Betriebssysteme

7. Tutorium - Paging

Peter Bohner

7. Januar 2026

1 Paging

Paging

What is that? What is the difference to Segmentation?

- Virtual memory is broken into fixed-size chunks (*pages*)
- Physical memory is broken into fixed-size chunks of the same size (*frames*)
- Virtual pages are mapped to page frames Always? No! (Paged out, zero pages, . . .)

Benefits over Segmentation?

- Virtual memory does not need to map to *continuous* physical memory
- Swapping in/out is easier
- No external fragmentation, little internal

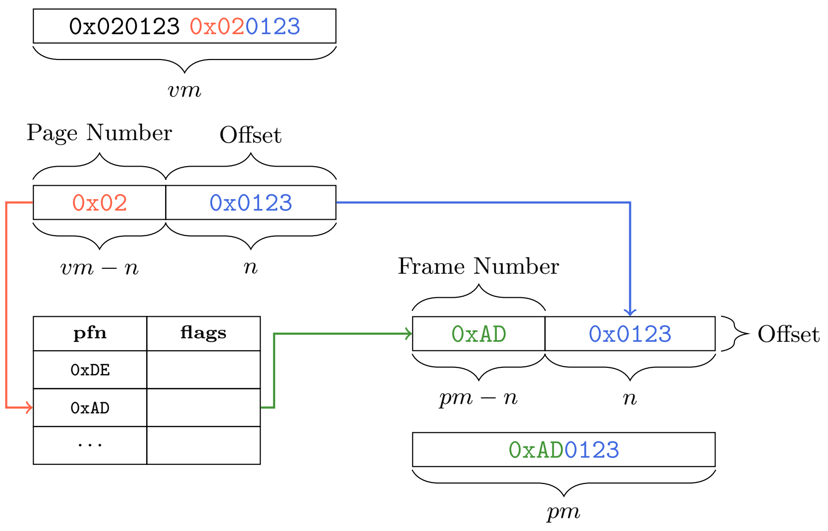
Paging - Single Level Page Table

Segment and Page tables

Segment Number	Base	Limit
0	`0xdead`	`0x00ef`
1	`0xf154`	`0x013a`
2	`0x0000`	`0x0000`
3	`0x0000`	`0x3fff`

Paging - Single Level Page Table

Segment and Page tables



Single Level Page Table - Disadvantages

Why could that be a bit problematic?

- Increases with the size of the *virtual* address space
- 64 Bit AS, 4KiB (2^{12}) pages $\Rightarrow n = 12 \Rightarrow 2^{vm-n} = 2^{64-12} = 2^{52}$

$\Rightarrow$ If every entry was 1 Bit we'd need (asking `units`...)

```
You have: 2^52 bit
You want: tebibyte
        * 512
        / 0.001953125
```

- You might *not* have that much memory to spare :)

Single Level Page Table - Disadvantages

Math is fun, let's do some math

Calculate the space requirements for a single level page table with

- 32-bit virtual addresses, 4KiB pages, 4 bytes per page table entry
- 48-bit virtual addresses, 4KiB pages, 4 bytes per page table entry

32-bit

- vm = 32, 4Kib = 2^{12} $\Rightarrow$ n = 12
- $2^{32-12} = 2^{20}$ entries $\Rightarrow 2^{20} \cdot 2^{2} = 2^{22}$ Byte (4 MiB)

48-bit

- vm = 48, 4Kib = 2^{12} $\Rightarrow$ n = 12
- $2^{48-12} = 2^{36}$ entries $\Rightarrow 2^{36} \cdot 2^{2} = 2^{38}$ Byte (256 GiB)

Alternatives to Single Level Page Tables

Mutli-Level page tables

$$\overbrace{\boxed{\texttt{0xDE}}}^{\text{1st Level PT}}\overbrace{\boxed{\texttt{AD}}}^{\text{2nd Level PT}}\overbrace{\boxed{\texttt{BE}}}^{\text{3rd Level PT}}\overbrace{\boxed{\texttt{EF}}}^{\text{Offset}}$$

Benefits and Drawbacks?

- Pointer chasing down each level $\Rightarrow$ More memory accesses

+ Address spaces are *sparse* $\Rightarrow$ Only instantiate page tables you need

Inverted Page Table

What do those do?

- Map *physical* addresses to virtual ones
- Why? Physical address space is much smaller and you don't need a table per address space
- If you don't have a table per address space, how can you still have multiple?

⇒ e.g. Linked List of entries per physical frame

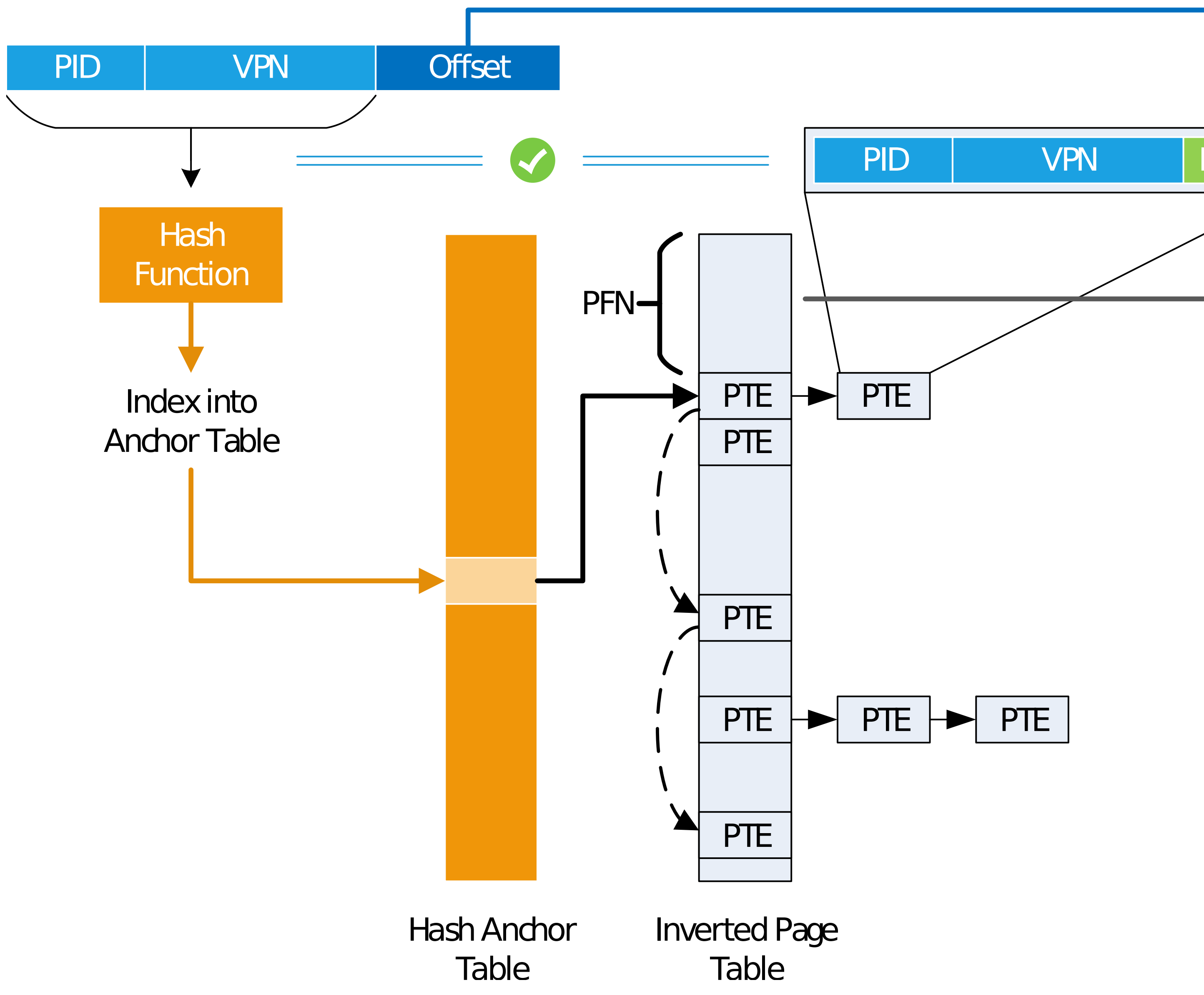
Any drawbacks?

- We mostly care about *the other direction*, i.e. virtual ⇒ physical
- That requires iteration :(

Inverted Page Tables

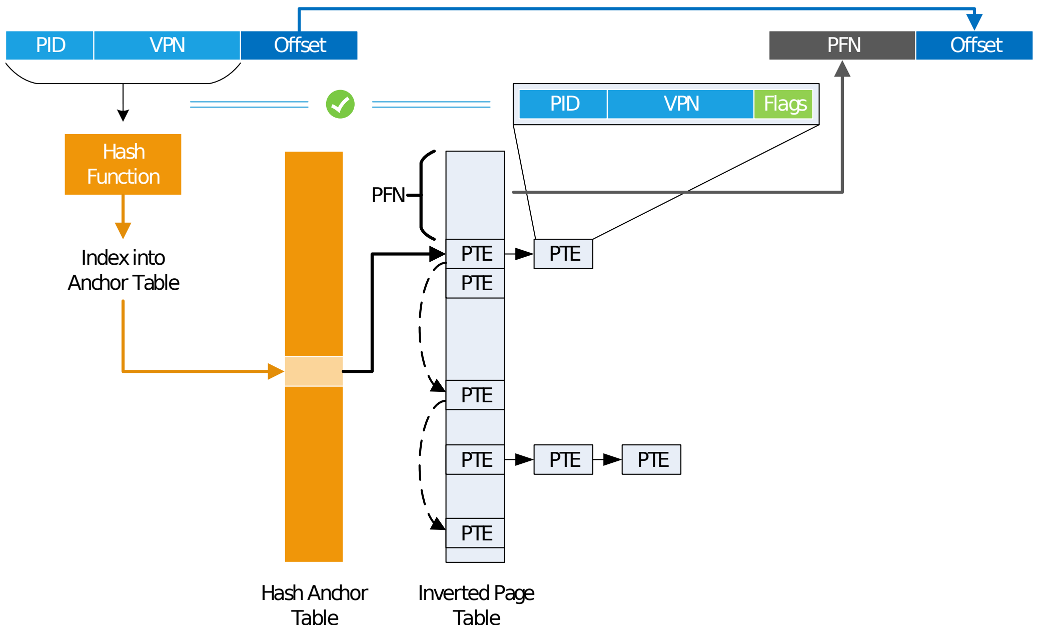
How can we speed them up?
After having attended *Algorithmen I* we all know: Hacktables are O(1)

Hashed inverted page tables



Inverted Page Tables

Hashed inverted page tables



Why do pages have a valid/present bit?

What is it for?

- The page might not be present. Why? Swapped out, not zeroed yet, . . .

What happens on access if it is not set?

- Page fault!
- Handle it and do sth. sensible (or crash the process. . .)

2 TLB

TLB

What does that refer to?

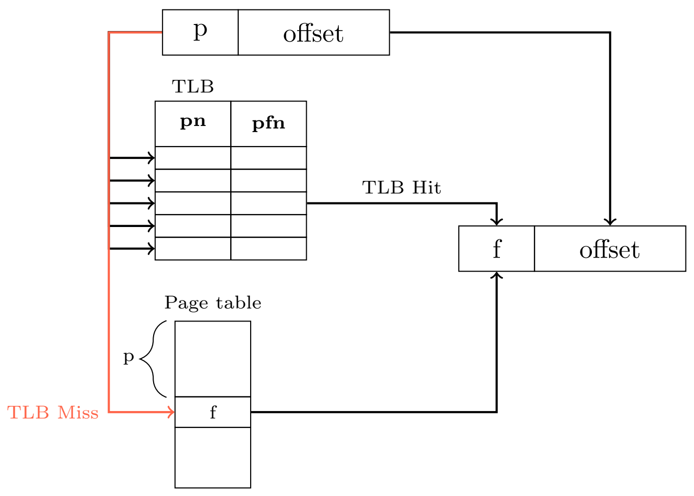
Translation **L**ookaside **B**uffer. What's that for?

- Lookup from virtual to physical address can be *slow*
- You need to translate addresses *all the time*

⇒ There is no problem you can't solve with another caching layer (except having too many caching layers)
Nearly the Fundamental theorem of software engineering

TLB

TLB layout



Switching tables

Why is switching page tables (`CR3` register) expensive?
You need to flush the TLB!

How could you make this cheaper by modifying the hardware?
Tag TLB entries with an address space identifier
    Each lookup also checks the Tag ⇒ Can ignore entries for old processes ⇒ No need to flush
    This is implemented in many newer architectures (e.g. ARM).

TLB - Misses

What is the difference between software and hardware walked Page Tables?
The TLB is *the same*, what differs is the behaviour on a *cache miss*.

Hardware walked

- Hardware looks at your page table
- Hardware *resolves* the physical address using it
- On failure a page fault is raised

⇒ What does that imply? Page table layout is *fixed*

Software walked

- Transfers control to TLB miss handler
- Kernel routine walks the page table and tries to find a mapping
- Loads that mapping into the TLB and can choose which entry to evict!
- If there is none ⇒ Jump to page fault handler

TLB - Software or hardware walked

What are benefits / drawbacks of software walked TLBs?

+ Free to choose page table layout (fit your algorithm)
+ Free to choose which TLB entries to evict ⇒ Use optimized strategy
- Greater overhead

TLB - Contents

What information does the TLB store?

- Physical Frame Number, Virtual Page Number
- Valid / Present Bit
- Modified bit, permissions, ...

TLB - Page faults

When is a TLB miss or page fault raised?
Due to the TLB:

- Software walked: No entry found in TLB ⇒ TLB miss exception
- Hardware walked: No mapping in page table ⇒ Page fault

And other problems: *Invalid access to a mapped page*

- Software walked: Raise some kind of TLB fault if the page is e.g. read-only
- Hardware walked: Page fault raised, page fault handler has to find out what happend

3 Page Fault Handling

Page Fault Handling

What is Demand-Paging and Pre-Paging?
Demand-Paging:

- Load pages on-demand, right when they are needed

Pre-Paging:

- Loaded Pages speculatively in batches, even *before* you need them

Why would you (not?) use Demand-Paging?

+ Only loads needed data ⇒ Less memory wasted
- Generates lots of page faults before working set is in memory

Page Fault Handling

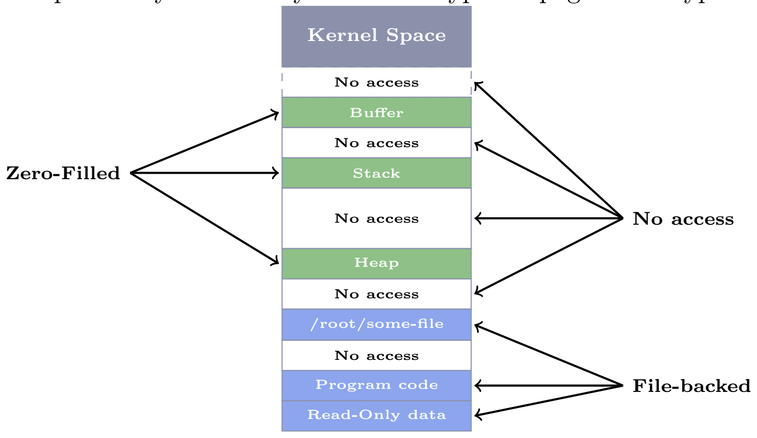
Why would you (not?) use Pre-Paging?

+ Might reduce number of page faults
- Loads more than needed ⇒ Wasteful
- More I/O ⇒ Slower?
+ HDDs a lot faster when reading chunks

On-Demand Paging

Different kind of page faults

Not all pages are created equal. Do you have any idea what types of page faults typically exist?



On Demand Paging

Different kind of page faults

Why are pages to generic memory zero filled? It could leak other processes' memory otherwise (called a Covert Channel).

And there is one other kind of page fault...

...The page was stolen by the OS and swapped out![1em] Also supported on some systems: *Purgable memory.* Stolen from Apple and also implemented in SerenityOS in this video.

Page faults

What kind of information does the page fault handler need?

- Access flags: Can the user perform the operation on this page?
- Where to find the most recent version (different for zero filled, file backed, etc.)

CoW

How could you implement Copy-on-Write memory?

- Mark memory as read-only on fork
- Add an additional `CoW` flag: When a page fault is raised check it, copy the page and clear the `CoW` and `ro` flag

4 Page replacement

Pager

The pager in some systems tries to keep „spare pages“. Why?

- OS needs free (pre-zeroed) frames to assign to processes
- What happens when there are none and a page fault occurs?

⇒ Needs to write dirty pages back to disk

⇒ Sloow

Page Replacement - Global and Local Algorithms

If you need to swap out a page, what pages do you search for a victim?
Local page replacement algorithms:

- Only look through the frames *of that process*

Global page replacement algorithms:

- Look through *all* frames, even that of other processes

Page Replacement - Global and Local Algorithms

What are the pro and cons of local algorithms?

- + Guaranteed number of pages per application
- + Potentially faster
- - Guaranteed number of pages per application ⇒ Can not adapt as easily to changing demands
- - Difficult selection of optimal number of frames per application (see point above)

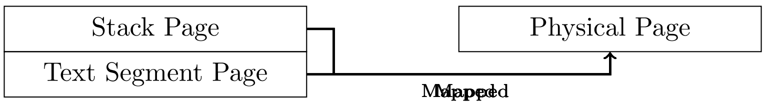
Every process should get an equal number of pages. Do you use a Global or Local Algorithm?
Local

Page Replacement - Working set

What is the working set?

- The set of pages that a process accessed in the last Δ page references

What is Thrashing?



- Not enough frames to fit the working set

⇒ Pages will be stored to disk and reloaded very often

4.1 Page Replacement Policies

Page Replacement Policies

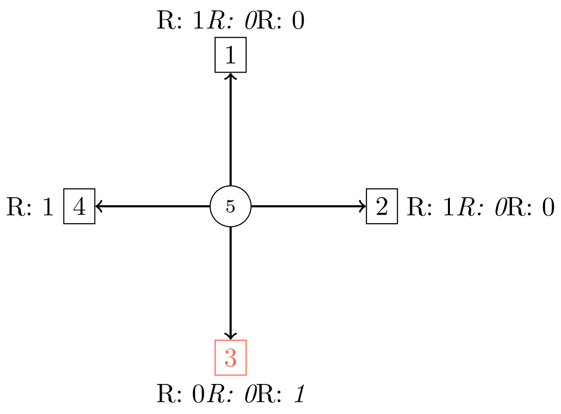
What are those?
We need to find a victim page to replace with a new one.

Common strategies

- **FIFO:** First page to be loaded is the first to be unloaded
- **LRU:** Least recently used page is evicted
- **Optimal:** Evict the page that is used the *furthest into the future*. Feasible? Used as a *benchmark*
- **Clock:** Uff, let's talk about that on its own page

Page Replacement Policies – Clock

Clock



Page Replacement Policies

Where did that R come from?

- Referenced (and modified) bits are set by the CPU when accessing or writing to the page
- Referenced bits are periodically cleared by the kernel using timer interrupts

Page Replacement Policies

Do it

- Clock: Ordered by Load time ASC, Head is on Frame 3
- Reference order: 4, 0, 0, 0, 2, 4, 2, 1, 0, 3, 2

Frame	Virtual page	Load time	Access time	Referenced	Modified
0	2	60	161	0	1
1	1	130	160	0	0
2	0	26	162	1	0
3	3	20	163	1	1

Page Replacement Policies

How well does LRU work for the Stack, the Heap and the Code segment?

- Stack: Beautifully. The older it is the longer it will take to be reached again
- Code: Mostly, loops are mostly linear and it follows certain patterns
- Heap: Well, the heap has more random access patterns. So not that well, probably