DEPARTMENT OF
NETWORKED SYSTEMS
AND SERVICES

COMPUTER ARCHITECTURES

Virtual Memory

Gábor Horváth, ghorvath@hit.bme.hu

Budapest,
2023. 04. 03.

MŰEGYETEM 1782

VIRTUAL MEMORY

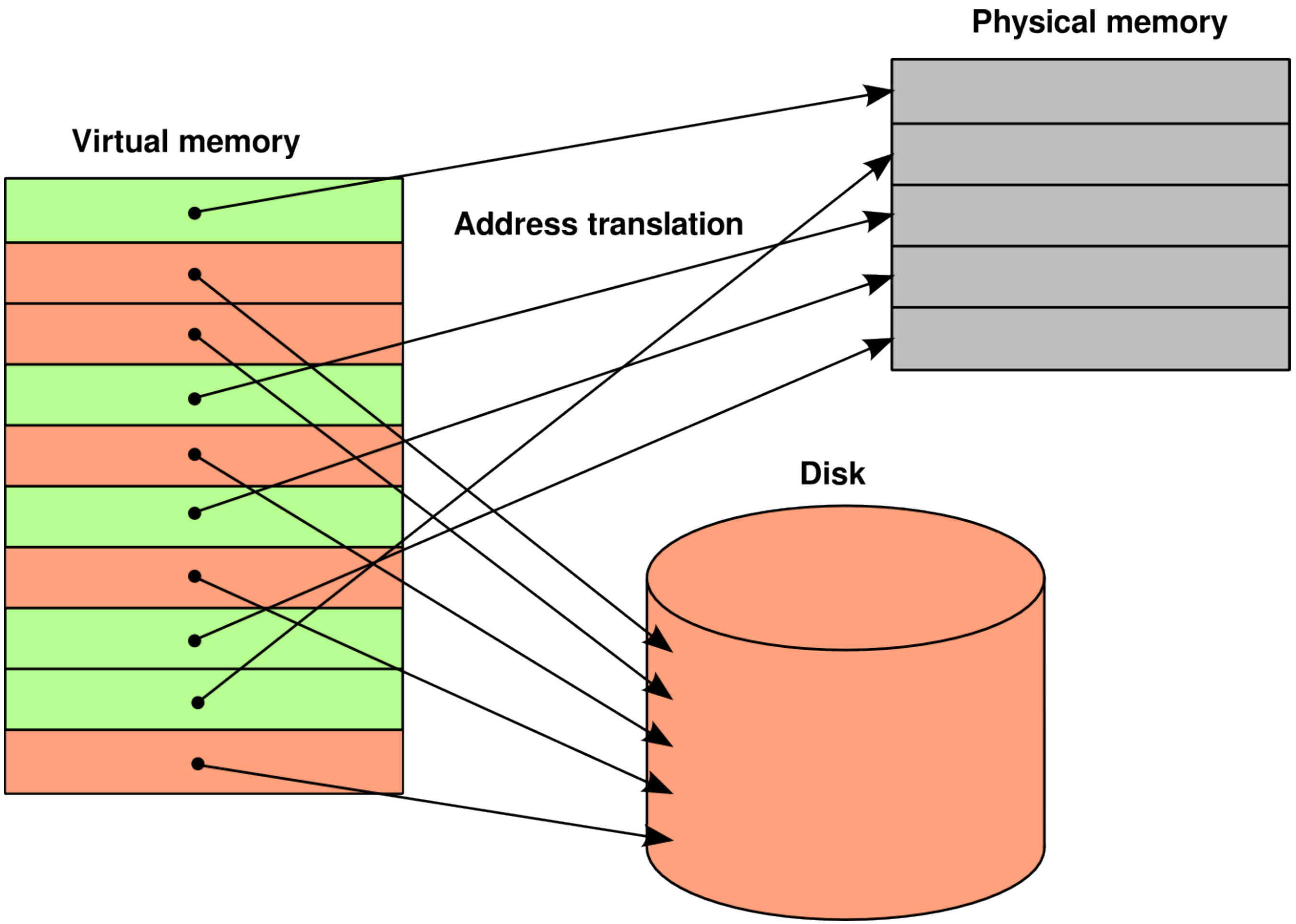
- Motivation:
  - Multi-tasking operating systems
    - Tasks come and go continuously
    - The memory share available for each task is varying
  - There are cheap 64 bit CPUs available
    - For instance, AMD64 is capable of using 256 TB RAM
    - We can not plug in that much RAM
- The CPU offers all its address space to the programs
  - From address 0 to the top of the address space
  - Memory size can never be the issue in this case
- This „virtual” (offered) amount of memory needs to be covered by physical memory

  → **Virtual memory**

VIRTUAL MEMORY

- Programs use: ***Virtual addresses***
- On the address lines of the CPU and the bus we have: **Physical addresses**
- The mapping between virtual and physical is called: ***Address Translation***
  - Done by: ***MMU*** (Memory management unit)
    - Built into the CPU in most cases
  - Unit of address translation is called:
    - ***Page,*** if it has fixed size, or
    - ***Segment,*** if it has variable size
- If not all pages fit into the memory
  → they are written to the disk (swapping)

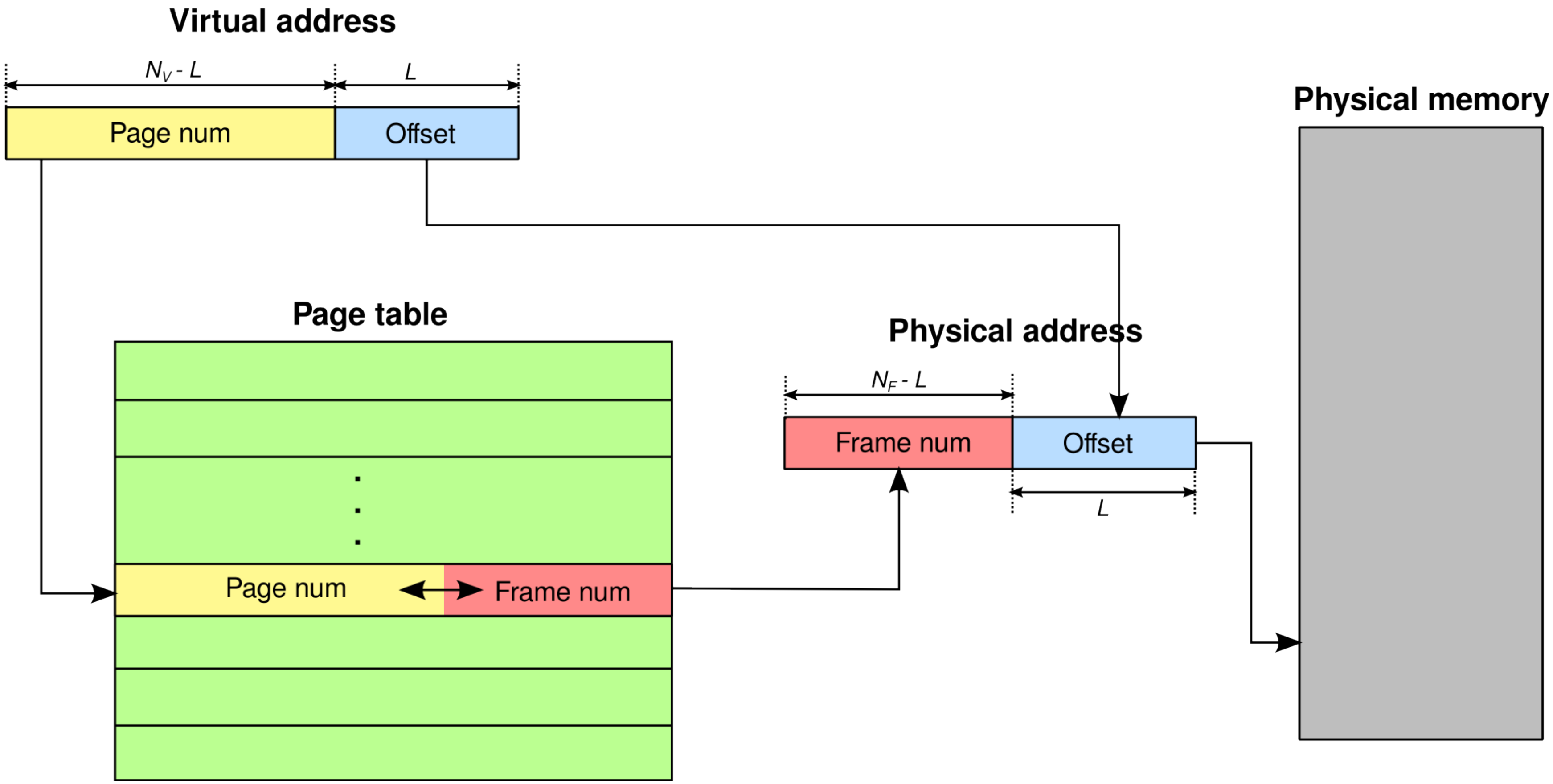
VIRTUAL MEMORY



ADDRESS TRANSLATION

- The virtual address space is partitioned to fixed size ***pages***
- The physical address space is partitioned to ***frames***, having the same size as pages
- Sizes:
  - Size of pages = 2^L
  - Lower L bits of addresses: offset from the start of the page
  - Upper bits:
    - In case of virtual memory: Page identifier
    - In case of physical memory: Frame identifier
- Page ↔ Frame mapping is stored in the ***page table***
- Contents of the page table:
  - The page number
  - The frame number
  - Protection information (rights to read/write)
  - Control bits:
    - Valid (means: the given page is in the physical memory)
    - Dirty (means: the page has been changed, since it was read from disk)

ADDRESS TRANSLATION



- Remark: this figure is „theoretical”. The implementation of the mapping between page numbers and frame numbers will be explained later.

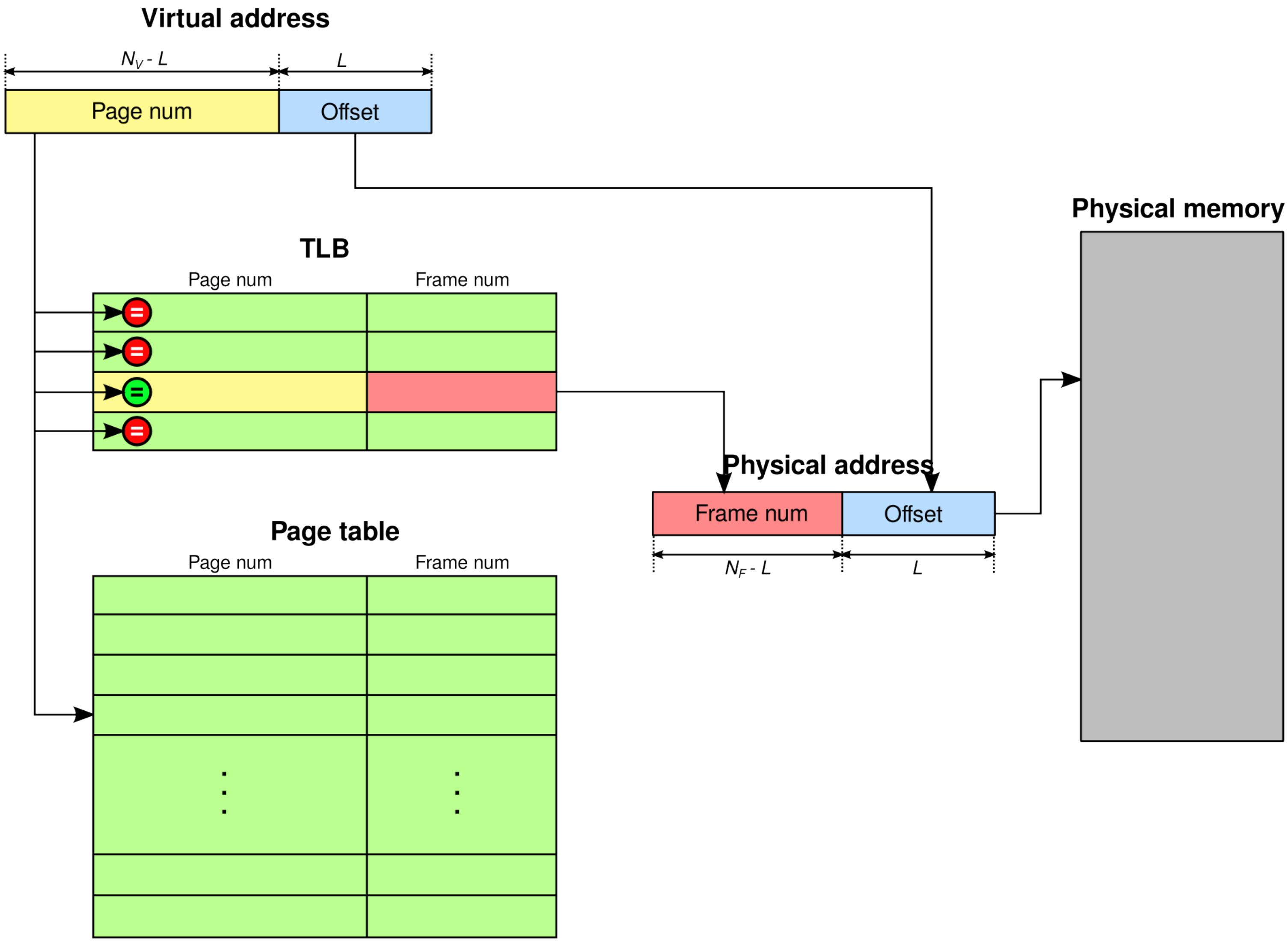
ADDRESS TRANSLATION

- Assume the program wants to access the memory
- It uses virtual addresses
- The upper bits are the page number
- The page table is looked up to find the frame on which the page is stored
- If the page is in the memory according to the page table:
  - Knowing the frame number the physical address is assembled
    - The upper bits are the frame number
    - The lower L bits are the offset
      - Copied from the lower L bits of the virtual address
- If the page is not in the memory (**page fault**):
  - The CPU asks the operating system to load the page from the disk
  - The operating system throws out a rarely used page from the memory
  - ...and puts the requested page into its place
  - The operating system then updates the page table
- The CPU puts the physical address onto its address lines to serve the memory read/write request

TLB

- Each memory access initiated by the program consists of:
  - Address translation: obtain physical address from the virtual one
  - The actual memory operation on the physical address
- But the page table is stored in the memory as well!
- **1 memory access of the program**
  → costs 2 memory accesses in the reality !!
- But the memory is slow
  - And now we need to access it twice as many times!
- Remedy: locality
- ***TLB***: Translation Lookaside Buffer
  - A special cache in the CPU that stores the virtual ↔ physical mapping of pages that are used most of the time
  - When doing address translation the CPU looks into its TLB first

ADDRESS TRANSLATION WITH TLB



FEATURE OF THE TLB

- **TLB coverge:** the amount of memory covered by the pages that have translation information stored in the TLB
  - The larger it is the less we need to access the page table residing in the slow memory
- Implementation of the TLB: **content addressable memory**

  → It has a large surface and consumes a lot of energy

  → It is small :(

TLB size:	16 – 512 entries
TLB hit time:	0.5 – 1 clock cycles
Translation time when TLB misses:	10 – 100 clock cycles
TLB miss rate:	0.01% – 1%

PAGE TABLE IMPLEMENTATIONS

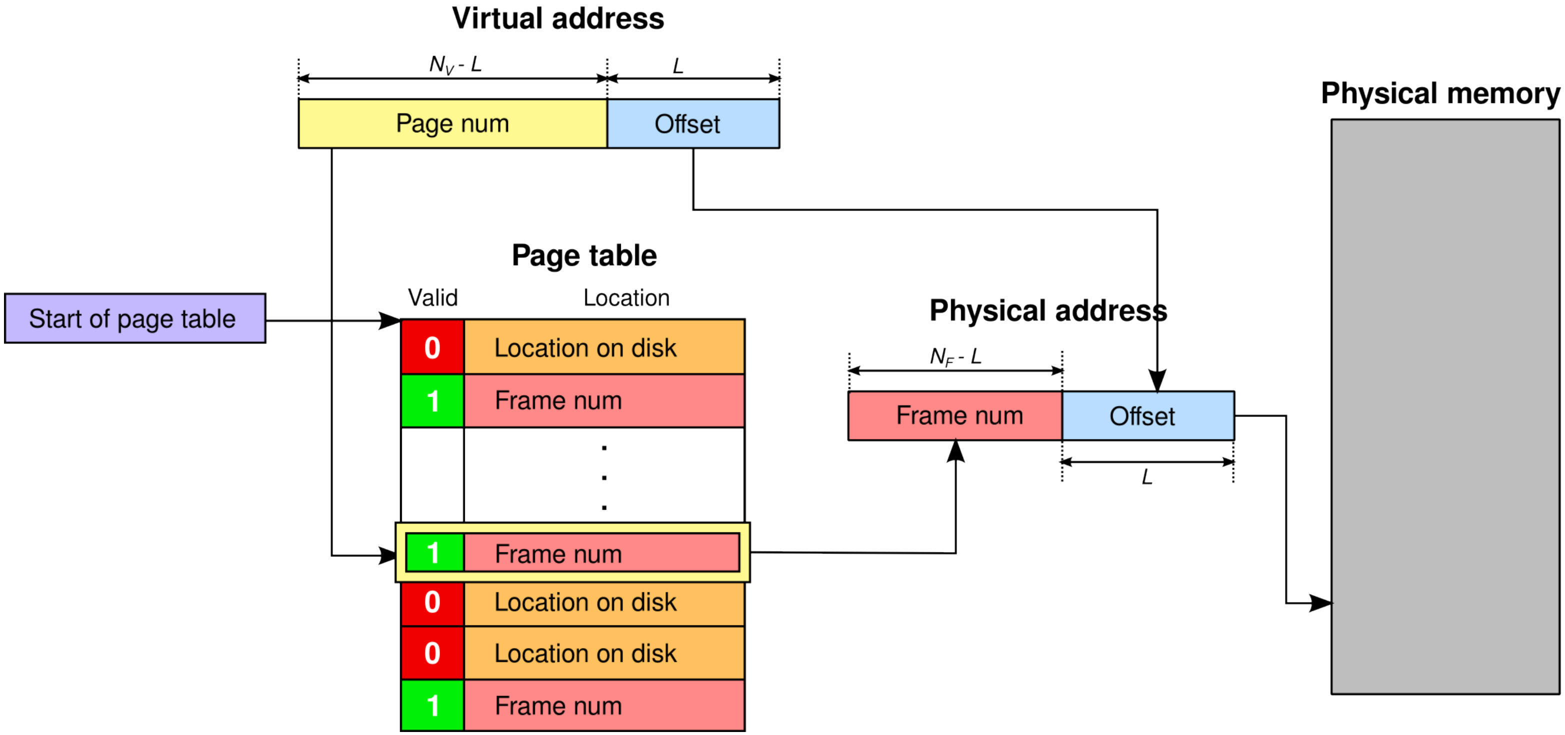
- Goal:
  - Perform address translation as fast as possible

    → we are typically looking for the frame number based on the page number

    → while doing the lookup we want to touch the slow memory as few as possible
  - The page table should consume as few memory as possible
- Solution:
  - Special data structures
    - Single-level page table
    - Multi-level (hierarchical) page table
    - Inverse page table
    - Etc.

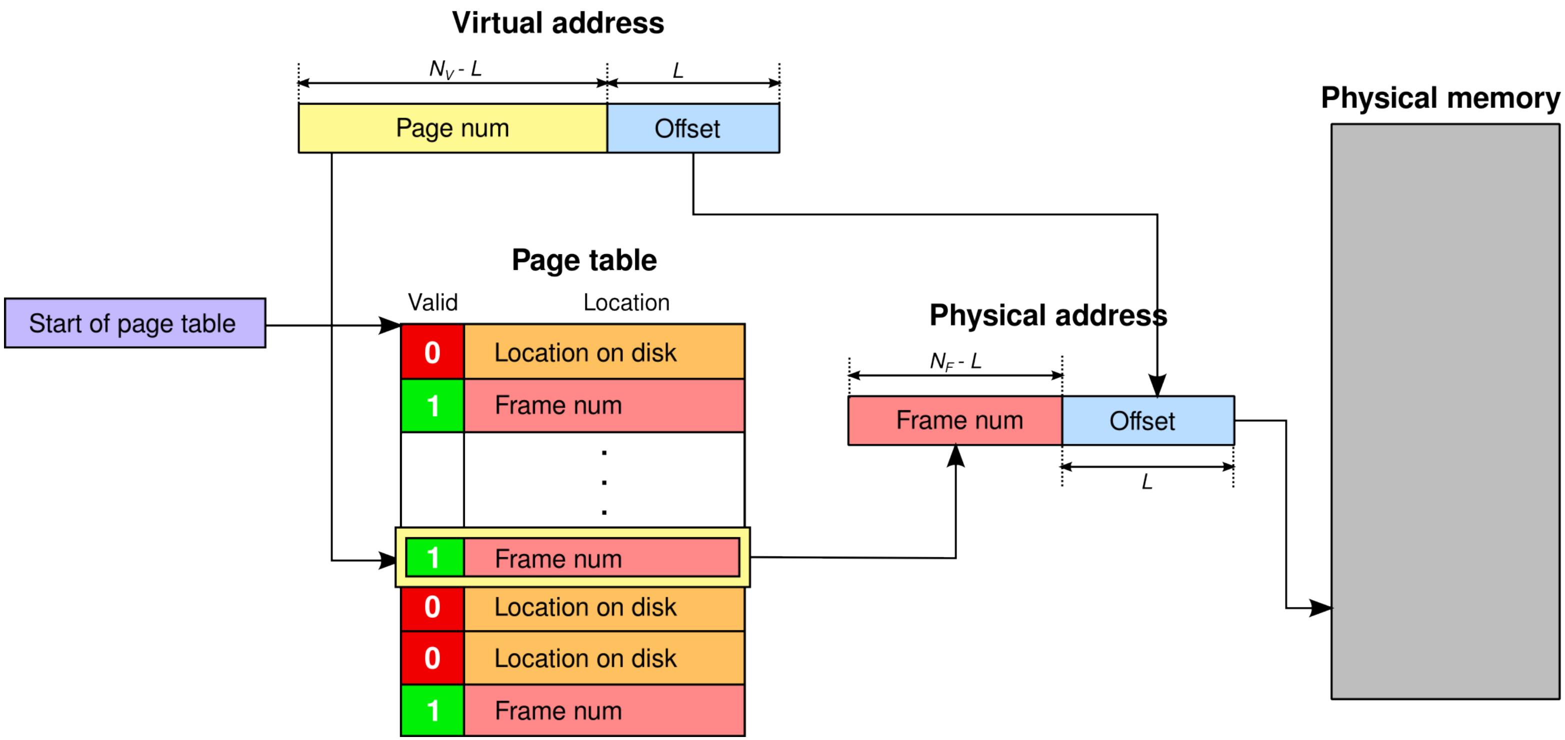
SINGLE-LEVEL PAGE TABLES

- A single-level page table is simply an array of page table entries
- Entry i.:
  - Valid bit: is the page in the memory, or is it in the disk?
  - If valid: the frame where page i. is stored in the physical memory
  - If not valid: where on the disk is the page stored



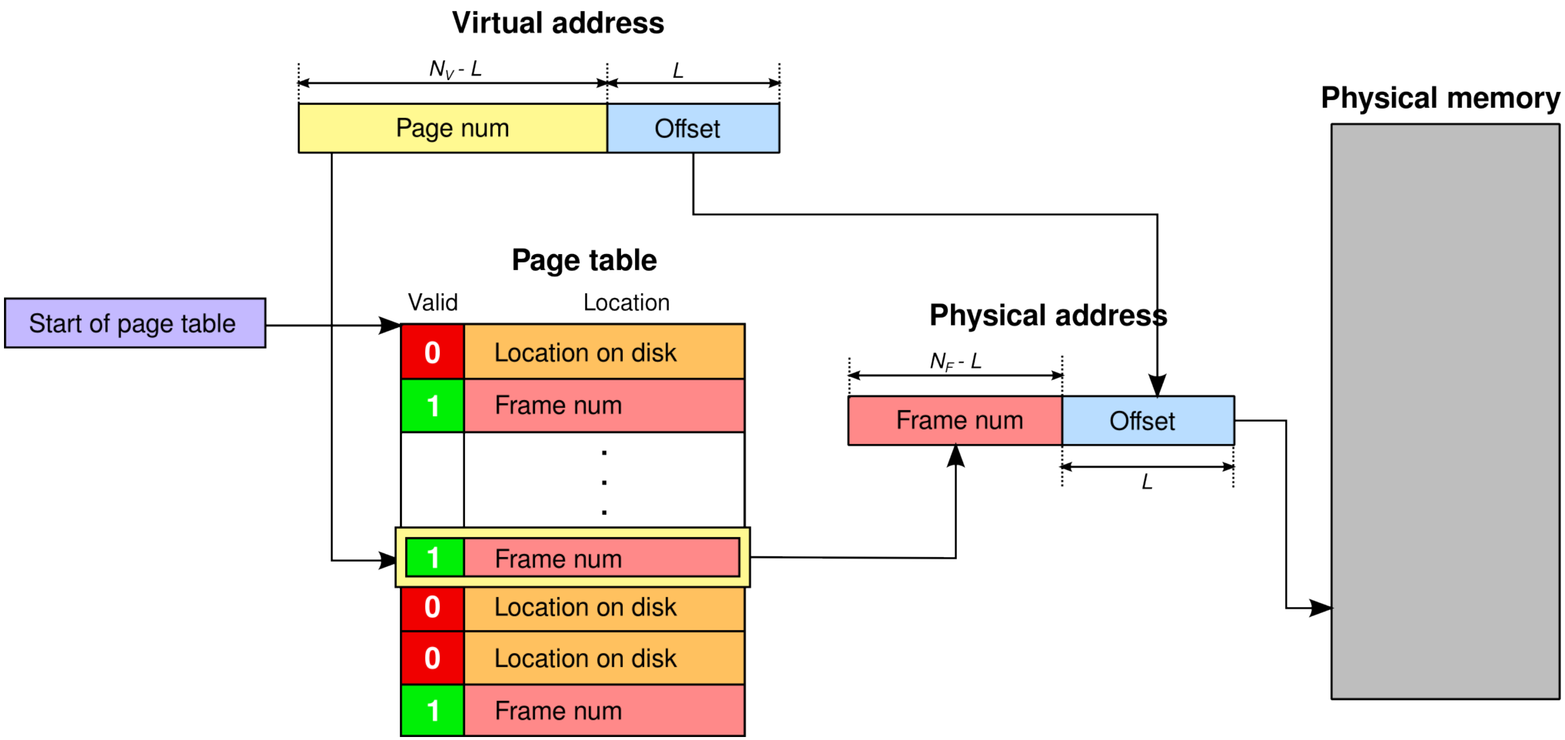
SINGLE-LEVEL PAGE TABLES

- Lookup procedure: fast
- Finding the entry associated with page i. needs exactly 1 memory operation



SINGLE-LEVEL PAGE TABLES

- Size of the page table entries: small
  - The page identifier is not stored → it is the index of the array
  - The location on the disk can be stored in case of invalid entries



SINGLE-LEVEL PAGE TABLES

- Fast, needs a single memory access during lookup, the entries are small, ...

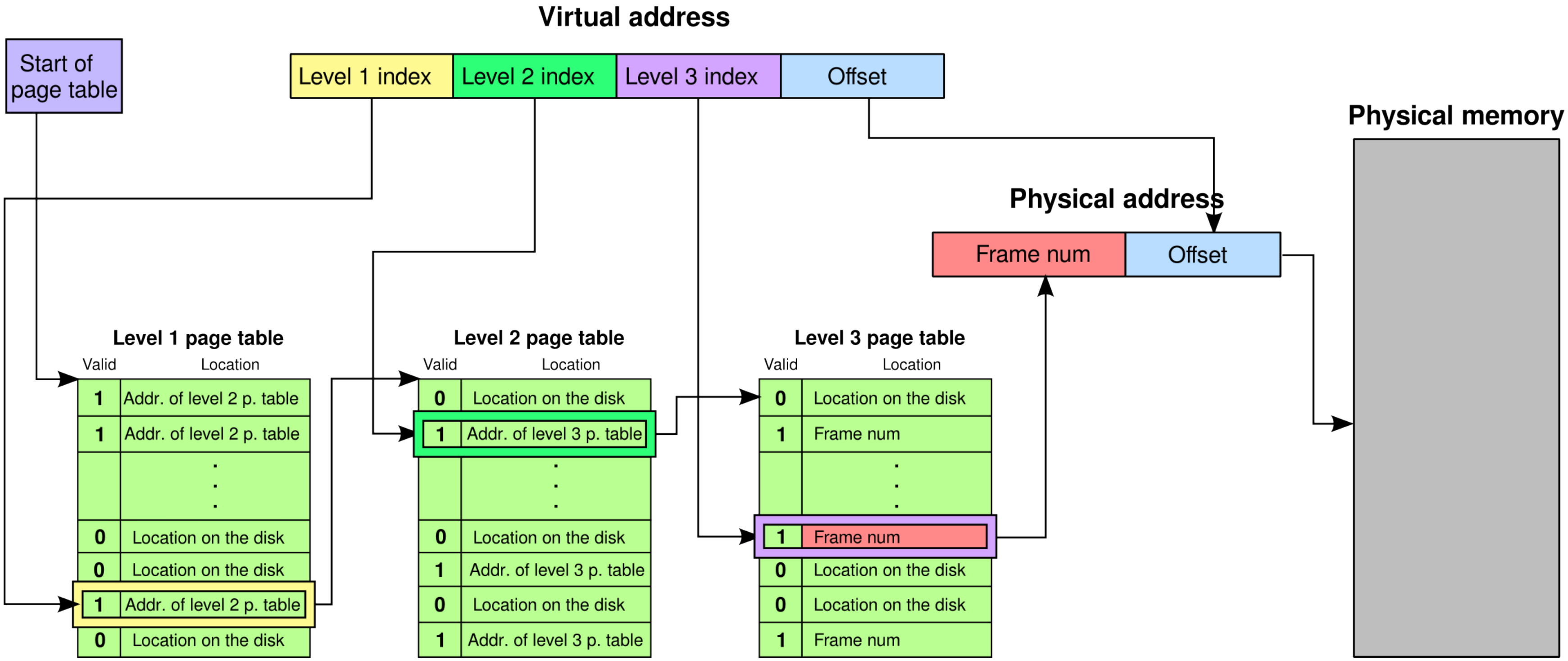
  ...isnt it the ideal way to store the page table?

Problem: the entire page table needs to be present in the memory all of the time!

- Quick calculations:
  - With 32 bit addresses and 4 kB pages:
    - Page size: 12 bit, number of pages: 32-12 = 20 bit, 1 mega-page
    - 4 byte is enough to store 1 entry
    - Size of the page table: 4 MB
  - With 64 bit addresses and 4 kB pages:
    - Page size: 12 bit, number of pages: 64-12 = 52 bit, 2^{52} pages
    - 8 byte is enough to store 1 entry
    - Size of the page table: $8 * 2^{52} = 32 * 2^{50}$ = 32 PB !!!

HIERARCHICAL PAGE TABLES

- Idea: let us cut the page table into pages as well
- The locations of pages storing page table entries are stored in an other page. The location of these pages are stored on a third page, etc.



Level 1 page table

Valid	Location
1	Addr. of level 2 p. table
1	Addr. of level 2 p. table
	⋮
0	Location on the disk
0	Location on the disk
1	Addr. of level 2 p. table
0	Location on the disk

Level 2 page table

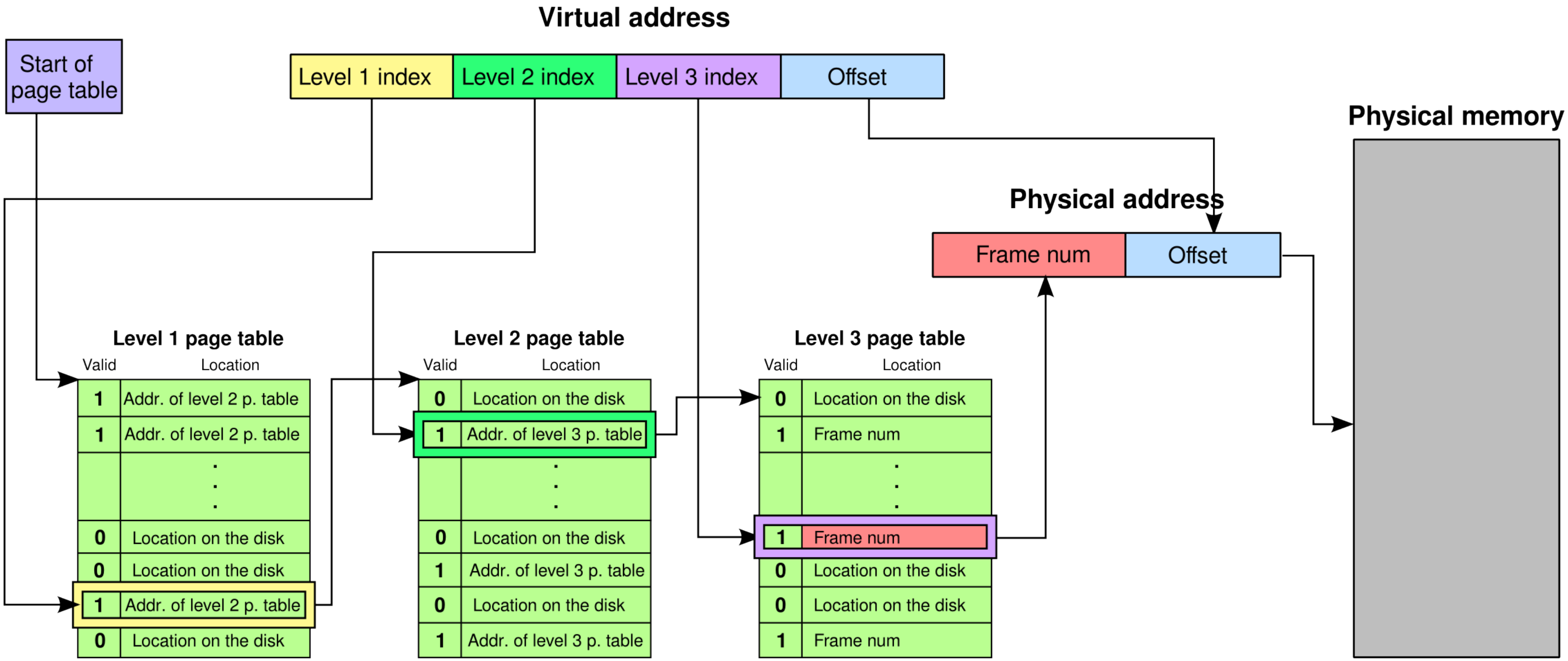
Valid	Location
0	Location on the disk
1	Addr. of level 3 p. table
	⋮
0	Location on the disk
1	Addr. of level 3 p. table
0	Location on the disk
1	Addr. of level 3 p. table

Level 3 page table

Valid	Location
0	Location on the disk
1	Frame num
	⋮
1	Frame num
0	Location on the disk
0	Location on the disk
1	Frame num

HIERARCHICAL PAGE TABLES

- Only those parts of the page table are stored in the memory that are in use indeed
- The memory is touched several times during the traversal → slow!



Level 1 page table

Valid	Location
1	Addr. of level 2 p. table
1	Addr. of level 2 p. table
	⋮
0	Location on the disk
0	Location on the disk
1	Addr. of level 2 p. table
0	Location on the disk

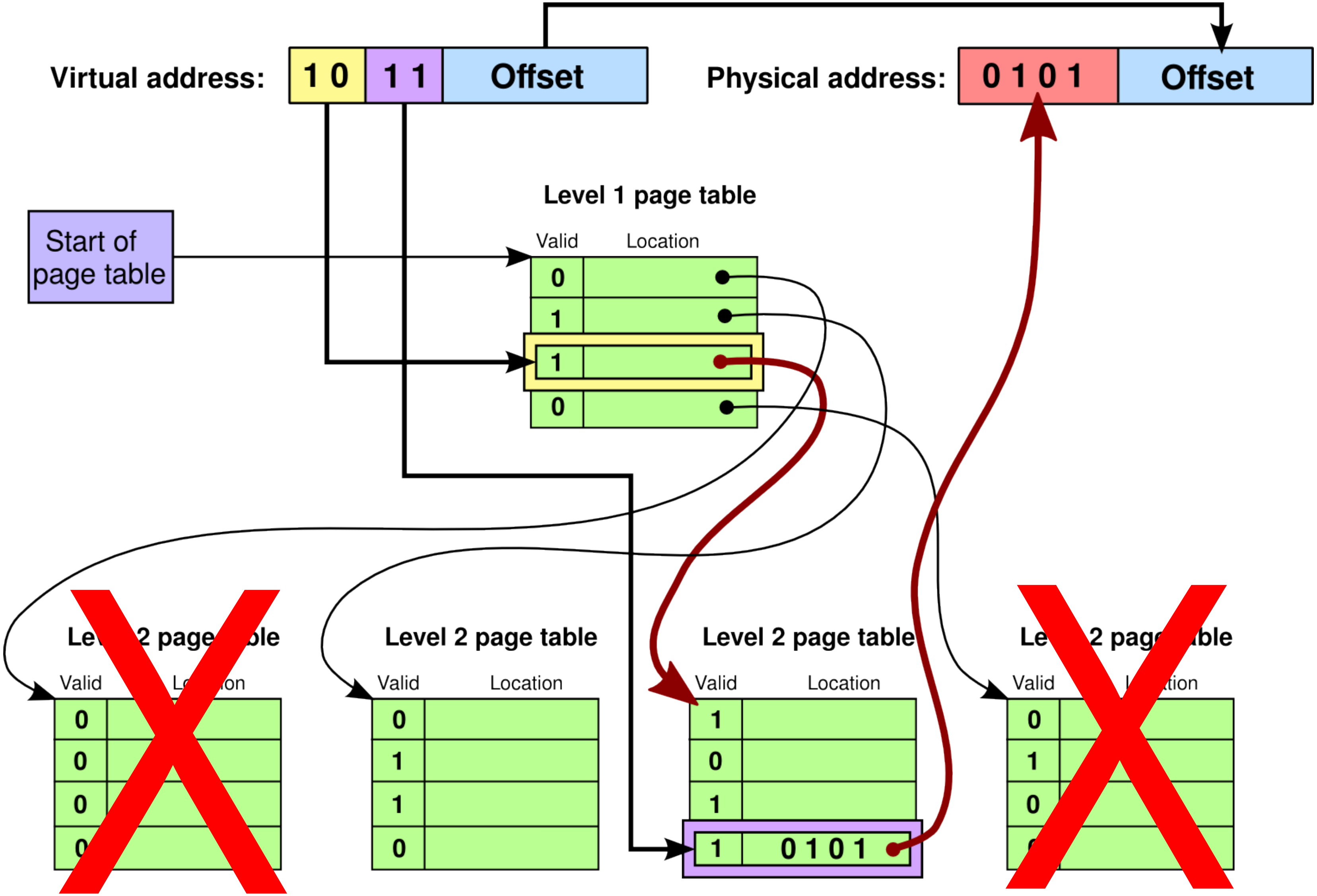
Level 2 page table

Valid	Location
0	Location on the disk
1	Addr. of level 3 p. table
	⋮
0	Location on the disk
1	Addr. of level 3 p. table
0	Location on the disk
1	Addr. of level 3 p. table

Level 3 page table

Valid	Location
0	Location on the disk
1	Frame num
	⋮
1	Frame num
0	Location on the disk
0	Location on the disk
1	Frame num

TWO-LEVEL HIERARCHICAL PAGE TABLE



HIERARCHICAL PAGE TABLES

- Quick calculation:
  - With 32 bit addresses and 4 kB pages:
    - Page size: 12 bit, 4 byte is enough to store 1 entry
    - One page can store 1024 entries → we need 10 bits to index it
    - There are 1024 pages storing page table entries, and a single page that contains pointers to these pages
    - Partitioning a 32 bit address = 10 + 10 + 12 bit (first level index, second level index, offset)

      → Two-level page table is enough
  - With 64 bit addresses and 4 kB pages:
    - Page size: 12 bit, 8 byte is enough to store 1 entry
    - One page can store 512 entries → we need 9 bits to index it
    - Partitioning a 64 bit address = 7 + 9 + 9 + 9 + 9 + 9 + 12 bit

      → A six-level page table is required!

      6 memory operations are needed for each address translation !!!
- X86, ARM architectures are using hierarchical page tables

HIERARCHICAL PAGE TABLES

- 6 levels are way too many!
- Practical mitigation: usage of a smaller virtual address range
- Example:

```
cat /proc/cpuinfo | grep -E "sizes|name" | head -n 2
model name      : Intel(R) Xeon(R) CPU     E5440  @ 2.83GHz
address sizes   : 38 bits physical, 48 bits virtual
```

- Calculation with 48 bit virtual addresses and 4 kB pages:
  - Page size: 12 bit, we use 8 bytes to store 1 entry
  - One page can store 512 entries → we need 9 bits to index it
  - Partitioning a 48 bit address = 9 + 9 + 9 + 9 + 12 bit
    - Only four-level page table is required!
    - $2^{48} = 2^{8} * 2^{40}$ = 256 TB virtual address space
    - $2^{38} = 2^{8} * 2^{30}$ = 256 GB physical memory

HARDWARE MANAGED TLB

- As explained previously
- Address translation: **CPU/MMU...**
  - First it tries to translate the address from the TLB
  - In case of TLB miss: ***it traverses the page table***
    - Valid = 1: updates TLB, done.
    - Valid = 0: Page fault! → calls the operating system and tries again
- Page faults are managed by: **The operating system...**
  - It receives the page number (needed for the address translation)
  - It loads it from the disk (with its private algorithms & data structures)
  - … if it is on the disk at all!
    If nobody else has used that page before, it creates a new one
  - It puts it into the physical memory (throws out someone else, if there is no space)
  - ***It updates the page table***
- Examples: x86, x86-64, ARM, POWER

SOFTWARE MANAGED TLB

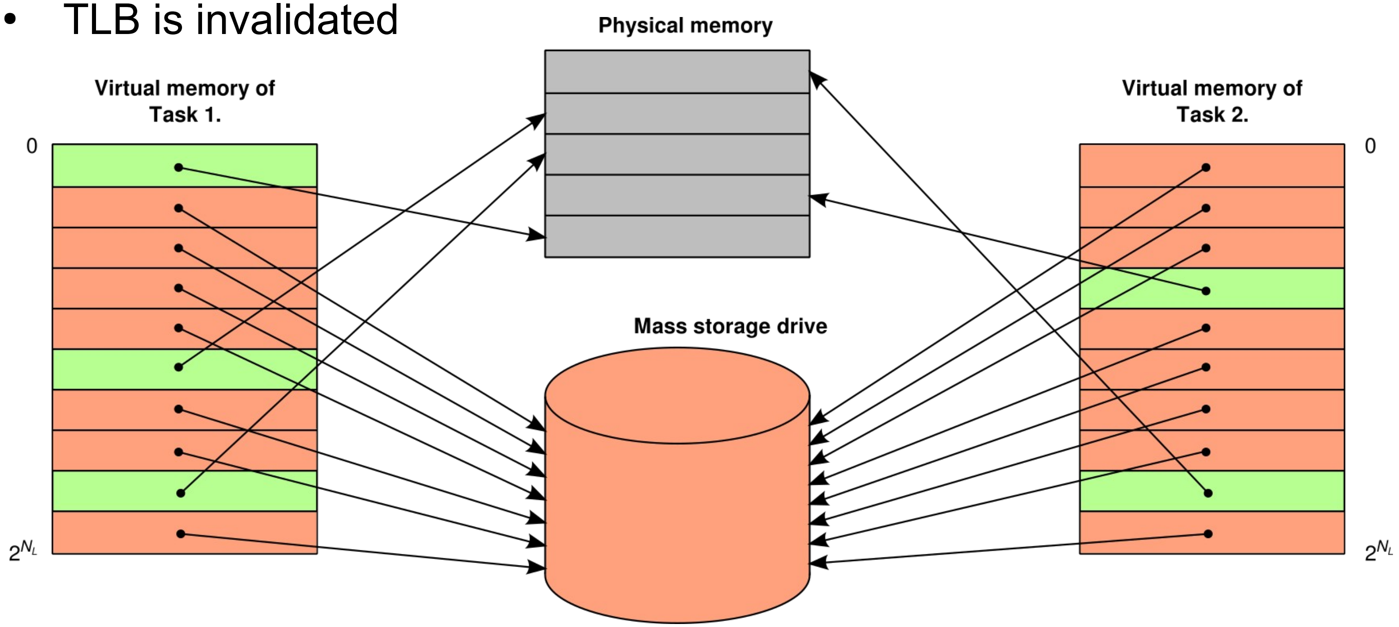
- A less popular alternative solution
- Address translation: **CPU/MMU**...
  - First it tries to translate the address from the TLB
  - In case of TLB miss: ***it calls the operating system***, then it tries again
- TLB fault is resolved by: **The operating system...**
  - It receives the page number
  - It traverses its own private page table
    - Valid = 1: updates the TLB of the CPU with the proper page ↔ frame mapping
    - Valid = 0: like before (swapping, page table update)
  - ***It updates the TLB of the CPU***
- Benefits:
  - No hardware given restrictions, the operating system can introduce a more sophisticated page table any time by a software update
  - More complicated page tables can also be used
- Drawbacks:
  - Much slower address translation (in case of TLB miss)
- Examples: SPARC, Alpha, MIPS, PA-RISC

DIMENSIONING CONSIDERATIONS

- How large the pages should be?
  - As large as possible, since
    - The TLB coverage will be larger
    - Page faults occur less frequently
    - At page fault the page is loaded form the disk
      - Disk loads large pages as fast as small pages
        - Why not to load a larger amount of data at once
  - As small as possible, since
    - With small pages the page contains only those data that are actually used
    - ...and we do not want to waste the small and expensive memory with storing data which accidentally falls onto the same page but which we don’t need
  - In the practice: 4 – 8 kB
    - For special purposes: huge pages (2MB, 1GB)

VIRTUAL MEMORY IN A MULTI-TASKING ENVIRONMENT

- Every task gets the entire virtual address space (from address 0)
- Solution:
  - Every task has a separate page table
  - At task switching:
    - The pointer to the page table is switched
    - TLB is invalidated



- Shared memory regions can be defined, with the same underlying frames
  - Application: e.g. the code segment of multiple instances of the same program

DEPARTMENT OF
NETWORKED SYSTEMS
AND SERVICES

MŰEGYETEM 1782