Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Operating Systems

Part 1: Virtualization – 4) Memory 2

Revision: `master@323240b (20230907-115823)`

BTI1341 / Fall 2023/24

P. Mainini / E. Benoist / C. Fuhrer / L. Ith

Outline

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Introducing Paging

Issues with Segmentation

While segmentation significantly reduces internal fragmentation, *external fragmentation* becomes a key issue.

One solution would be regular compaction by reordering used segments in physical memory; however, permanently moving memory is inefficient and not applicable in practice.

In addition, *large and sparsely used segments* (e.g. the heap) cause internal fragmentation which also should be reduced.

We will now introduce paging which provides a common solution for both issues.

Introducing Paging

With segmentation, the address space of a process was split up in segments of *variable* size, which is the main reason for external fragmentation.

Opposed to this, the main idea behind paging is to use some *fixed-size "pieces"* instead, and to map the virtual address space on those.

We call these pieces pages , they divide the physical memory into an array of fixed-size slots, which we call page frames .

Paging Example (Virtual)

In this simple example, we assume the *address space* of a process to be 64 bytes, consisting of 4 pages with 16 bytes each:[1]

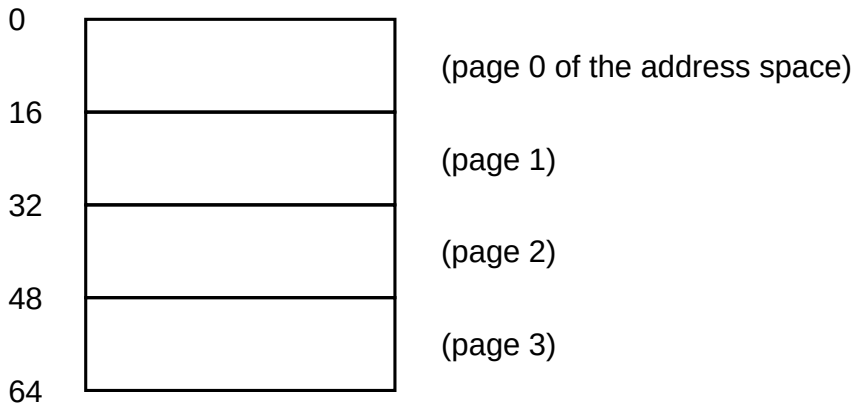

Figure: A 64 Byte Address Space

Courtesy of [ADAD18]

[1] On today's 64-bit Linux, the address space of a process is 2^{64} bytes, which is hard to imagine! The page size can be determined using "`getconf PAGESIZE`", and is typically 4096 bytes.

Paging Example (Physical)

The pages of the address space are placed in a physical memory of 128 bytes (i.e. 8 page frames of 16 bytes) as follows:

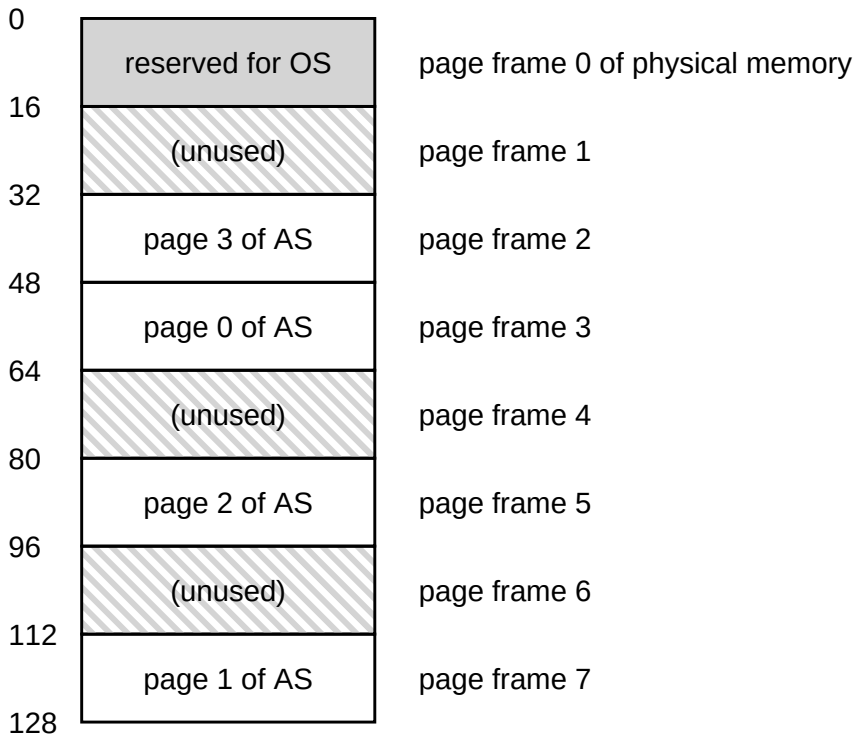

Figure: Placement of the Pages in Memory

Courtesy of [ADAD18]

The Page Table

Paging requires more complicated infrastructure (with HW support) to manage page frame mappings: the page table . It stores the address translations for every virtual page of the address space.

For our example, it looks as follows:

Virtual Page	**Page Frame**
0	3
1	7
2	5
3	2

As for segmentation, this information must be managed *per process*, i.e. every process has its own page table.

Address Translation

As for segmentation, some address translation is required to take place. The basic idea is similar: Split a *virtual* address in a virtual page number (VPN) and some offset within that page.

Then, translate the VPN into the physical frame number (PFN) :

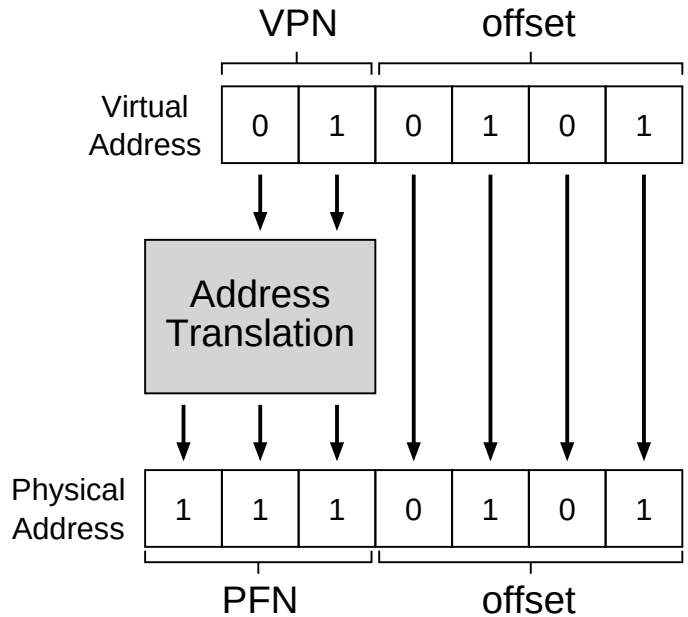

Figure: The Address Translation Process

Courtesy of [ADAD18]

Address Translation Example

Returning to our example: The given address space of 64 bytes can be represented using virtual addresses of 6 bits. Due to the page size of 16 bytes, it contains 4 pages in total, which require two bits for addressing. The remaining bits are used for the offset (refer to previous slide for an illustration).

Example:
Assume a memory access to *virtual* address 21 (binary: `010101`):

- VPN is `01`, or simply page 1 (starting at *physical* address 112)
- Offset is `0101`, the 6th byte within that page

Knowing that page 1 is mapped to PFN 7 (binary: `111`), the virtual address can be translated to the physical address `1110101` (decimal: 117). ⚠ *Note that offset remains the same.*

Next Steps

As we now have a basic understanding of paging, some important questions arise:

- How does a page table look like? What does it contain?
- Where is it stored?
- How big is it?
- Which impact has paging on performance?

We will answer those (and others) in the following sections.

More About Page Tables

Page Table Storage

As pages in general are quite small, page tables tend to grow big.[2]

For example, consider pages of 4096 bytes (2^{12}) in an "old-fashioned" 32 bit address space: The OS must then somehow manage 2^{20} ($\approx 10^6$) address translations. Assuming 4 bytes per page table entry (PTE) and 100 processes, this would require some 400 MiB of memory!

From this, we can conclude:

1. Page tables cannot fit into any registers of an MMU: for now, we'll assume that they are stored in physical memory.
2. Some clever organization is required to keep memory overhead of page table management acceptable.
3. We will again run into serious performance issues due to storing the tables in memory…

[2] Modern CPUs also support *large pages*; more on this later.

The Page Table Entry (PTE)

For now, we assume a linear page table : a simple array of page table entries (PTE) : [$\texttt{PTE}_0$, $\texttt{PTE}_1$, …, $\texttt{PTE}_n$]

A page table base register (PTBR) points to the memory location of the table.

A PTE typically has some fields corresponding to the following:

Field	**Description**
Valid Bit	Indicates if a given page is *mapped* at all. Supports creation of *sparse* address spaces.
Protection Bits	Similar to segmentation: read, write, execute…
Present Bit	Used for swapping (explained in *05-memory-3*).
Reference Bit	Tracks if a page has been accessed.
Dirty Bit	Indicates if a page has been modified.

The x86 PTE

The following is the format of the PTE used on the x86 platform (c.f. [int, Vol. 3A, 4-17]):

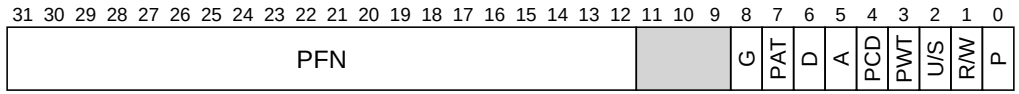

Figure: x86 4 KiB Page Table Entry

Courtesy of [ADAD18]

Field	Description
P	Present bit
R/W	Read/write: when 0, the page is read-only
U/S	User/supervisor: when 0, user-mode accesses are not allowed
PWT, PCD, PAT, G	Used for hardware caching
A	Accessed bit (reference bit)
D	Dirty bit
PFN	Physical Frame Number

Basic HW Paging Algorithm

The following pseudocode shows a possible MMU addressing algorithm for the values from the example when using paging:

```
PTBR = 0x...          // Page Table Base Register
VPN_MASK = 0x30       // 110000
OFFSET_MASK = 0xF     // 001111
OFFSET_SHIFT = 4

VPN = (VirtualAddress & VPN_MASK) >> OFFSET_SHIFT
PTEAddr = PTBR + (VPN * sizeof(PTE))
PTE = AccessMemory(PTEAddr)

if (PTE.Valid == False)
  RaiseException(SEGMENTATION_FAULT)
else if (CanAccess(PTE.ProtectBits) == False)
  RaiseException(PROTECTION_FAULT)
else
  Offset = VirtualAddress & OFFSET_MASK
  PhysAddr = (PTE.PFN << OFFSET_SHIFT) | Offset
  Register = AccessMemory(PhysAddr)
```

Tracing Memory Accesses

To inspect the effect of paging on memory accesses, we will now trace through a short sequence of code.

Assume a virtual address space of size 64 KiB and a page size of 1 KiB. The page table is linear and starts at physical address 1024. It contains the following pages:

VPN	PFN
0	3
1	4
39	7
40	8
41	9
42	10

Tracing Memory Accesses (cont.)

The figure on the next slide traces accesses in virtual and physical memory for the following code snippet:

```
int array[1000]:                    // address in EDI = 40000
for (int i = 0; i < 1000; i++)      // i is in EAX
  array[i] = 0;
```

Assuming the following assembly code:

```
1024: movl $0x0, (%edi,%eax,4)    ; Put 0 at address
                                  ; EDI+EAX*4
1028: incl %eax
1032: cmpl $0x03e8, %eax          ; Compare to 1000
1036: jne  0x1024
```

⚠ *Note that there are 10 memory accesses per loop iteration, 5 for instructions and data and 5 for the page table!*

Tracing Memory Accesses (cont.)

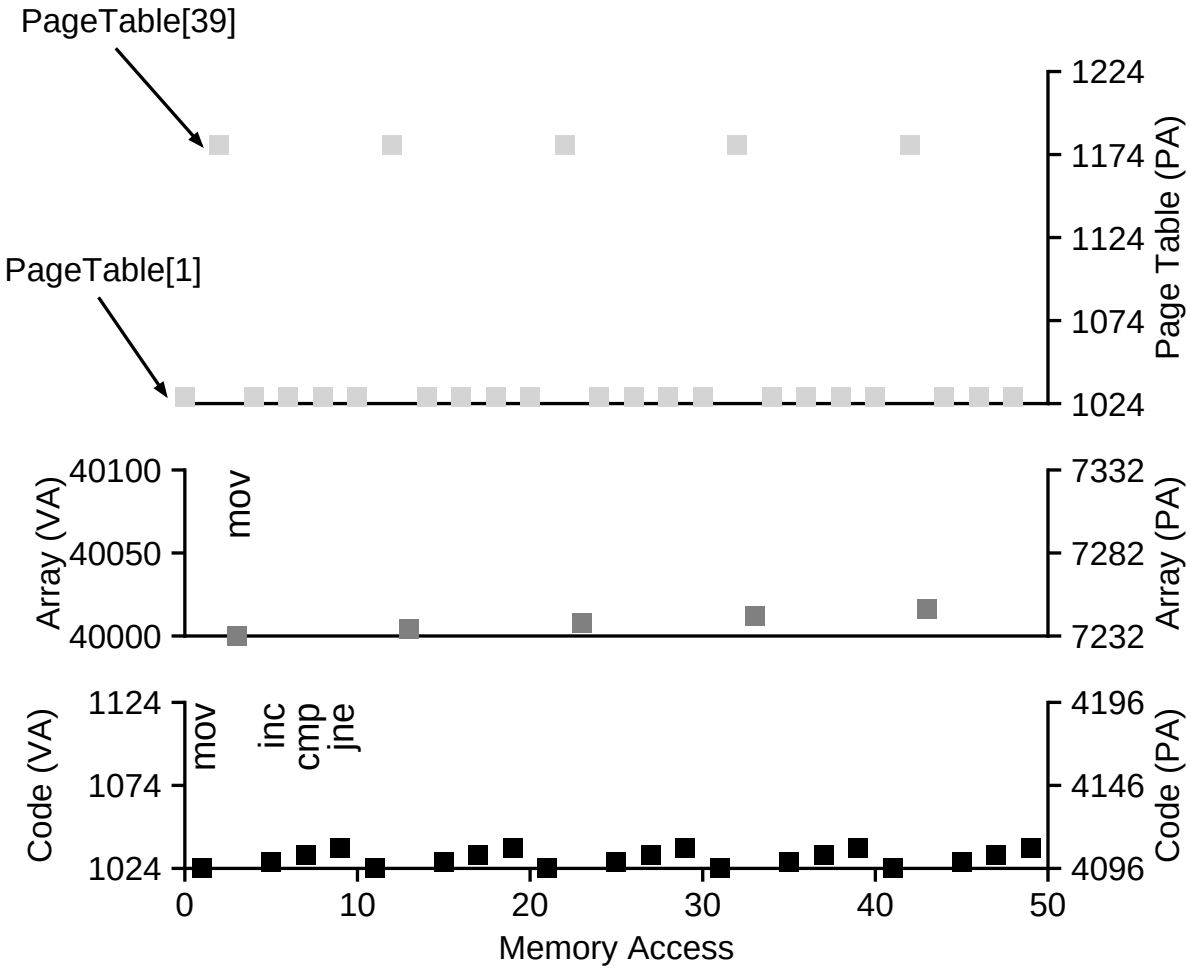

Figure: A Trace Over 5 Loop Iterations

Courtesy of [ADAD18]

Translation-Lookaside Buffers (TLBs)

Fixing Paging Performance

We have seen that paging requires a lot of mapping information (due to small pages) and generates an *additional* memory lookup on *every* memory access. For efficiency, this is not acceptable.

Hardware is again the solution: the translation-lookaside buffer (TLB) , a cache of mappings in the MMU.

For every memory access, the MMU first checks if the required address is in the TLB. If so, this is a TLB hit and address translation occurs without any page table access. Otherwise, it is a TLB miss and the PTE must be retrieved from the page table.[3]

[3] After a TLB miss, the PTE retrieved from memory is stored in the TLB.

TLB Hardware Overview

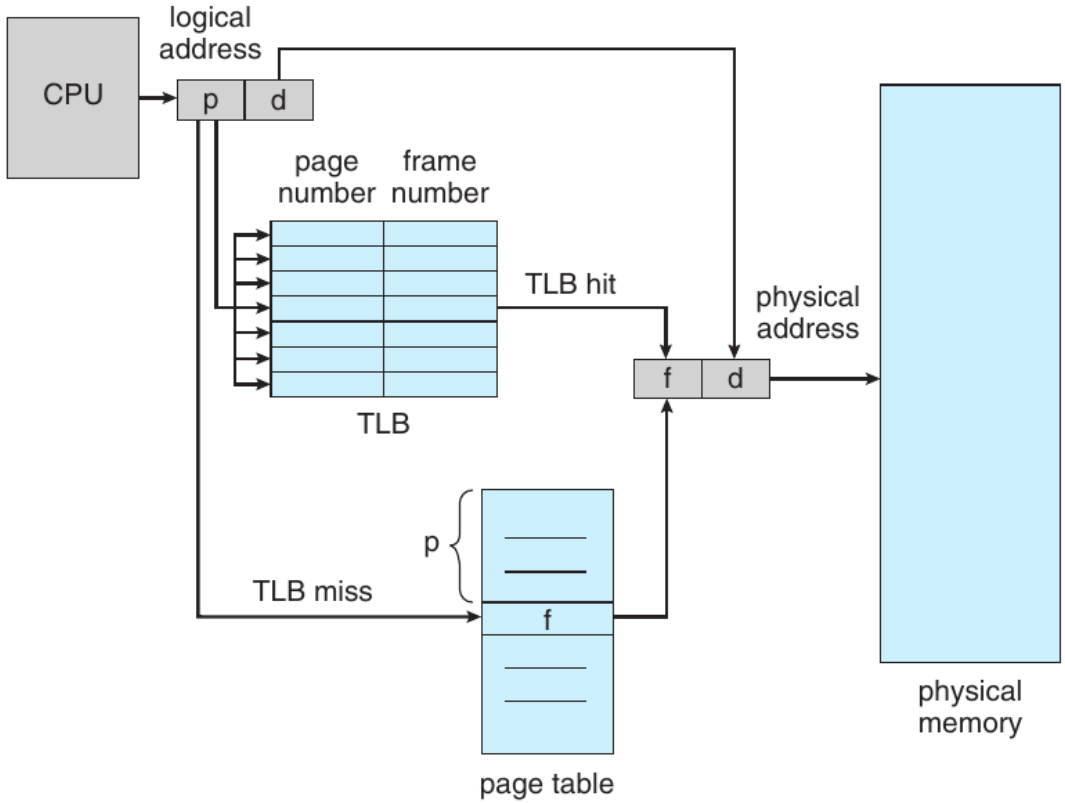

Figure: Paging Hardware With TLB

Courtesy of [SGG19]

Note: Logical address = virtual address, p = VPN, d = offset, f = PFN.

HW Paging Algorithm with TLB

The following pseudocode shows a possible MMU addressing algorithm when using the TLB:

```
VPN = (VirtualAddress & VPN_MASK) >> OFFSET_SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)    // TLB Hit
  if (CanAccess(TlbEntry.ProtectBits) == True)
    Offset = VirtualAddress & OFFSET_MASK
    PhysAddr = (TlbEntry.PFN << OFFSET_SHIFT) | Offset
    Register = AccessMemory(PhysAddr)
  else
    RaiseException(PROTECTION_FAULT)

else    // TLB Miss
  PTEAddr = PTBR + (VPN * 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()
```

TLB Example

Assume an 8-bit virtual address space with 16-byte pages (4-bit VPN). At address 100, there is an array of 10 integers (4 bytes each):

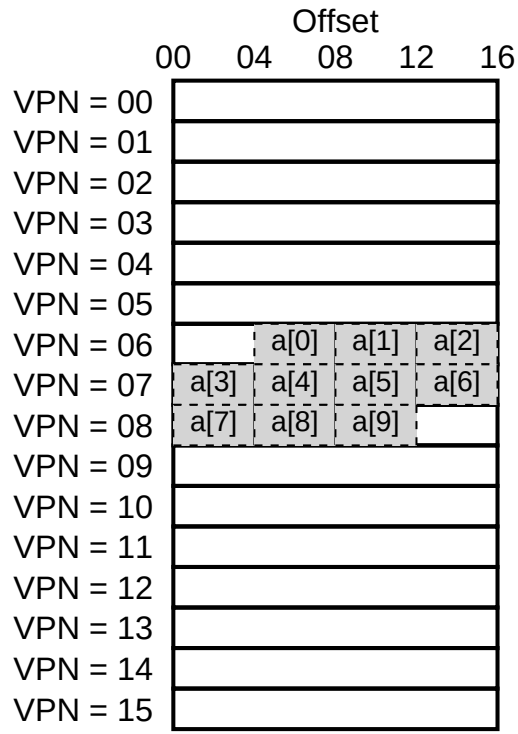

Figure: Array Example

Courtesy of [ADAD18]

TLB Example (cont.)

Again, we will loop through the array:

```
int sum = 0;
for (int i = 0; i < 10; i++) {
  sum += a[i];
}
```

When starting with an empty TLB and considering only memory accesses for the array itself, we obtain a TLB hit rate of 70%:

Access	TLB	Page Table
a[0]		✔
a[1]	✔	
a[2]	✔	
a[3]		✔
a[4]	✔	
a[5]	✔	
a[6]	✔	
a[7]		✔
a[8]	✔	
a[9]	✔	

Locality of Reference

70% looks like a lot. However, in practice, a good TLB hit rate is not an exception due to locality of reference .[4] It is the main principle behind hardware caches!

Due to intrinsic properties of code, and the way the CPU executes it, different types of locality occur in memory accesses. The two most important ones are:

Spatial Loc. It is likely, that a different access to a location close to a recently accessed location will occur.
Example: array accesses.

Temporal Loc. It is likely, that a recently accessed location will be accessed again in near future.
Example: counter variables.

[4] Sometimes also called the principle of locality .

Locality of Reference Example

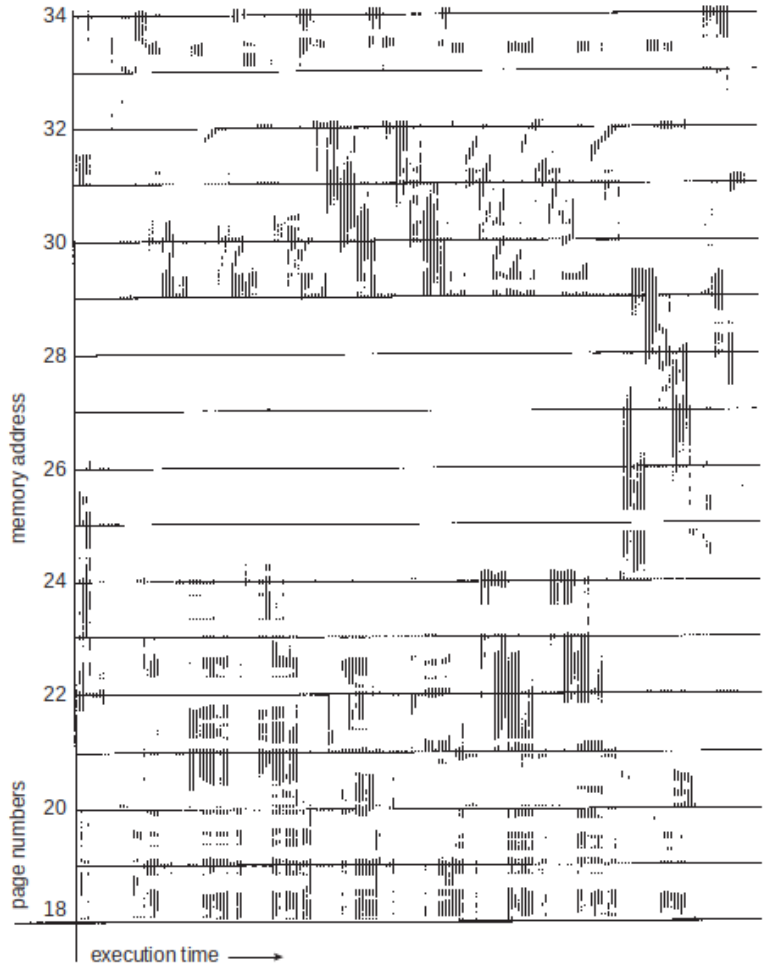

Figure: Locality in a Memory Reference Pattern

Courtesy of [SGG19]

Handling TLB Misses

TLB misses can be handled either in hardware or in the OS:

When handled by the *hardware*, format of the page table is defined by it and its location must be configured in a PTBR. This is how x86 does (PTBR is `CR3`). The algorithm is given on Slide 23.

Another option is to handle TLB misses in the OS (e.g. *MIPS R10k* and *SPARC v9*) architectures. This simplifies the addressing algorithm (see Slide 29), but complicates trap handling:

- `return-from-trap` after a TLB miss needs to return to the *previous* PC!
- Infinite TLB misses must be avoided (due to TLB-miss handler address not in TLB…)

HW Paging Algorithm with TLB (OS Handles Misses)

The following pseudocode shows a possible MMU addressing algorithm when using the TLB and TLB misses are handled by the OS.

```
VPN = (VirtualAddress & VPN_MASK) >> OFFSET_SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)    // TLB Hit
  if (CanAccess(TlbEntry.ProtectBits) == True)
    Offset = VirtualAddress & OFFSET_MASK
    PhysAddr = (TlbEntry.PFN << OFFSET_SHIFT) | Offset
    Register = AccessMemory(PhysAddr)
  else
    RaiseException(PROTECTION_FAULT)

else    // TLB Miss
    RaiseException(TLB_MISS)
```

TLB Contents

TLBs are fully associative (i.e. cache entries can be located anywhere in the TLB and are searched in parallel) and contain a few (say 64 or 128) TLB entries . A TLB might look like this:

VPN	PFN	Valid	Protection
10	100	1	read,write,execute
–	–	0	–
11	110	1	read,write
–	–	0	–

⚠ *In the TLB, the valid bit has different meaning: it only indicates if an entry is a valid translation or not (recall: in the page table, it states that a page is allocated; if not, a page fault occurs). Protection bits are similar to the protection bits in the page table.*

TLB and Context Switches

As TLB entries are *not* bound to a specific process, a problem occurs when performing a context switch.

Example:
Assume two processes, P1 and P2, which both have mapped page 10 (but to a different PFN). The TLB looks as follows:

VPN	PFN	Valid	Protection
10	100	1	read,write,execute
–	–	0	–
10	200	1	read,write,execute
–	–	0	–

How should the hardware know, which entry is currently valid?

TLB and Context Switches (cont.)

There are different solutions to this problem:

- *Flushing the TLB* on *every* context switch (i.e. mark entries as invalid). Could be done automatically on PTBR change or using a privileged instruction. However, it *destroys state* and is inefficient.
- *Marking process affiliation* in the TLB, using address space identifiers (ASID) , also called process-context identifiers (PCID) . This requires a separate register to indicate the currently running process.

Here, the same TLB with two processes but using *ASIDs*:

VPN	**PFN**	**Valid**	**Protection**	***ASID***
10	100	1	read,write,execute	*1*
–	–	0	–	–
10	200	1	read,write,execute	*2*
–	–	0	–	–

Typical TLB Characteristics

Here are some typical characteristics of real world TLBs:

- TLB size : 16 to 512 entries
- Block size: 1 to 2 page table entries
- Hit time: 0.5 to 1 clock cycle
- Miss penalty: 10 to 100 clock cycles
- Miss rate: 0.01% to 1%

Source: [PH18]

Multi-level Page Tables

Fixing Memory Consumption

By introducing TLBs, the performance issues due to additional memory accesses became manageable.

We still have the second issue though: *memory consumption*. Recall from Slide 13, that even moderate assumptions about the number of PTEs and their size, indicate that large amounts of memory will be required for the page table.

As an example: For the x86 PTE given on Slide 15 and a 32 bit address space, the table requires 4 MiB of memory per process!

Large / Huge Pages

A simple solution would be to *increase page size*. Indeed some CPUs support different page sizes since the 1990s.[5]

These so-called large pages or huge pages have mainly been introduced to reduce pressure on the TLB for applications which frequently use large data structures (e.g. RDBMS). As a side effect, they also reduce page table size.

On the downside: *internal fragmentation* returns! Thus, large pages in general are only used in specialized applications.

[5] x86 supports pages with 4 KiB, 2 MiB and 1 GiB, see [int, Vol. 3A, Section 4.10.2.1]

Sparse Page Tables

In general, it can be observed that page tables themselves are often sparse, i.e. only a few pages of the whole address space are valid / in use:

VPN	PFN	Valid	Protection	Present	Dirty
00	10	1	read,execute	1	0
01	–	0	–	–	–
02	–	0	–	–	–
03	–	0	–	–	–
04	23	1	read,write	1	1
05	–	0	–	–	–
06	–	0	–	–	–
07	–	0	–	–	–
08	–	0	–	–	–
09	–	0	–	–	–
10	–	0	–	–	–
11	–	0	–	–	–
12	–	0	–	–	–
13	–	0	–	–	–
14	28	1	read,write	1	1
15	04	1	read,write	1	1

Thus, memory consumption of page tables could be reduced by only keeping active parts in memory!

Multi-level Page Tables

One possible solution to make a page table sparse, is to turn it into a tree-like structure, called a multi-level page table . This approach is used by many modern systems, the basic idea is:

- Split the page table into *page-sized chunks*
  - If the whole chunk contains no valid PTEs, don't allocate it
- Use a page directory to look up the page table chunks
  - It returns the physical location of a chunk, if it contains any valid PTEs
  - *Or* it indicates that the complete chunk is unused

Linear- and Multi-level Page Table

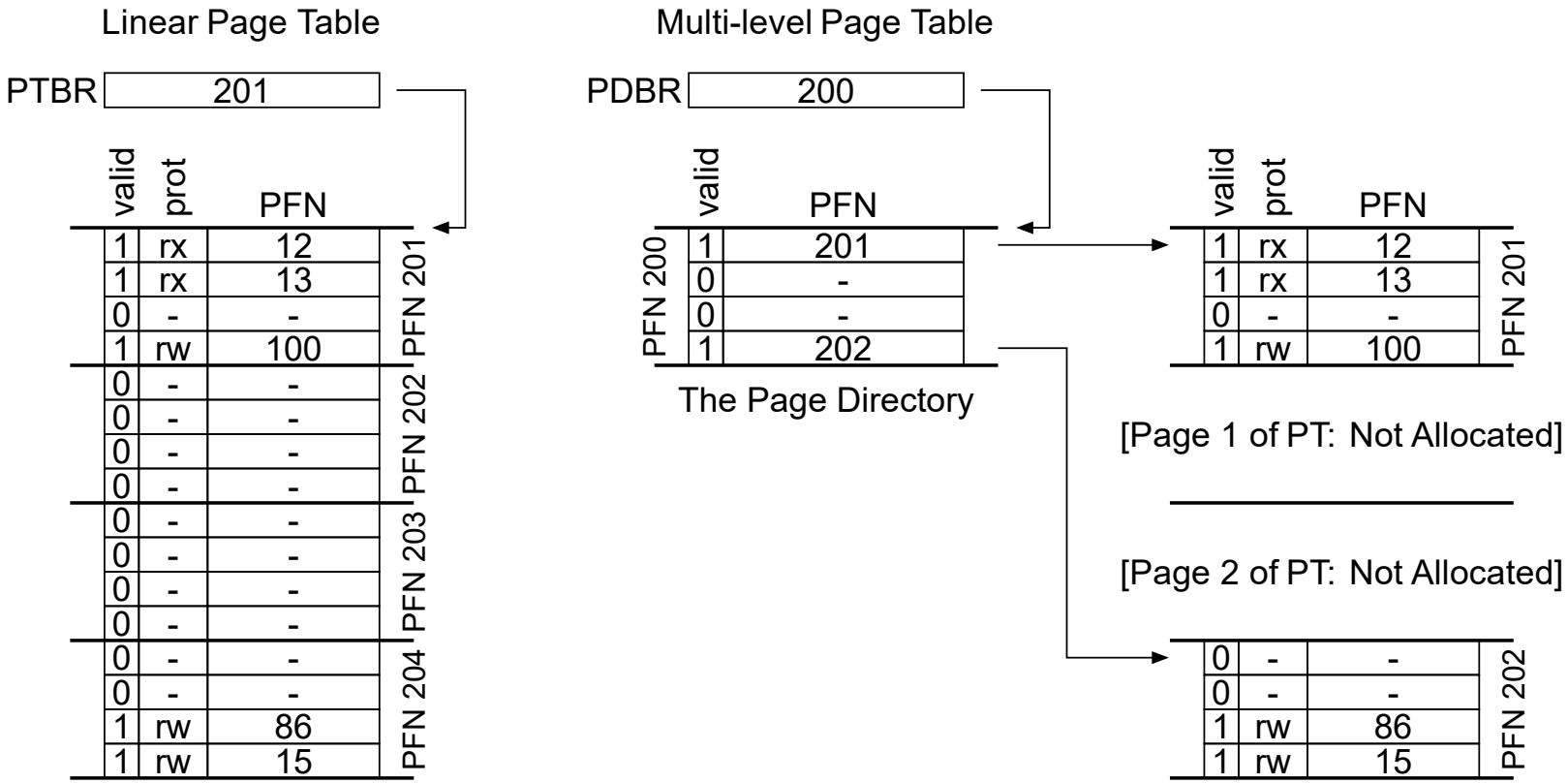

Figure: Linear- and Multi-level Page Table

Courtesy of [ADAD18], Own Modification

The Page Directory

The page directory is a simple table with one entry per chunk of the page table. It consists of page directory entries (PDE) , which at least contain the following fields:

- A *valid bit*, indicating if at least one page of the referenced page table chunk is valid[6]
- The PFN of the page table chunk

A page directory base register (PDBR) points the MMU to its physical location.

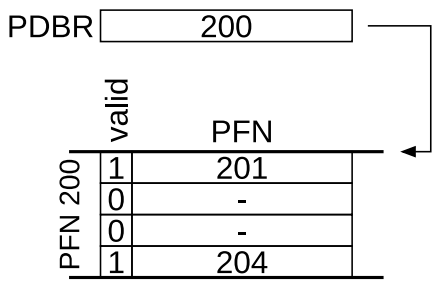

Figure: The Page Directory

Courtesy of [ADAD18], Own Modification

[6] ⚠ Again, this is not the same as the valid bit in the PTE!

Multi-level Pros and Cons

Advantages of multi-level page tables:

- Memory for the page table is allocated *proportionally* to the space used by a process (thus in general compact and supporting sparse address spaces)
- As the chunks of the table fit into pages themselves, managing memory for the page table is easier for the OS (it is free to place them wherever they fit)

However, multi-level page tables are more complex and they incur a *time-space trade-off*: on a TLB miss, two (or more) memory accesses are required to retrieve translation information.

Multi-level Example

Assume a 16 KiB address space (addresses are 14 bits) with pages of 64 bytes (VPN is 8, offset 6 bits). If a linear page table was used, it would contain 256 PTEs. Six pages are in use: 0 and 1 for code, 4 and 5 for heap and 254, 255 for the stack:

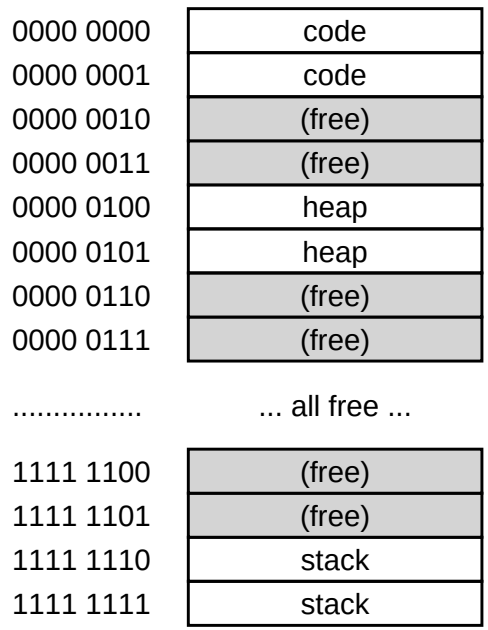

Figure: Example Address Space

Courtesy of [ADAD18]

Multi-level Example (cont.)

Assuming 4 bytes per PTE, a linear page table would be 1 KiB and require 16 pages ($256 \cdot 4/64$), holding 16 PTEs each.

We now split the VPN into 4 bits to address the page table chunk in the page directory and 4 bits to address an individual PTE in a given chunk:

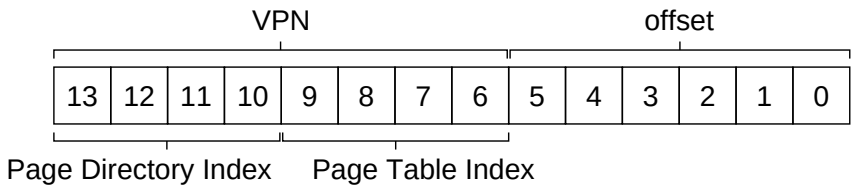

Figure: Splitting VPN into PDI and PTI

Courtesy of [ADAD18]

To address a page directory entry (PDE), we proceed as follows:

```
PDI = (VPN & PD_MASK) >> PD_SHIFT
PDE = PDBR + (PDI * sizeof(PDE))
```

Multi-level Example (cont.)

Based on the PFN in the PDE found in the previous step, the PTE can then be found (PTI is given in the address):

```
PTE = (PDE.PFN << OFFSET_SHIFT) + (PTI * sizeof(PTE))
```

Returning to our example, assume the following page directory (12 entries omitted):

PFN	Valid
100	1
–	0
...	...
101	1

Note: Only page table chunks 0 and 15 are valid.

Multi-level Example (cont.)

Page table chunk 0 (PFN 100) contains the pages for code and heap (8 entries omitted):

PFN	Valid	Protection
10	1	read,execute
23	1	read,execute
–	0	–
–	0	–
80	1	read,write
59	1	read,write
–	0	–
...	...	...

Multi-level Example (cont.)

Page table chunk 15 (PFN 101) contains the pages for the stack (12 entries omitted):

PFN	Valid	Protection
...	...	...
–	0	–
55	1	read,write
45	1	read,write

Note how the multi-level page table requires only 3 pages instead of 16 for the linear page table!

Multi-level Example (cont.)

We end this example by translating a virtual address into a physical address using the multi-level page table given before.

Transform virtual address 0x3F80 (binary: `11 1111 1000 0000`):

1. Extract page directory index (PDI, top 4 bits): `1111` is PDE 15 which is valid and points to PFN 101
2. Next 4 bits are the PTE index (PTI): `1110` is PTE 14 of page table chunk 15
3. Chunk 15, PTE 14 (page 254) is valid and mapped at PFN 55
4. Concatenate PFN with offset (0x00) to obtain the physical address:

   ```
   PhysAddr = (PTE.PFN << OFFSET_SHIFT) + Offset
   ```

   For PFN 55 (shift: 6) and offset 0, this is physical address 0x0DC0 (`00 1101 1100 0000`) □

More Levels

More than two levels are possible and required when supporting larger address spaces (e.g. Linux supports 4- and 5-level paging).

Example:
Given a 30-bit virtual address space and 512 byte pages (VPN is 21, offset 9 bits). A PTE is again 4 bytes. How many *page table levels* are required? Recall: the goal is to make every part of the page table fit into a page.

1. Determine how many PTEs fit in a single page. For the example: 128, requiring 7 bits. This leaves 14 bits for the PDE index, requiring 128 pages for the directory!
2. Add another level using 7 bits of the VPN. Now, everything fits into individual pages.

More Levels (cont.)

The example is illustrated in the following figure.

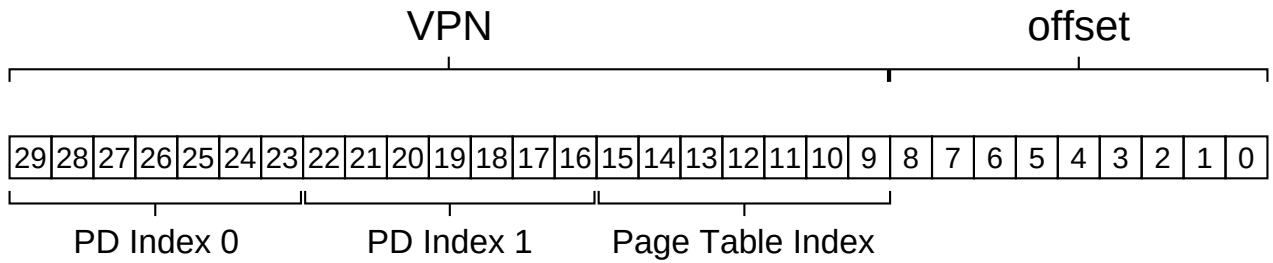

Figure: Two-level Page Table Addresses

Courtesy of [ADAD18]

Multi-level Addressing Algorithm

The following pseudocode shows a possible MMU addressing algorithm for a two-level page table.

```
VPN = (VirtualAddress & VPN_MASK) >> OFFSET_SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)    // TLB Hit
  if (CanAccess(TlbEntry.ProtectBits) == True)
    Offset = VirtualAddress & OFFSET_MASK
    PhysAddr = (TlbEntry.PFN << OFFSET_SHIFT) | Offset
    Register = AccessMemory(PhysAddr)
  else
    RaiseException(PROTECTION_FAULT)


else    // TLB Miss
  // 1) Get PDE
  PDI = (VPN & PD_MASK) >> PD_SHIFT
  PDEAddr = PDBR + (PDI * sizeof(PDE))
  PDE = AccessMemory(PDEAddr)
  if (PDE.Valid == False)
    RaiseException(SEGMENTATION_FAULT)
  else
    // Get PTE
    PTI = (VPN & PT_MASK) >> PT_SHIFT
    PTEAddr = (PDE.PFN << OFFSET_SHIFT) + (PTI * sizeof(PTE))
    PTE = AccessMemory(PTEAddr)
    if (PTE.Valid == False)
      RaiseException(SEGMENTATION_FAULT)
```

Multi-level Addressing Algorithm (cont.)

```
else if (CanAccess(TlbEntry.ProtectBits) == False)
  RaiseException(PROTECTION_FAULT)
else
  TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
  RetryInstruction()
```

Inverted Page Tables

Another idea is to use inverted page tables : Instead of having one page table per process, have a *single* table for all *physical pages* in the system. A PTE then indicates to which process a certain page belongs.

Inverted page tables tend to be large and require efficient strategies for searching. They are used for example on the *PowerPC* architecture.

Outlook on Memory 3

We have introduced paging, and had a look at translation-lookaside buffers (TLBs) and multi-level page tables.

Paging offers the following main advantages:

Flexibility The OS can support any address space layout required by a process without much effort (e.g: it does not have to care about stack orientation).

Simplicity Free-space management is much simpler: Creating or resizing an address space just requires finding enough free pages in the free list.

Efficiency Paging helps dealing with internal *and* external fragmentation. Using TLBs it is almost as efficient as direct memory accesses.

In the next part, we will look at the mechanism of swapping and corresponding policies, concluding the first part of this course.

Appendix

Bibliography

[ADAD18] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, 1.00 ed., Arpaci-Dusseau Books, August 2018, Available online: `http://ostep.org`.

[int] *Intel 64 and IA-32 architectures software developer manuals*, `http://www.intel.com/products/processor/manuals`.

[PH18] David A. Patterson and John L. Hennesy, *Computer Organization and Design, RISC-V Edition*, 5 ed., Morgan Kaufmann, 2018.

[SGG19] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne, *Operating system concepts*, 10th ed., Wiley Publishing, 2019.