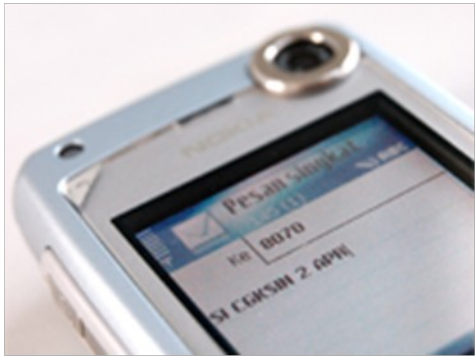




Computer Architecture

Virtual Memory

Gábor Horváth
associate professor
BUTE Dept. Of Telecommunications
ghorvath@hit.bme.hu

2017. március 30.
Budapest

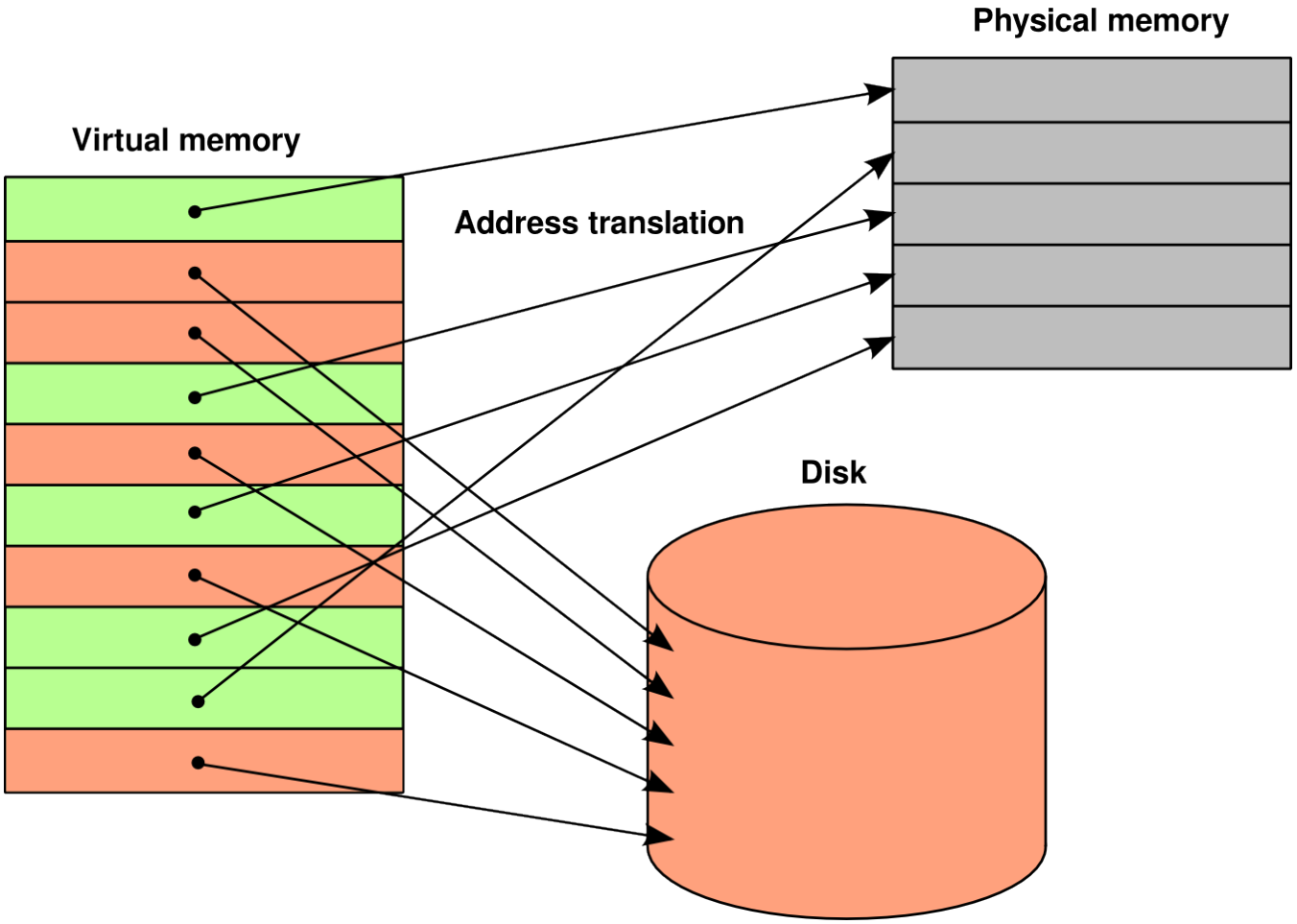
Virtual Memory

- Motivation:
  - 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 amount of RAM we have might not be enough
  - It would be nice if the amount of RAM available was hidden from the programs
    - They think that there is enough amount of RAM present
- The CPU offers all its address space to the programs
- This „virtual” (offered) amount of memory needs to be covered by physical memory

Virtual memory

- Programs use: ***Virtual addresses***
- The address lines on 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:
    - ***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)

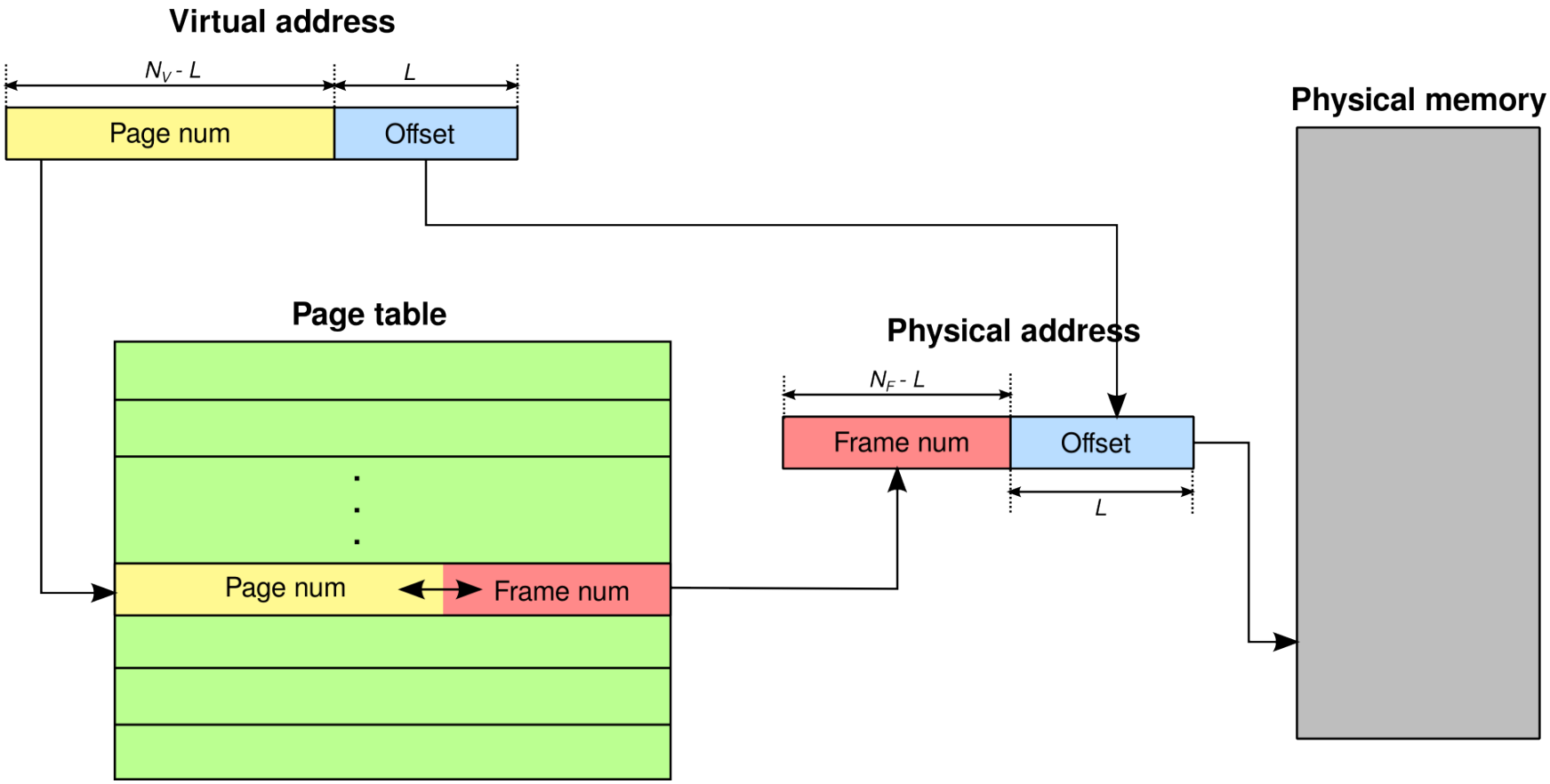
Virtual Memory



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
    - Dirty

Address Translation



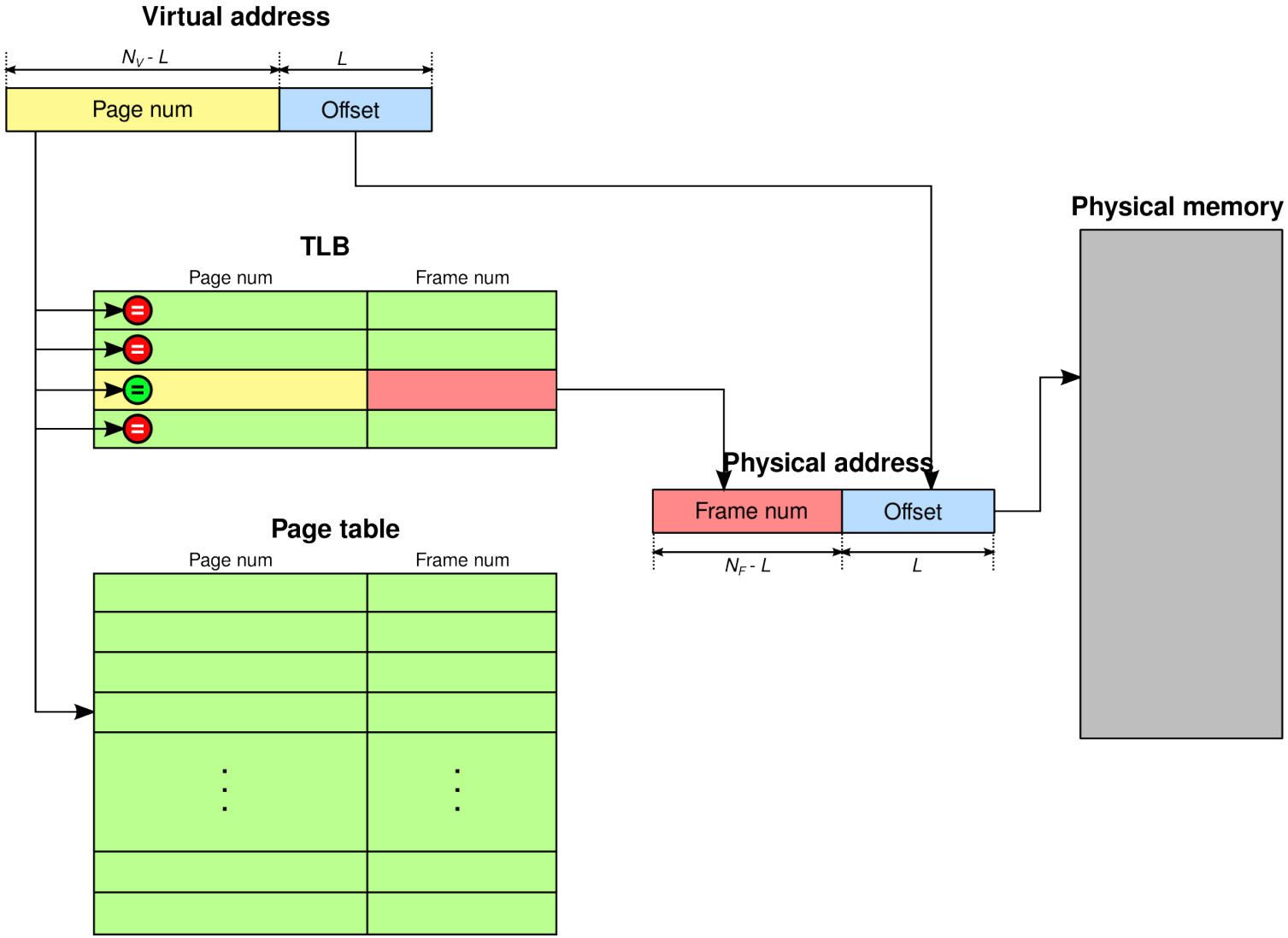
Address Translation

- 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 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 it twice as much
- 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 resiging 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

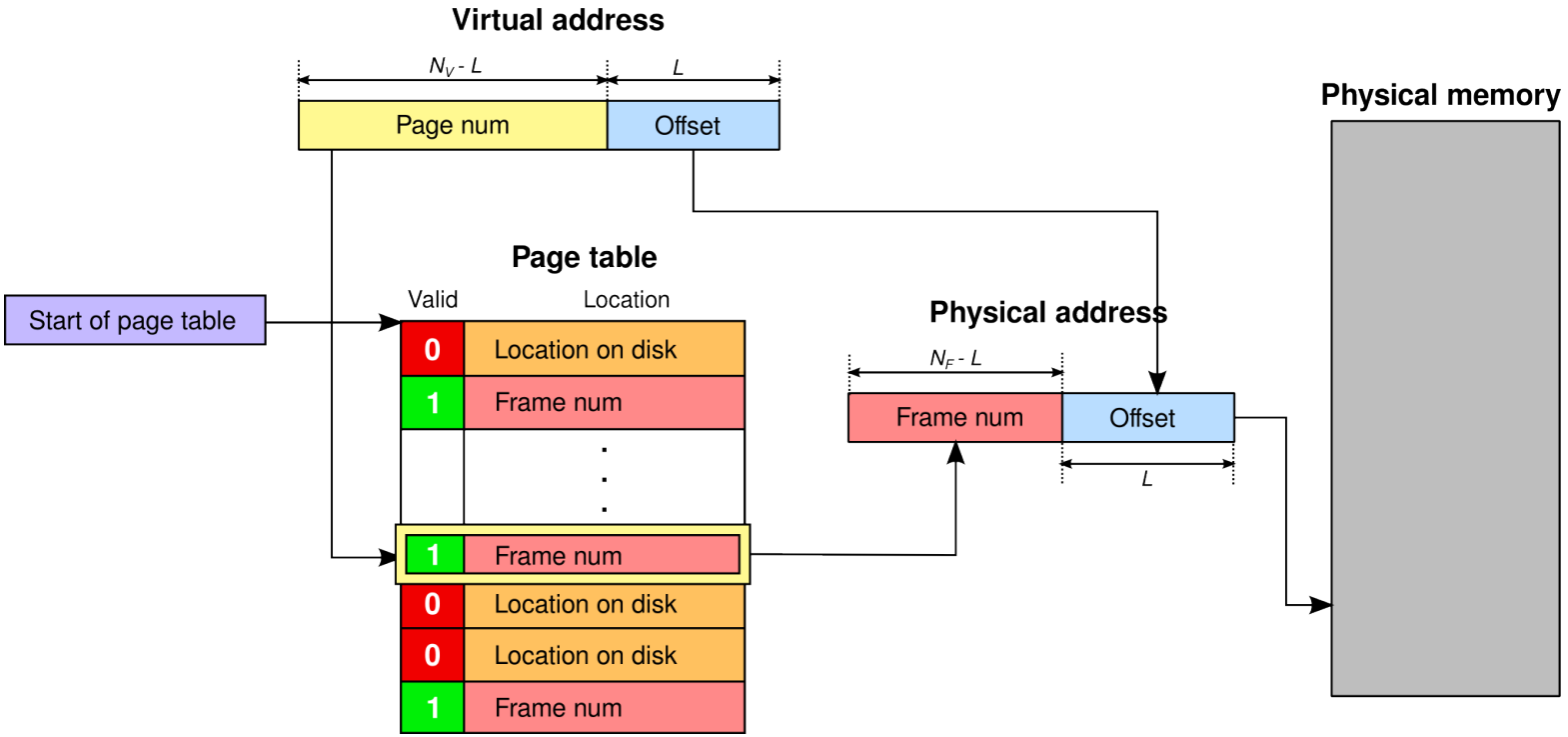
- Goal:
  - Perform address translation as fast as possible

    → 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

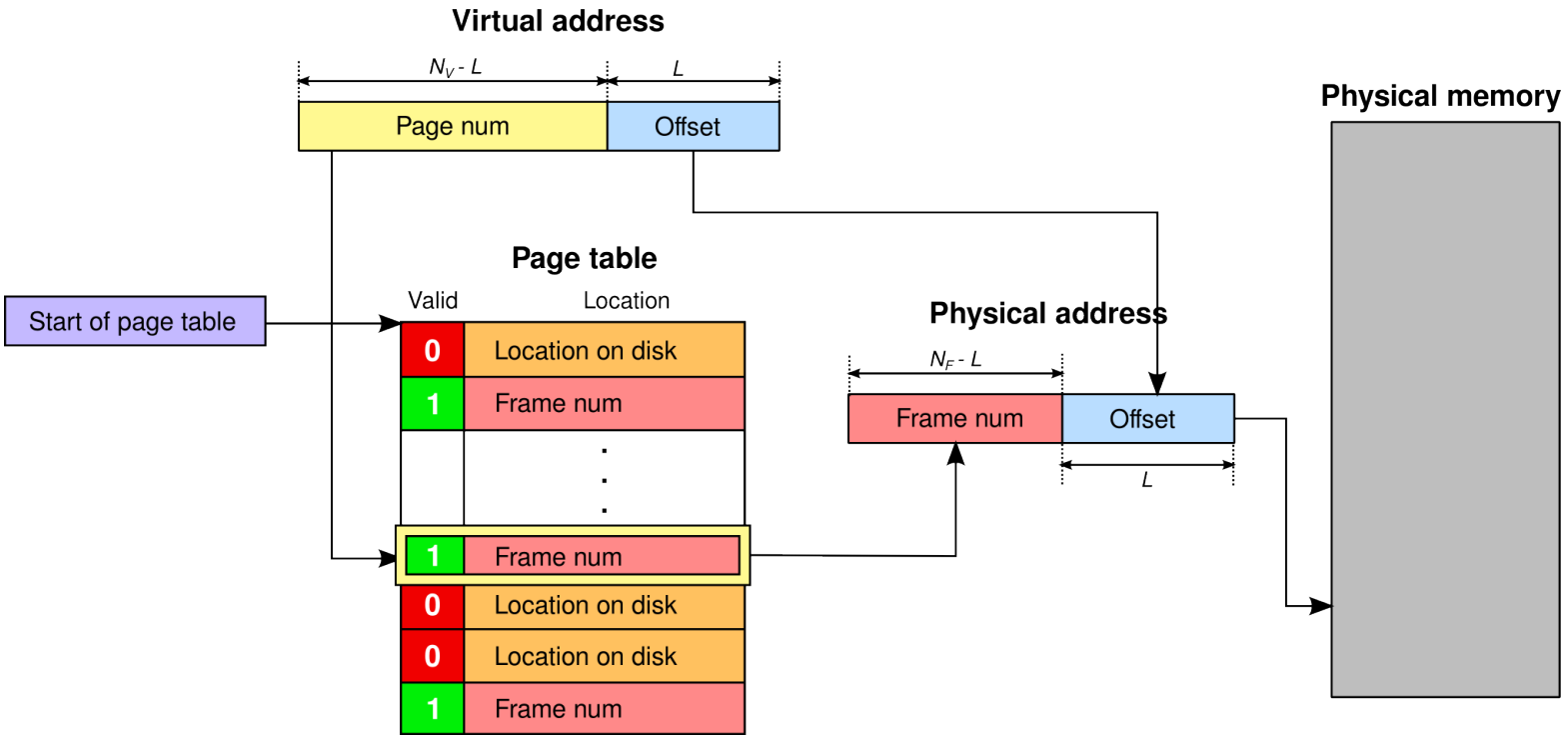
Single-level page tables

- 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



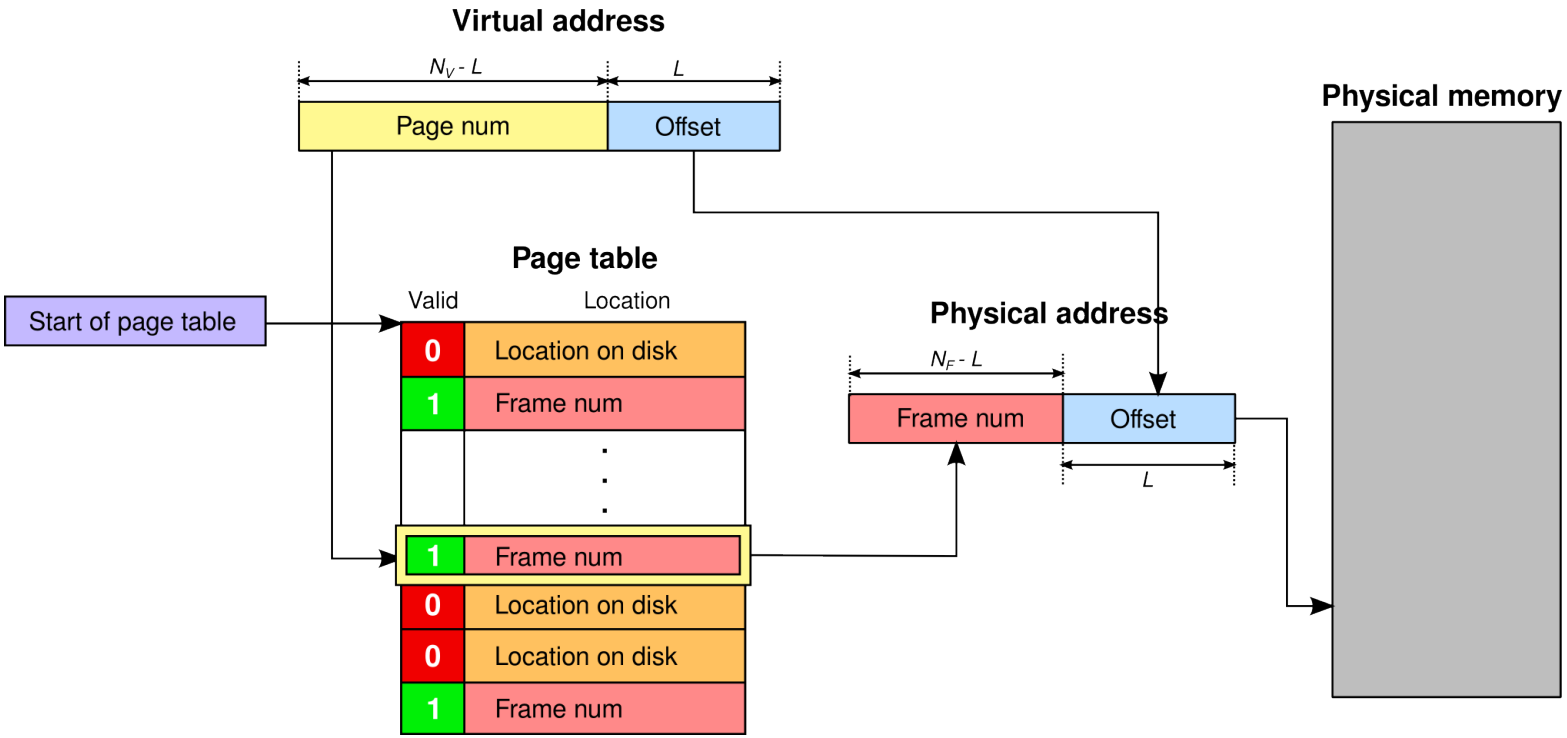
Single-level page tables

- 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

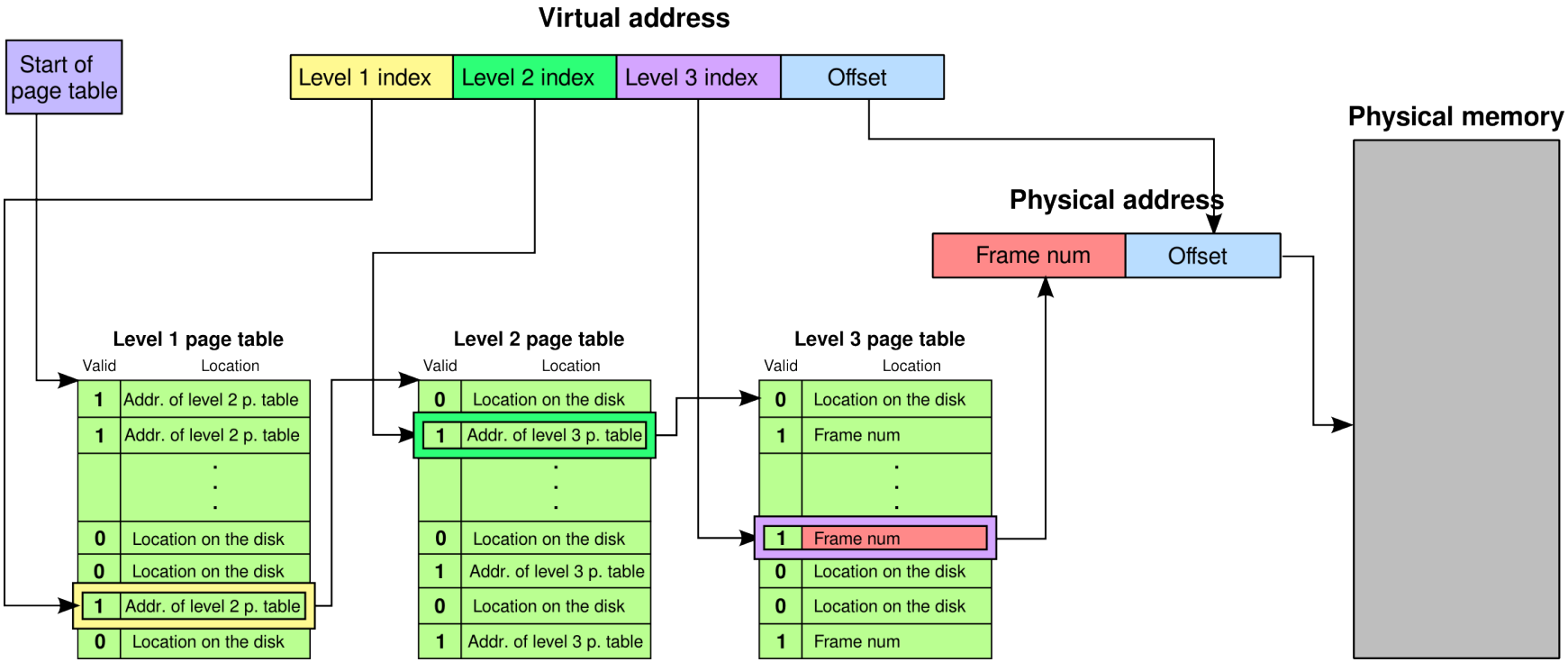
- Fast, needs a single memory access during lookup, the entries are small, ...
  ...isnt it the ideal way to store the page table?

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 PB !!!

Hierarchical page tables

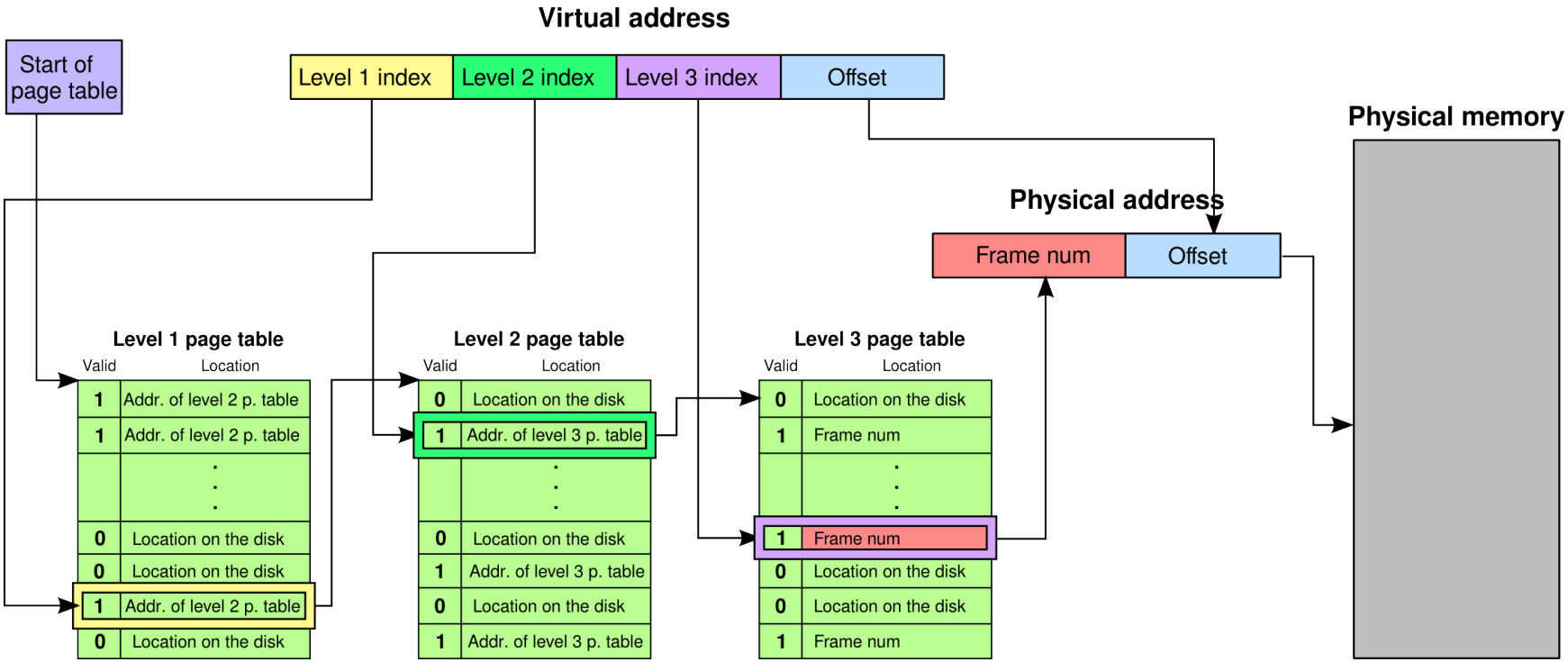
- Idea: let us cut the page table to pages as well
- The locations of pages storing page table entries are stored in an other page. The location of these pages are stored on a third page, etc.



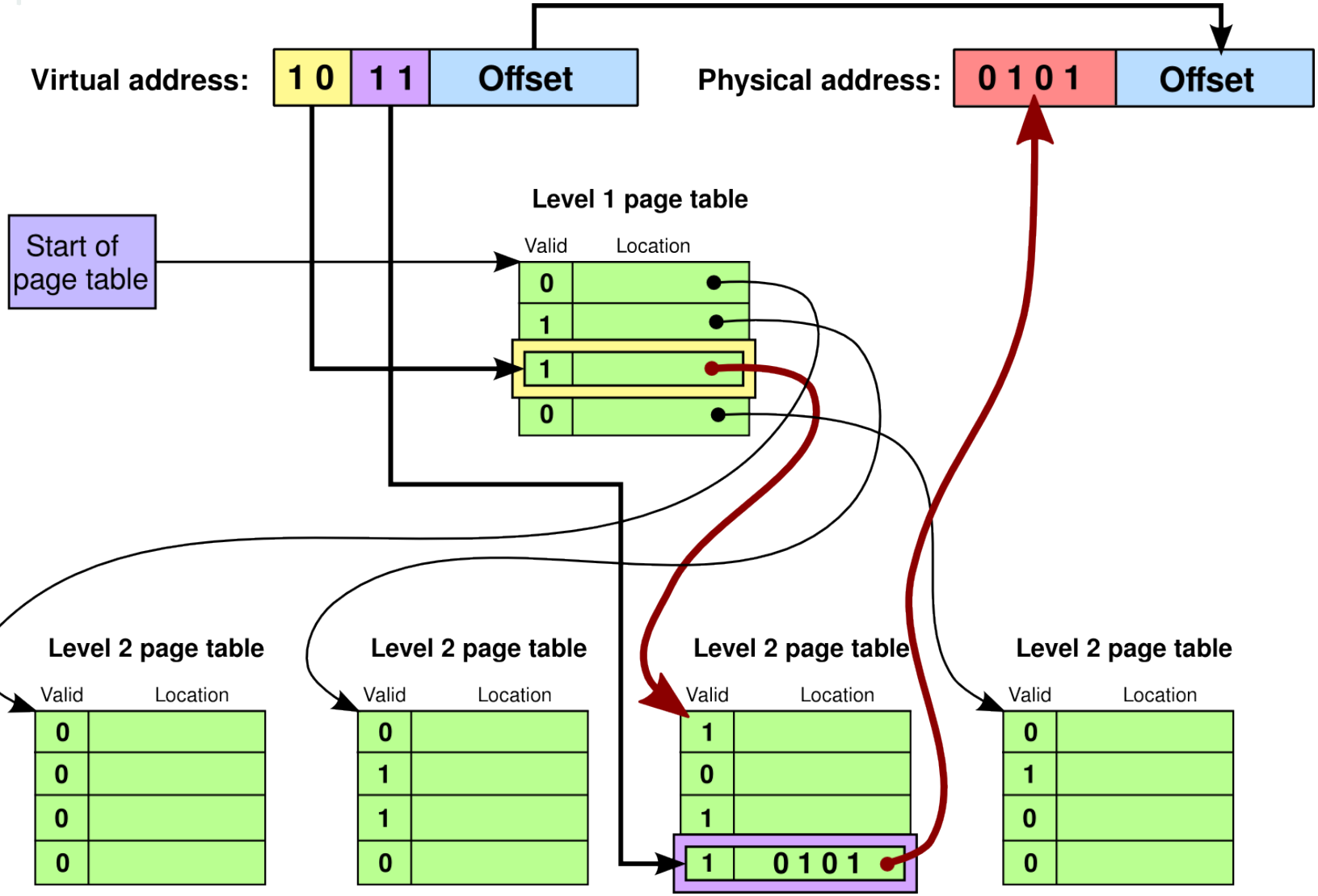
Hierarchical page tables

- 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!



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 10 bits to index it
    - Partitioning a 64 bit address = 7 + 9 + 9 + 9 + 9 + 9 + 12 bit

      → A six-level page table is required!

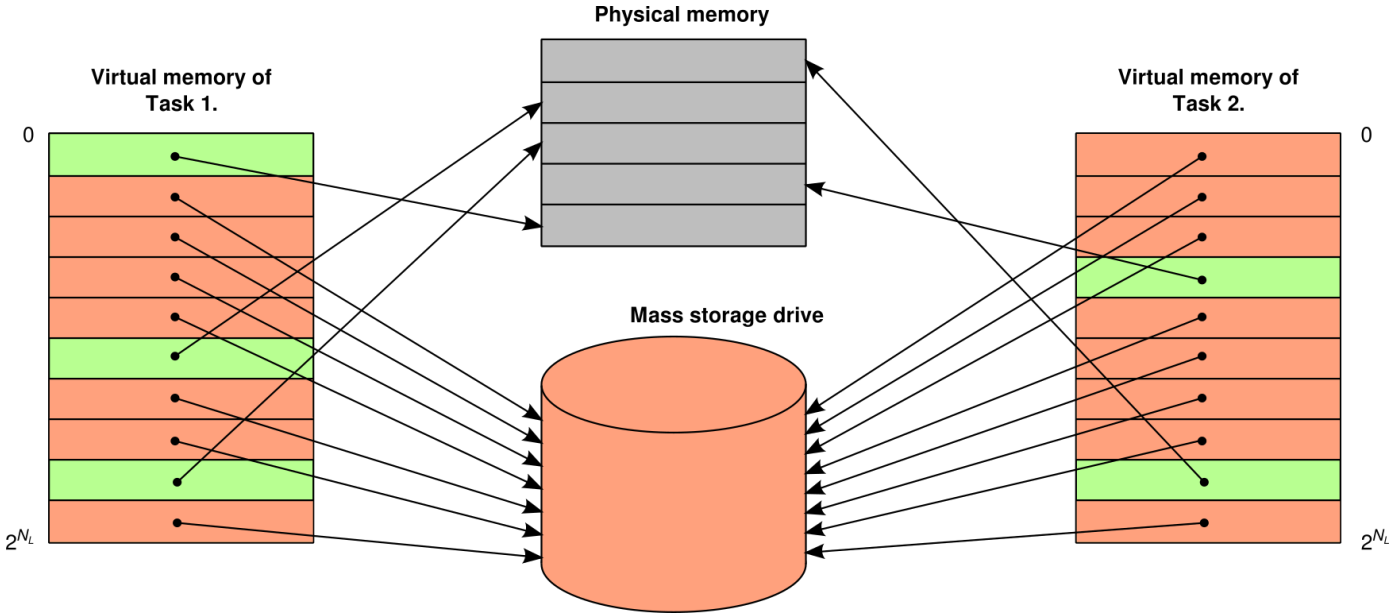
      6 memory operations are needed for each address translation !!!
- X86, ARM architectures are using hierarchical page tables

Dimensioning considerations

- How large the pages should be?
  - As large as possible, since
    - The TLB coverage will be larger
    - Page faults occure 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 dont need
  - In the practice: 4 – 8 kB

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