those four fetches and one

explicit update.)



➜ read the whole Page 11, for clarification.