Universität Bamberg

Operating Systems Engineering

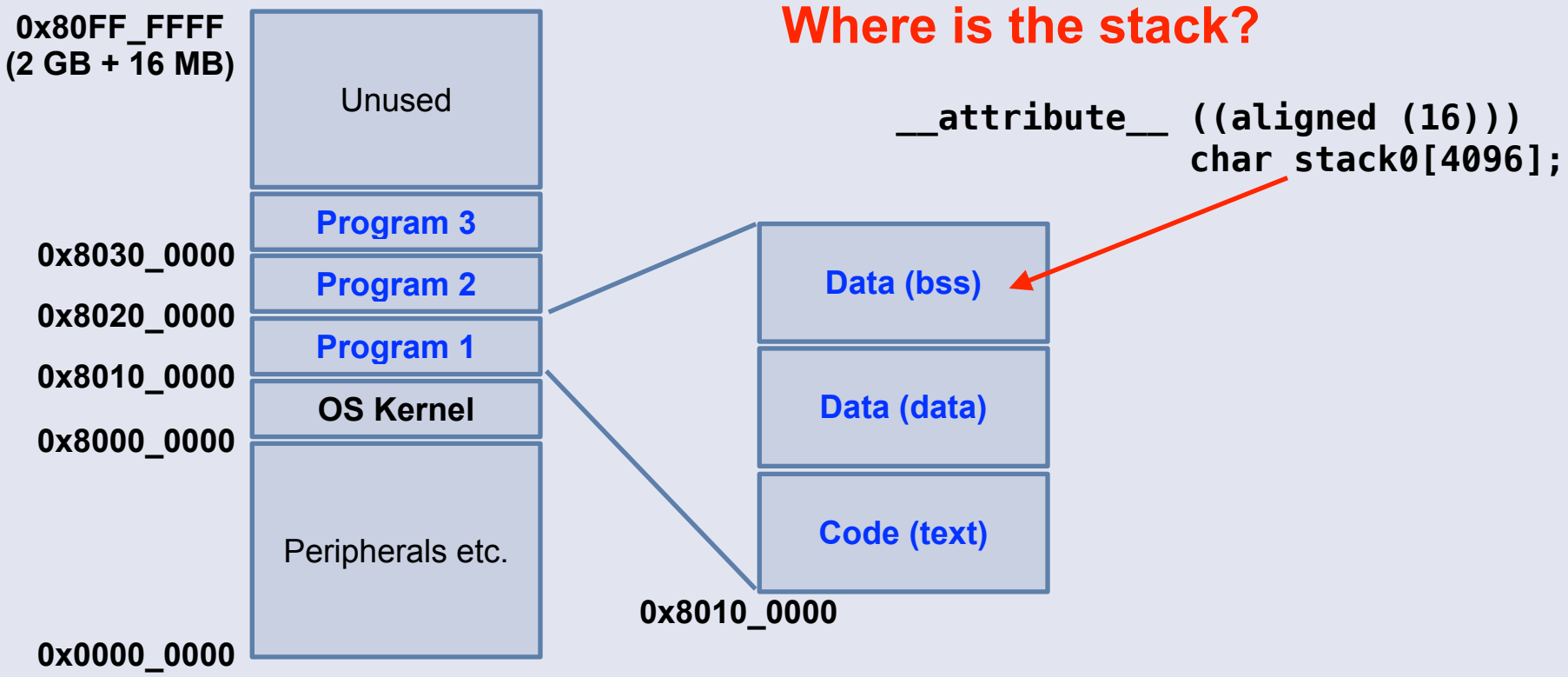
Lecture 7: Virtual Memory

Michael Engel (michael.engel@uni-bamberg.de)
Lehrstuhl für Praktische Informatik, insb. Systemnahe Programmierung
https://www.uni-bamberg.de/sysnap

Cooperative multitasking: memory view

Universität Bamberg

- **All processes have to remain in memory now**
  - Code (text segments) is static and could be reloaded, but the data programs manipulate (data + bss segments) would need to be saved



Virtual Memory

- Only using physical memory can be problematic
  - Programs have to be **linked to different memory ranges** in order **not to interfere** when running at the same time
    - …like in our OS!
  - Protection of memory is coarse-grained
    - PMP enables protection, but number of PMP entries is limited
  - Cannot (easily) handle programs/data larger than main memory

- **Solution:**
  - Add a component that translates addresses generated by the CPU (and, in turn, by programs) from those used to address the physical memory in a computer
  - Create the illusion that each process can use the whole (virtual) address space for itself

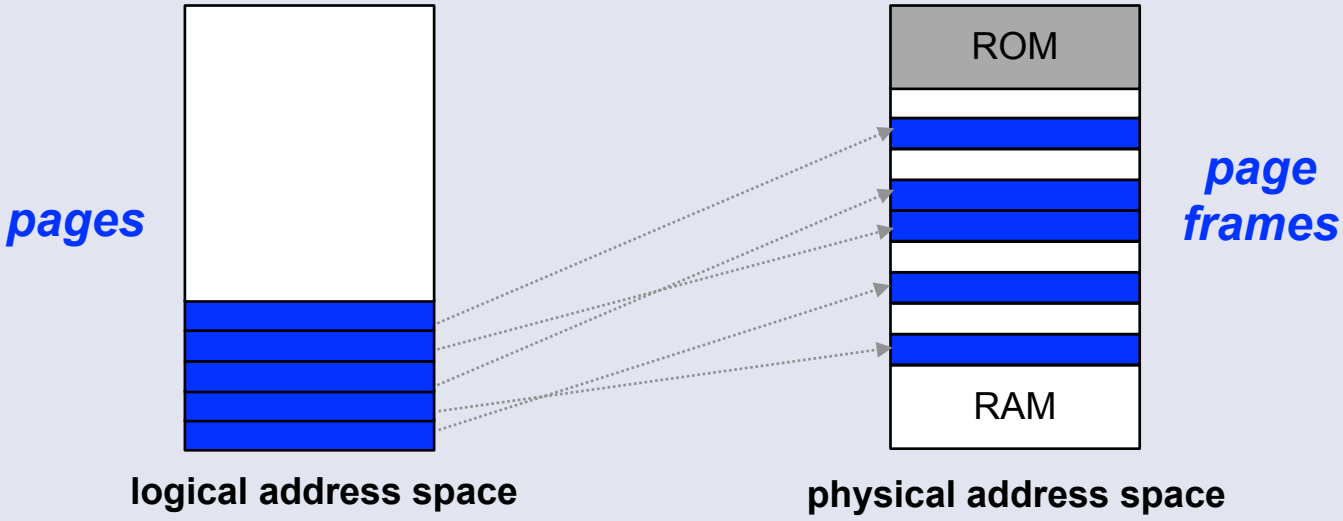
Virtual Memory

- **Additional benefit:**
  Decouple memory requirements from available amount of main memory
  - Processes do not access all memory locations with the same frequency
  - Certain instructions are used (executed) only very infrequently or not at all (e.g. error handling code)
  - Certain data structures are not used to their full extent
  - Processes can use more memory than available as main memory

- **Idea:**
  - Create the illusion of a large main memory
  - Make currently used memory areas available in main memory
  - Intercept accesses to areas currently not present in main memory
  - Provide required areas on demand
  - Swap or page out areas which are (currently) not used

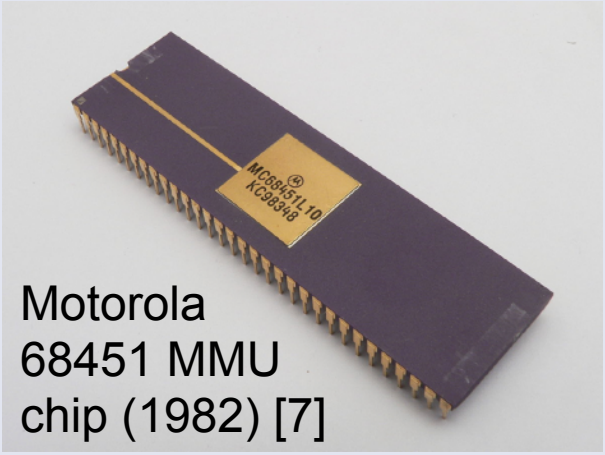
Paging

- **Logical address space** is split into **pages** of identical size
  - Pages can be located at arbitrary positions in the physical memory address space
  - Solves the fragmentation problem
  - No compaction necessary
  - Simplified memory allocation and swapping



MMU

- Idea: intercept “virtual” addresses generated by the CPU
  - MMU checks for “allowed” addresses
  - It translates allowed addresses to “physical” addresses in main memory using a translation table
- Problem: translation table for each single address would be large
  - Split memory into pages of identical size (power of 2)
  - Apply the same translation to all addresses in the page: **page table**
- MMUs were originally separate ICs sitting between CPU and RAM
  - Or even realised using discrete components (e.g. in the Sun 1 [8])
  - Higher integration due to Moore’s Law ➛ fit on CPU chip now!

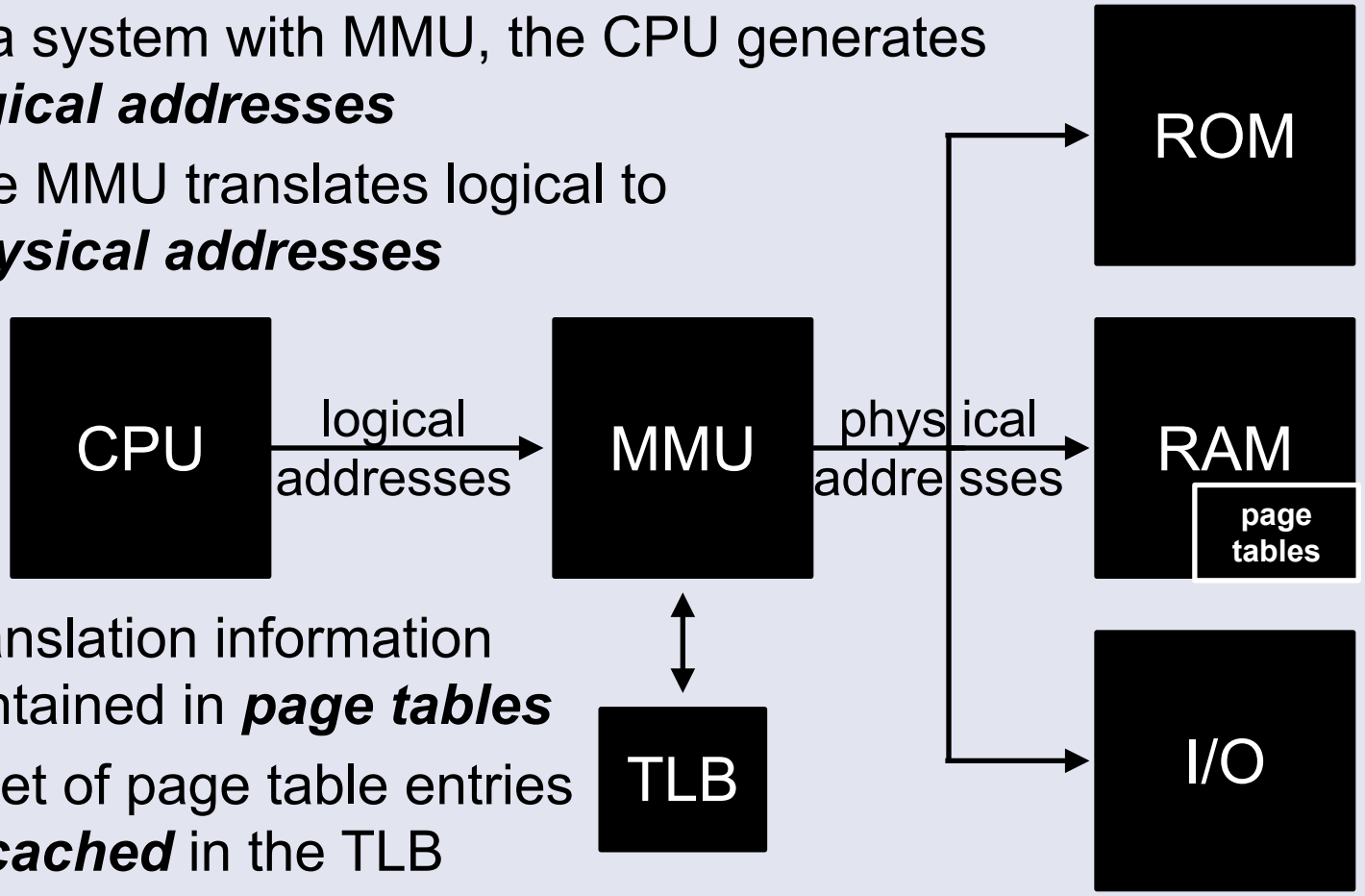


Motorola 68451 MMU chip (1982) [7]

[Wikimedia by David Monniaux, CC BY-SA 3.0]

Structure of a computer with MMU

- Addresses generated by software in the CPU are no longer directly used to address memory or peripherals
- In a system with MMU, the CPU generates ***logical addresses***
- The MMU translates logical to ***physical addresses***



- Translation information contained in ***page tables***
- A set of page table entries is ***cached*** in the TLB

Naive page tables

- We need to provide a physical ("**page frame**") address for each memory page
- How much memory does the page table use itself?

- Example: 4 GB (32 bit) address space, 4 kB (2^{12}) page size
  ⇒ $2^{32} / 2^{12} = 2^{20}$ (~1 million) entries – per process!

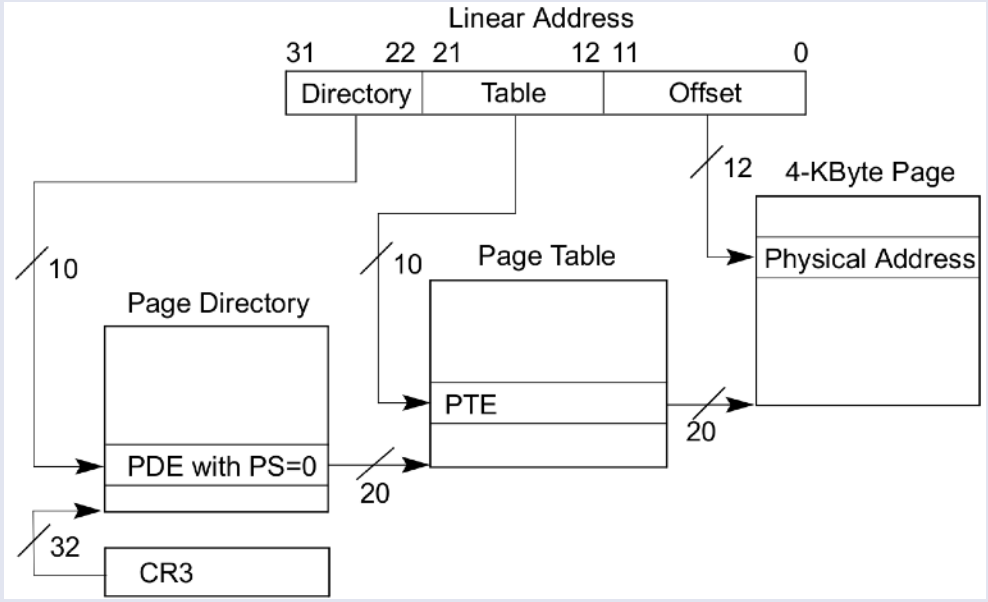
  ⇒ with 4 bytes per entry, the page table would use 4 MB RAM!

- Most of the page table entries would be empty
  - …unless a process uses a significant portion of its 4 GB memory space

- Idea: use ***sparse storage*** for page tables

Hierarchical page table structure

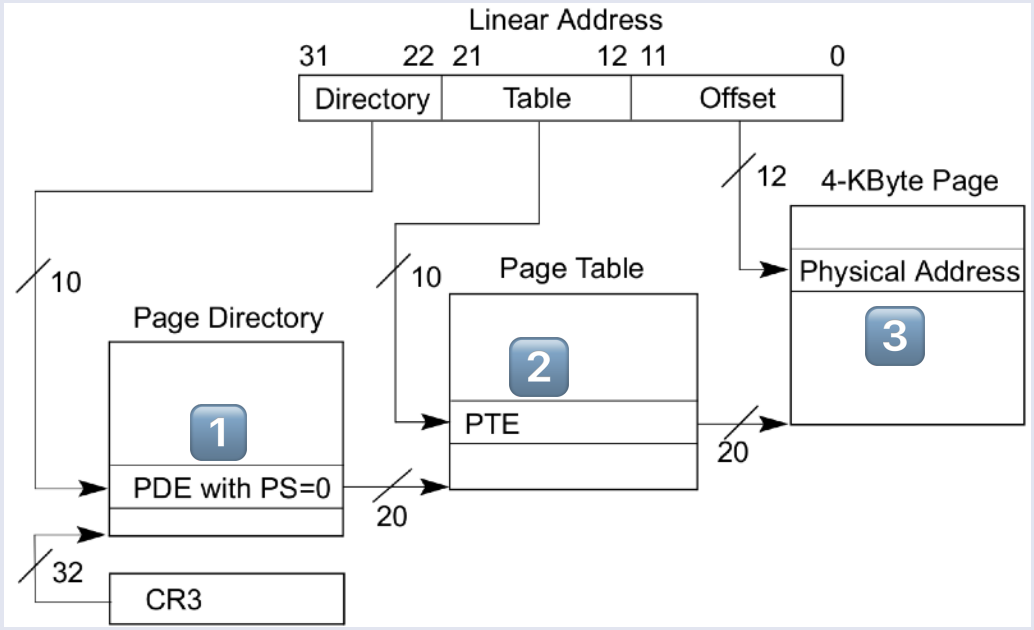
- Split memory into **pages** of identical size (power of 2)
- Apply the same translation to all addresses in the page: **page table**
- Find a compromise **page size** allowing flexibility and efficiency
  - Typically several kB: 4 kB=2^{12} bytes (x86), 16 kB (Apple M1)
- Use sparse multi-level page tables ➛ reduce page table size
- For 32 bit x86:
  - Page size:
    - 2^{12} = 4096 bytes
  - Page table:
    - 2^{10} page entries
  - Page directory:
    - 2^{10} page tables



Address translation

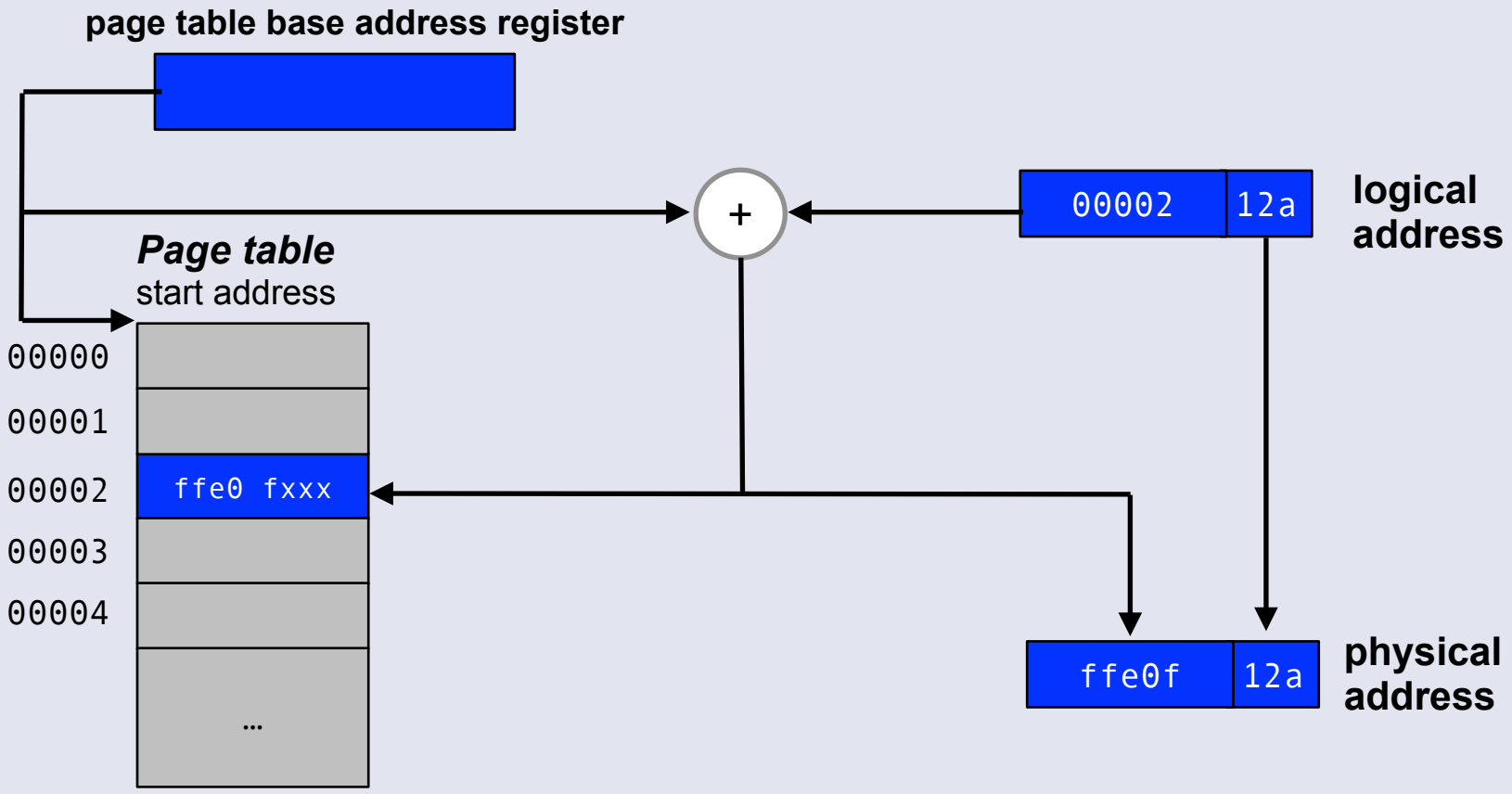
- The MMU splits the virtual (or “linear”) address coming from the CPU into three parts:
  - 10 bits (31–22) page directory entry (PDE) number
  - 10 bits (21–12) page table entry (PTE) number
  - 12 bits (11–0) page offset inside the references page (untranslated)

- **Translation process:**
  - Read PDE entry from dir.:
    ➛ address of one page table
  - Read PTE entry from table:
    ➛ physical base address of memory page
  - Add offset from original virtual address (bits 11–0) to obtain the complete physical memory address



MMU with page tables

- A table is used to translate *page addresses* into *page frame* addresses



MMU with page tables (2)

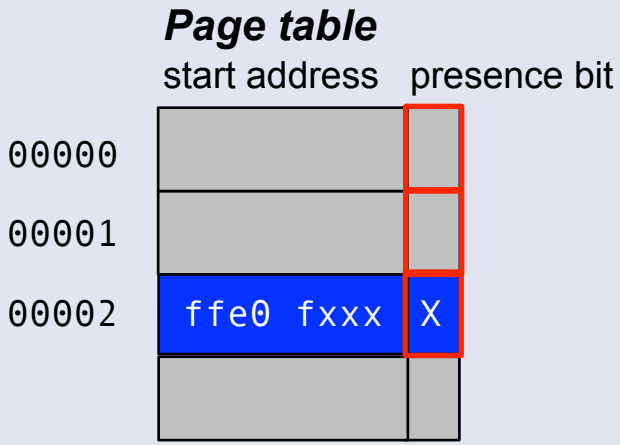
- Page-based addressing creates ***internal fragmentation***
  - The last page is often not used completely
- Page size
  - small pages reduce internal fragmentation, but increase the size of the page table (and vice versa)
  - common page sizes: 512 bytes — 8192 bytes

- Page tables are large and have to be kept in main memory
  - The OS has to ***create the page tables*** for each process before starting it and may have to ***update the page tables*** during the execution of a process
- Large number of implicit page accesses required to map an address

Page table metadata

- The physical address is not sufficient:
  Additional information required to efficiently handle virtual memory

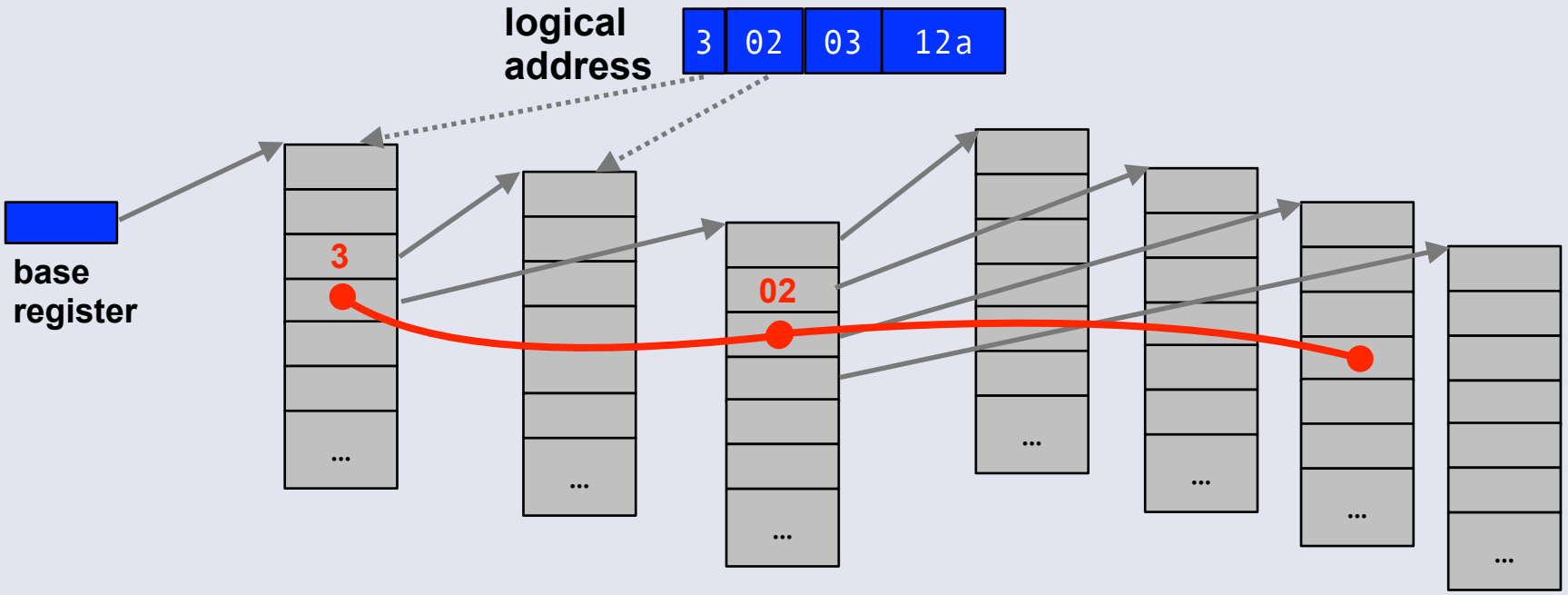
- Hardware support
  - If the ***presence bit*** is set,
    nothing changes
  - If the presence bit is cleared,
    a trap is invoked (***page fault***)
  - The trap handler (part of the OS) can
    now initiate the loading of the page
    from background storage
    (this requires hardware support in the CPU)



Multi-level page addressing

Universität Bamberg

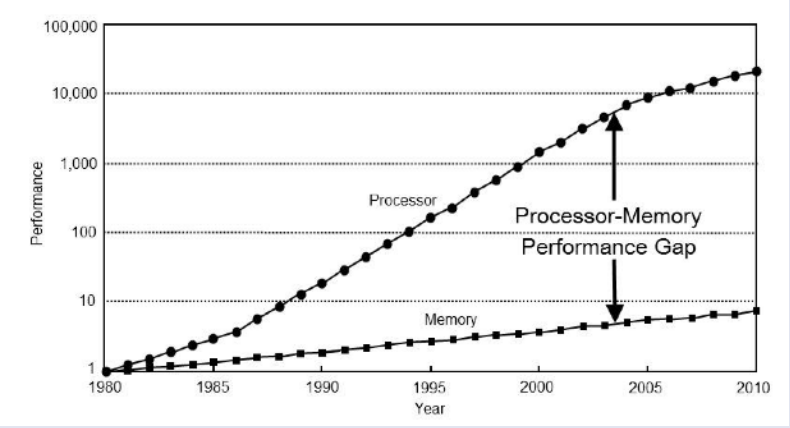
- Example: two-level page addressing



- Presence bit also for all entries in higher levels
  - This enables the swapping of page tables
  - Tables can be created at access time (on demand) – saves memory!
- However: even more implicit memory accesses required

Speeding up address translation

- Where is the page table stored?
- Can be several MB in size
  ➛ doesn't fit on the CPU chip!
- Page directory and page tables are in **main memory**!



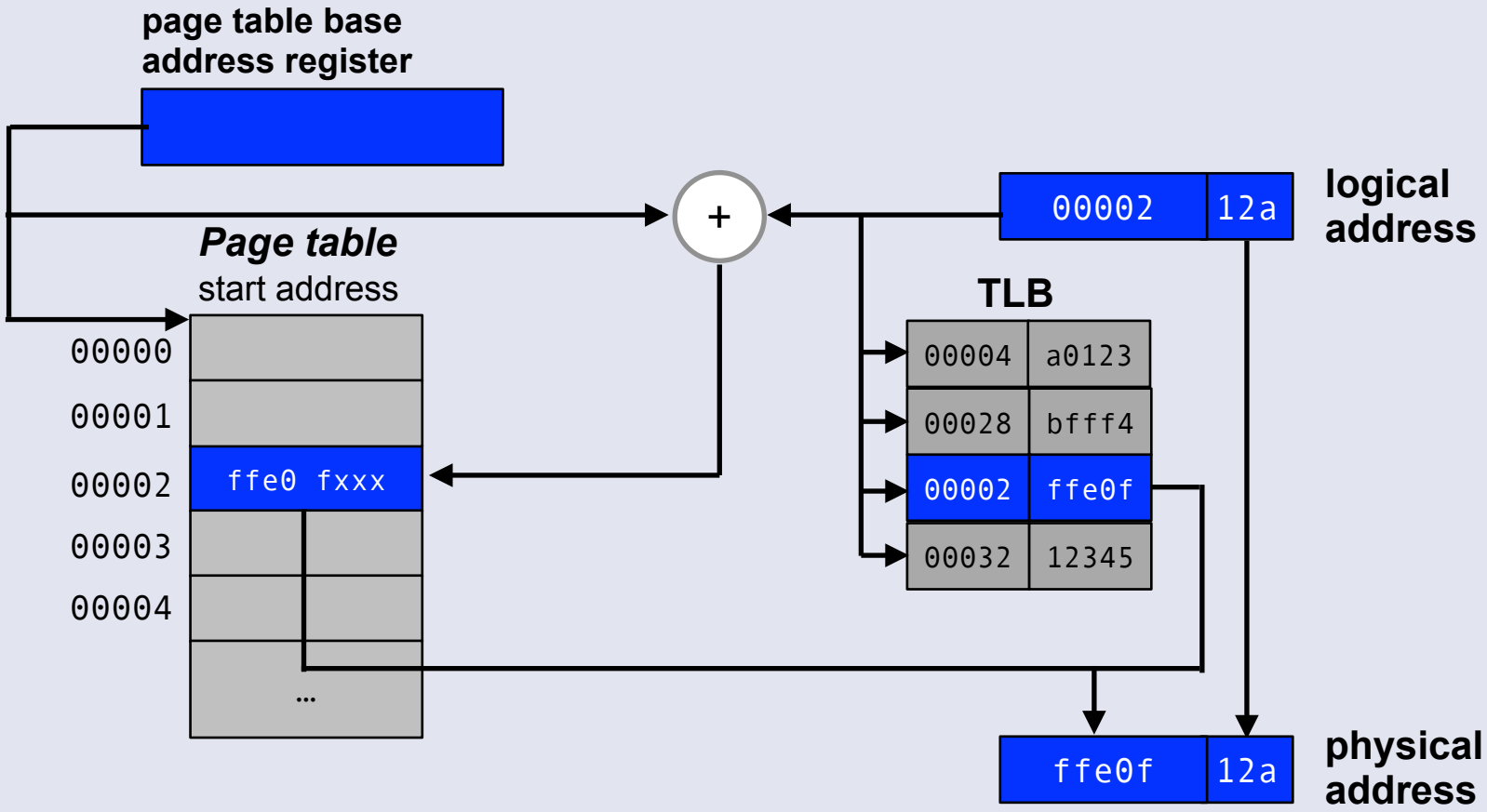
- Using virtual memory address translation requires **three main memory accesses**!

- Same idea as with regular slow memory access: use cache!
- The MMU uses a special cache on the CPU chip: the **Translation Lookaside Buffer** (TLB)
- Caches commonly (most often? most recently?) used PTEs

Translation lookaside buffer (TLB)

Universität Bamberg

- Fast cache which is consulted before a (possible) lookup in the page table:



Translation lookaside buffer (TLB)

- Fast access to page address mapping, the information is contained in the (fully associative) TLB memory
  - no implicit page accesses required
- TLB has to be ***flushed*** when the OS switches context
  - *Address space identifiers* allow to have mappings for different processes in the TLB at the same time (these don't have to be used)
- If a translation is not contained in the TLB, the related access information is entered into the TLB
  - An old TLB entry has to be selected to be replaced by the new one

- TLB sizes:
  - Intel Core i7: 512 entries, page size 4 kB
  - UltraSPARC T2: data TLB = 128, Code TLB = 64, page size 8 kB
  - Allwinner/T-Head C906 RISC-V: 128-512 entries (D1 chip: 256 joint entries)
  - Larger TLBs are currently not implementable due to timing and cost considerations

Virtual memory on RISC-V

- The modes of the virtual address translation are titled "sv**xx**"
  - **xx** is a number giving the amount of bits to be translated
- 32-bit RISC-V systems have a sv32 virtual memory system
  - two translation levels with 10 bit (1024 entries) each, 12 bit (4 kB) page size

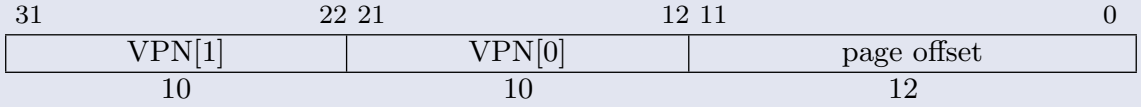


Figure 4.12: Sv32 virtual address.

- 64-bit RISC-V has several options (qemu supports all three):
  - sv39, 48, and 57 are currently defined: three translation levels

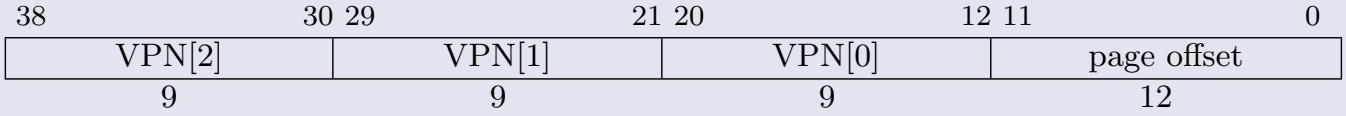


Figure 4.15: Sv39 virtual address.

Page tables on RISC-V

The format of page table entries is defined by the specific sv mode

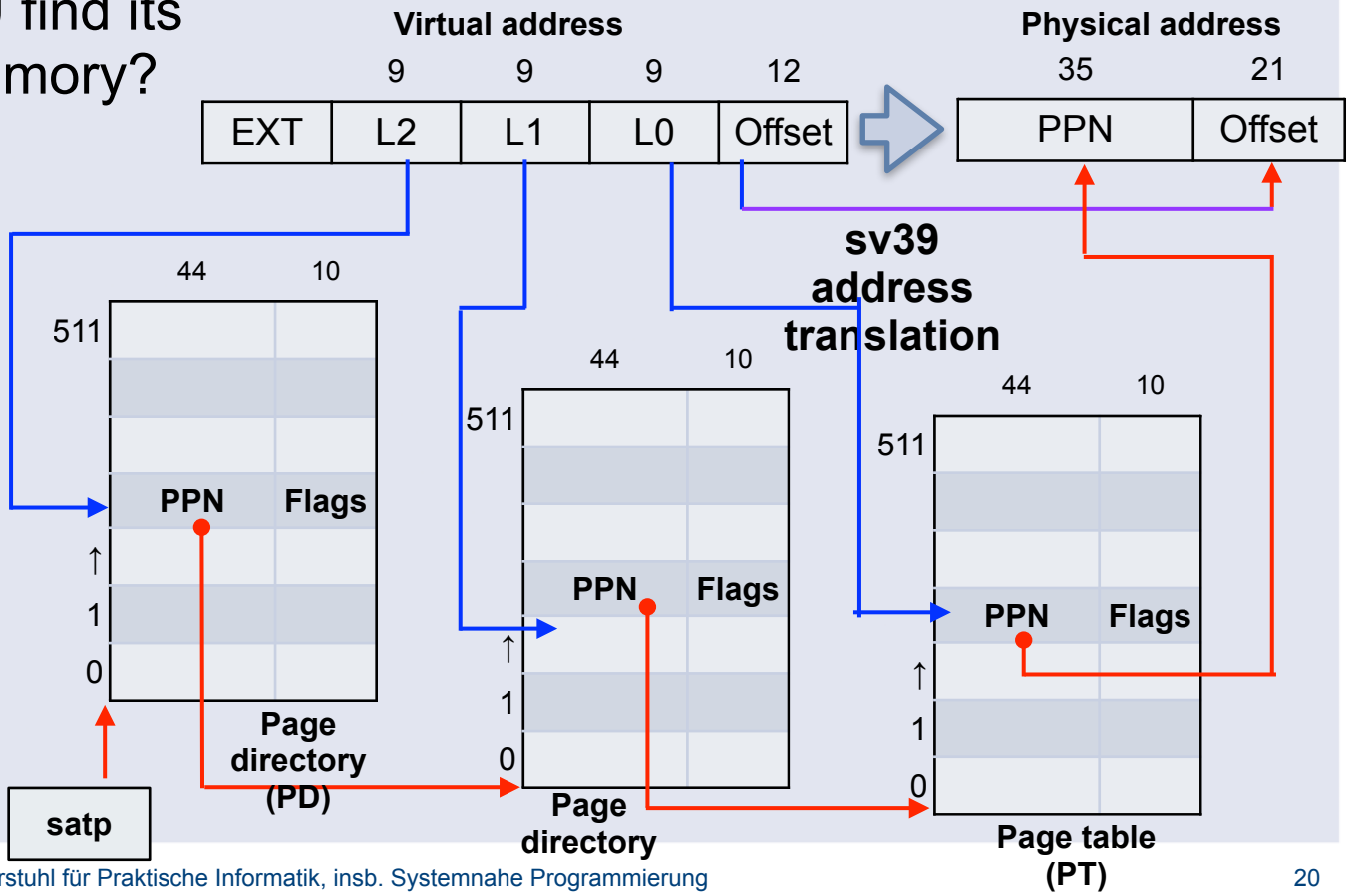
63	62 61	60 54	53 28	27 19	18 10	9 8	7	6	5	4	3	2	1	0
N	PBMT	*Reserved*	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V
1	2	7	26	9	9	2	1	1	1	1	1	1	1	1

Figure 4.21: Sv39 page table entry.

- MMU replaces bits 12–38 of virt. address with ***physical page nr.*** (**PPN**)
  - 44 physical page number bits available
  - *larger physical than logical address space possible!*
- Meta information:
  - **V**(alid): is this entry valid? *(this is the only bit set for page directories!)*
  - Protection bits: **R**(ead), **W**(rite), (e)**X**(ecute) permissions for the page
  - **U**(ser): translation for U-mode only, not for S-mode
  - **G**(lobal) bit: entry valid in all address spaces
  - **A**(ccessed): virtual page has been used since the A bit was cleared
  - **D**(irty): virtual page has been written since the D bit was cleared

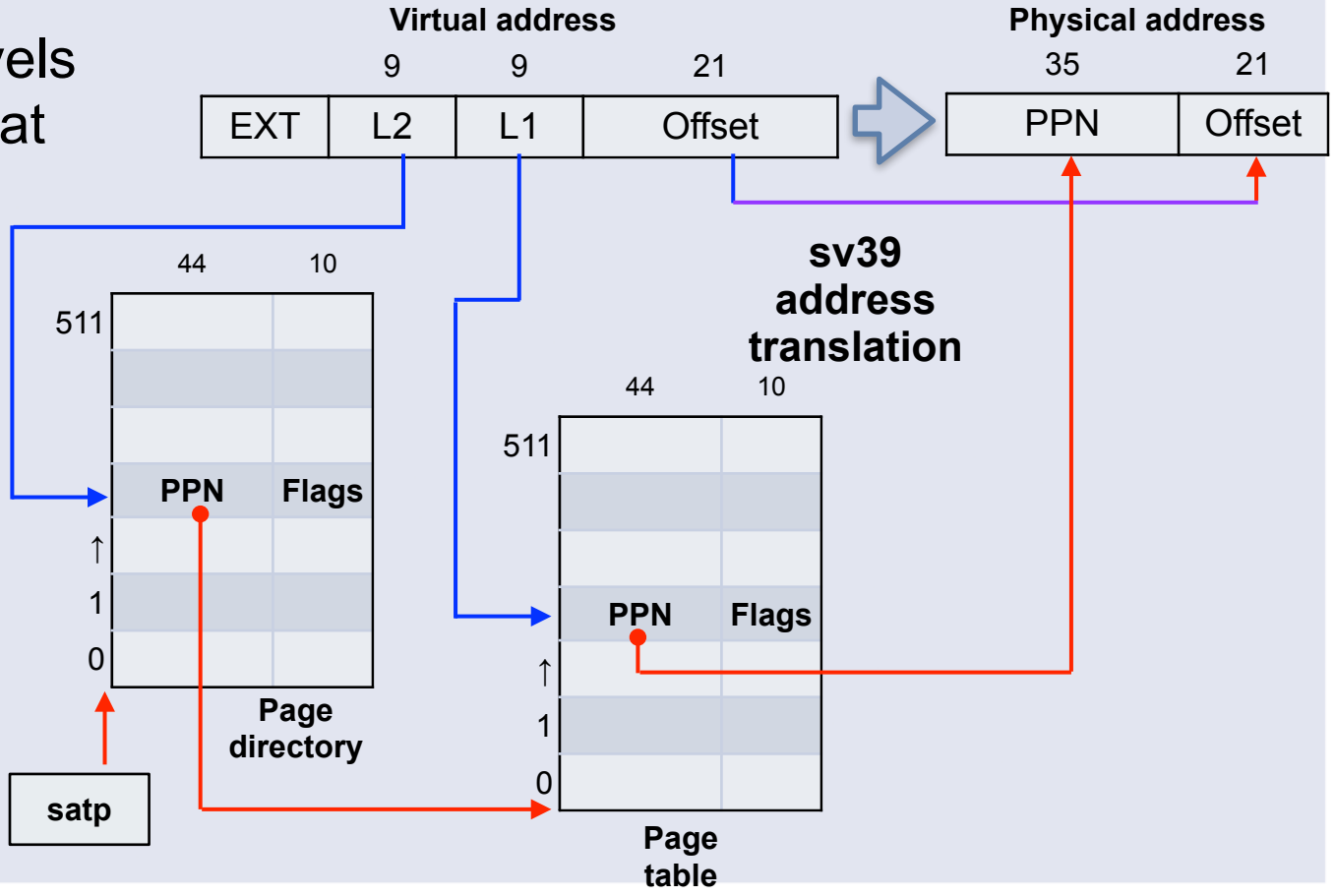
Virtual memory on RISC-V

- Virtual address translation only takes place in S- and U-mode
  - Our OS runs in M-mode: kernel can't use address translation
    - "Real" RISC-V OSes usually run in S-mode
- How can the CPU find its page tables in memory?
  - `satp` CSR supervisor address translation pointer
  - Points to start of **phys. page** with top-level page table



Large Pages for RISC-V Virtual memory

- With three levels of lookup, pages are 2^{12}=4096 byte in size
- We can also create **large pages** by stopping the translation process early
- If the **flags** on levels L2 or L1 indicate at least one bit of R, W, or X set, then the **page table walk** is terminated at this level
- **Stop at L1:** page size = 4096*512 = 2MB



Virtual memory on RISC-V

Universität Bamberg

- Virtual address translation is disabled by default
  - Enabled by writing to `satp` CSR

Figure 4.12: RV64 Supervisor address translation and protection register `satp`, for MODE values Sv39 and Sv48.

- Bits 0–43: Physical page number of top-level page table
- Bits 44–50: Address space ID (which process?)
- Bits 60–63: MMU mode

RV64		
Value	Name	Description
0	Bare	No translation or protection.
1–7	—	*Reserved*
8	Sv39	Page-based 39-bit virtual addressing.
9	Sv48	Page-based 48-bit virtual addressing.
10	*Sv57*	*Reserved for page-based 57-bit virtual addressing.*
11	*Sv64*	*Reserved for page-based 64-bit virtual addressing.*
12–15	—	*Reserved*

MMU mode bits

- Ensure all translations are **synchronized**
  - Use `sfence.vma` instruction after changing value in `satp`

RISC-V translation process – 3 stages

Universität Bamberg

- Virtual address `0x7d_beef_cafe`: exactly 39 bits
- binary:
  `0b0111_1101_1011_1110_1110_1111_1100_1010_1111_1110`

VPN[2]	VPN[1]	VPN[0]	Offset
502	503	252	

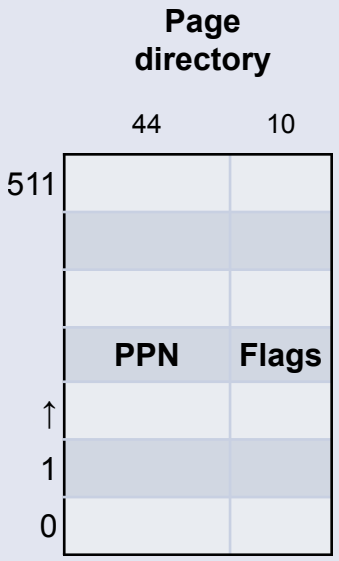
1. Read `satp` register and find the top of level 2's page table (PPN << 12).
2. Add `offset * size`,
   offset = VPN[2] = `0b1_1111_0110` = 502 * 8 = `satp` + 4016
3. Read this entry
4. If `V(valid)=0`, page fault
5. If this entry's `R`, `W`, or `X` bits are 1, this is a leaf, otherwise it is a branch
6. The `PPN[2] | PPN[1] | PPN[0]` address shows where in physical memory the next page table is located
7. Repeat at #2 until a leaf is found
8. Leaf: `PPN[2]`, `PPN[1]`, and `PPN[0]` give physical page number

Virtual memory on RISC-V

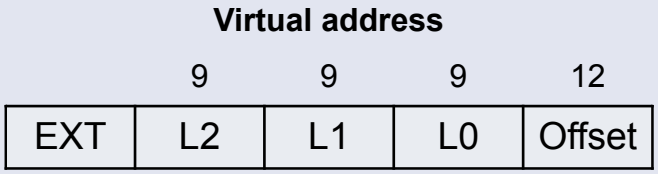
- What happens if an address cannot be translated?
  - No entry for logical page number found
  - Entry found, but no valid bit
- What happens if a PTE does not allow the attempted access?
  - e.g. trying to write to a page which has only the R bit set
- Result – **exception called "page fault":**
  instruction, load, and store exceptions can occur
  - An instruction page fault occurs during the instruction fetch
    - page entry is not valid or does not have the X bit set to 1
  - Page faults are trapped by the CPU, `mcause`/`scause` values:
    - 12: instruction page fault, 13: load page fault,
      15: store page fault
- **Important:** *PMP configuration is evaluated before VM translation!*

Page tables in memory

- Page tables for each process have to be stored in memory
  - Accessed using **physical addresses**
- 9 Bits per level → 2^9 = 512 entries per directory/table
  - Each entry uses 8 bytes
    → 512*8 = 4096 bytes per directory/table
  - This is equal to the page size: *coincidence*? 😀

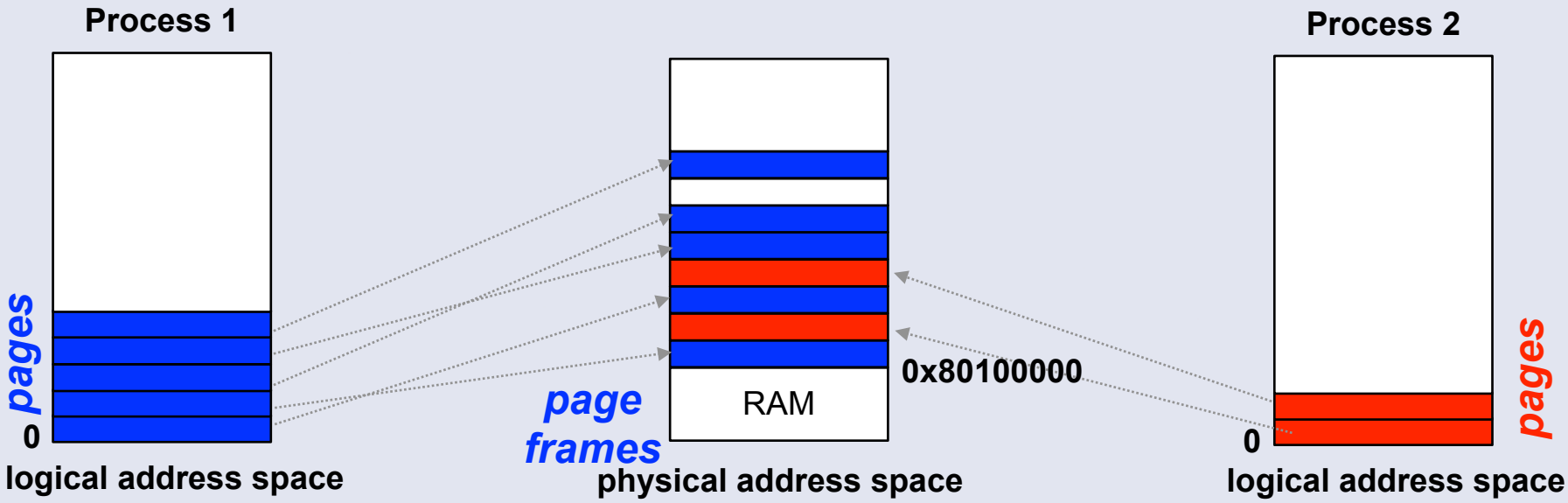


- Simplest case: **single-level page table**
  - Only one 4096 byte page needed
  - Page size = $2^{(9+9+12)}$ = 2^{30} bytes = 1 GB!
- Two-level page tables are more practical (2 MB pages)
  - …for us right now at least
  - Require two 4096 byte pages



Paging and multitasking

- Each process has its **own logical address space**
  - E.g. starting at virtual address 0 for both
- Processes need separate page tables = VA→PA mappings
  - Switch mapping with context switch – don't forget to flush the TCB! (`sfence.vma` instruction)
  - Store separate `satp` CSR value for each process



Simple two-level page table setup

- Let's keep it simple for now
  - **One** 2 MB page for each process only
  - Accordingly, we need a 2 MB **page frame** for each process now

- New physical address space layout:
  - Kernel is located at 0x8000_0000 → 0x801F_FFFF
  - Process 1 is loaded at 0x8020_0000 → 0x803F_FFFF
  - Process 2 is loaded at 0x8040_0000 → 0x805F_FFFF
  - …and so on

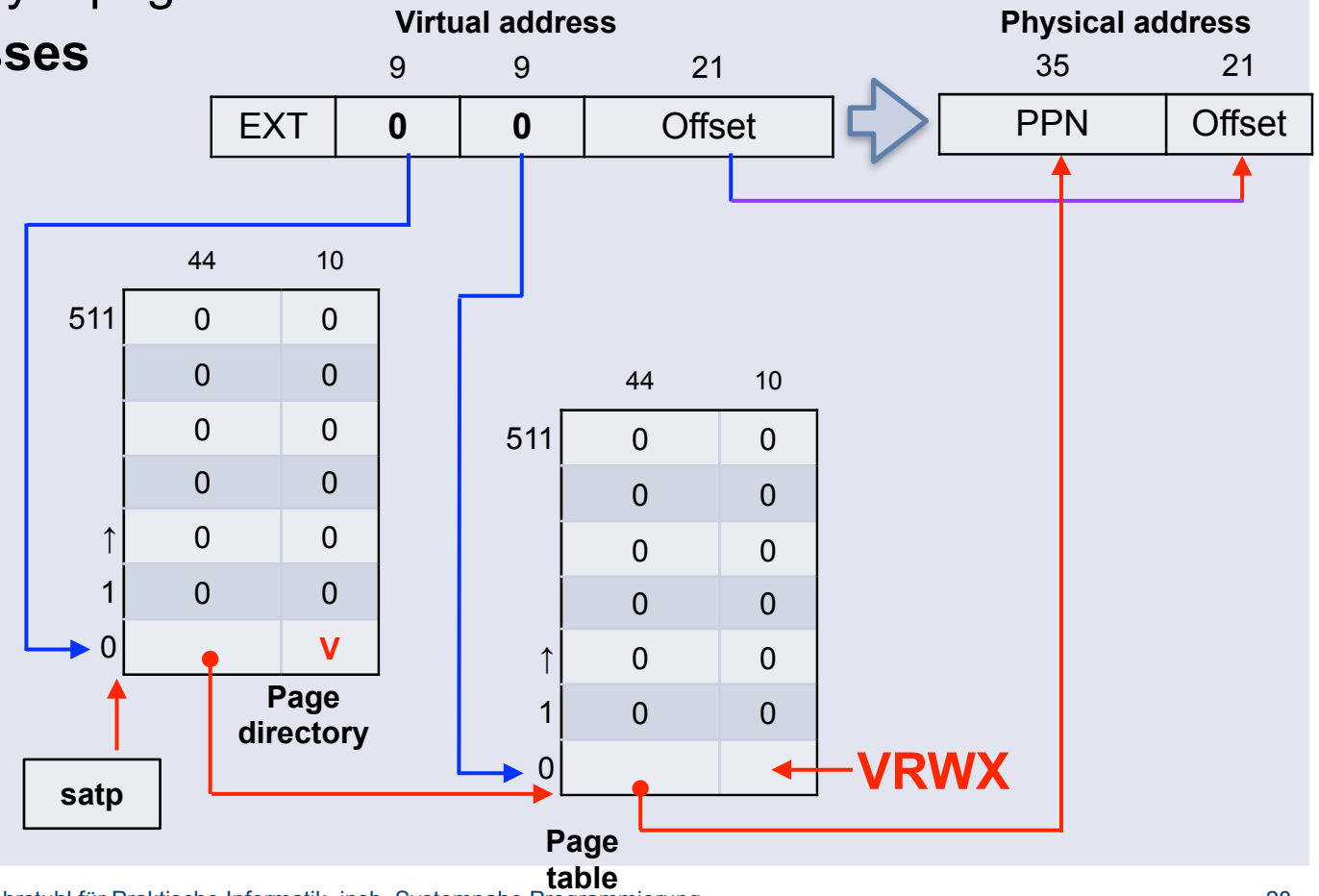
- The **physical address** of the only page frame for process **n** is:

  PA = 0x8000_0000 + **n** * 0x0020_0000 with n ∈ { 1,…,max }

Simple two-level page table setup

Universität Bamberg

- With one 2 MB page frame per process:
  - the process page table needs only a single directory and table
  - so, two 4096 byte pages
- All **virtual addresses** are within the range **[0, 2GB[**
- Only entry 0 is needed then in both the page directory and the page table!
- PD and PT can use contiguous memory page frames!



Virtual memory and multitasking

- **Benefit** of virtual memory for our compilation process:
  - All processes have the **illusion** of using memory starting at 0
  - With a **single** 2 MB entry for a page allocated per process, the **virtual address space** for each process runs from 0x0000_0000 … 0x001F_FFFF
  - Thus, we can now **use the same linker script** for all processes that links their code and data segments starting at address 0
  - …and we can link each user space program separately!
- We still need to be able to **load programs** → lab session
- Enabled for qemu with the "loader" command line option: https://qemu-project.gitlab.io/qemu/system/generic-loader.html

```
qemu-system-riscv64 ……
            -device loader,addr=<addr>,file=<data>
```

Conclusion

- Virtual memory is useful for better isolation between processes and fine-grained translation
  - Page granularity used to reduce translation overhead
- Translation by MMU using page tables (+TLB cache)
  - Logical addresses generated by the CPU are translated to physical addresses
  - Processes can have identical virtual addresses (e.g. all processes start at virtual address 0, which is mapped to separate pages for each process)
- To save memory, page tables are *sparse* and *hierarchical*
  - RISC-V: two (32 bit) or three-level (64-bit) address translation
    - Different modes (sv39, 48, 57) available for 64-bit
  - Address of root level of page table in `satp` CSR

References

1. U. Drepper, *What Every Programmer Should Know About Memory*, RedHat Inc., 2007
2. Palmer Dabbelt, *Paging and the MMU in the RISC-V Linux Kernel*, https://www.sifive.com/blog/all-aboard-part-9-paging-and-mmu-in-risc-v-linux-kernel
3. Stephen Marz, *The Adventures of OS: Memory Management Unit*, https://osblog.stephenmarz.com/ch3.2.html