Virtual Memory (optional)

Why Virtual Memory?

Motivations:

- Protection
  - Ensure that programs cannot interfere with each others
- Sharing of memory between programs to increase memory utilization
  - As running programs only actively use a fraction of the memory
- Allowing a program to exceed the size of the main memory
  - Use secondary storage (e.g. magnetic disks) to backup
  - i.e. make use of the main memory as a "cache" for magnetic disks

In systems today,

- Memory address in our programs is considered as **virtual address**

Virtual memory is the technique to seamlessly **map** virtual addresses to physical addresses
(seamlessly = automatically map in hardware)

Virtual Address vs Physical Address

The processor generates virtual addresses

- While the memory is accessed using physical addresses

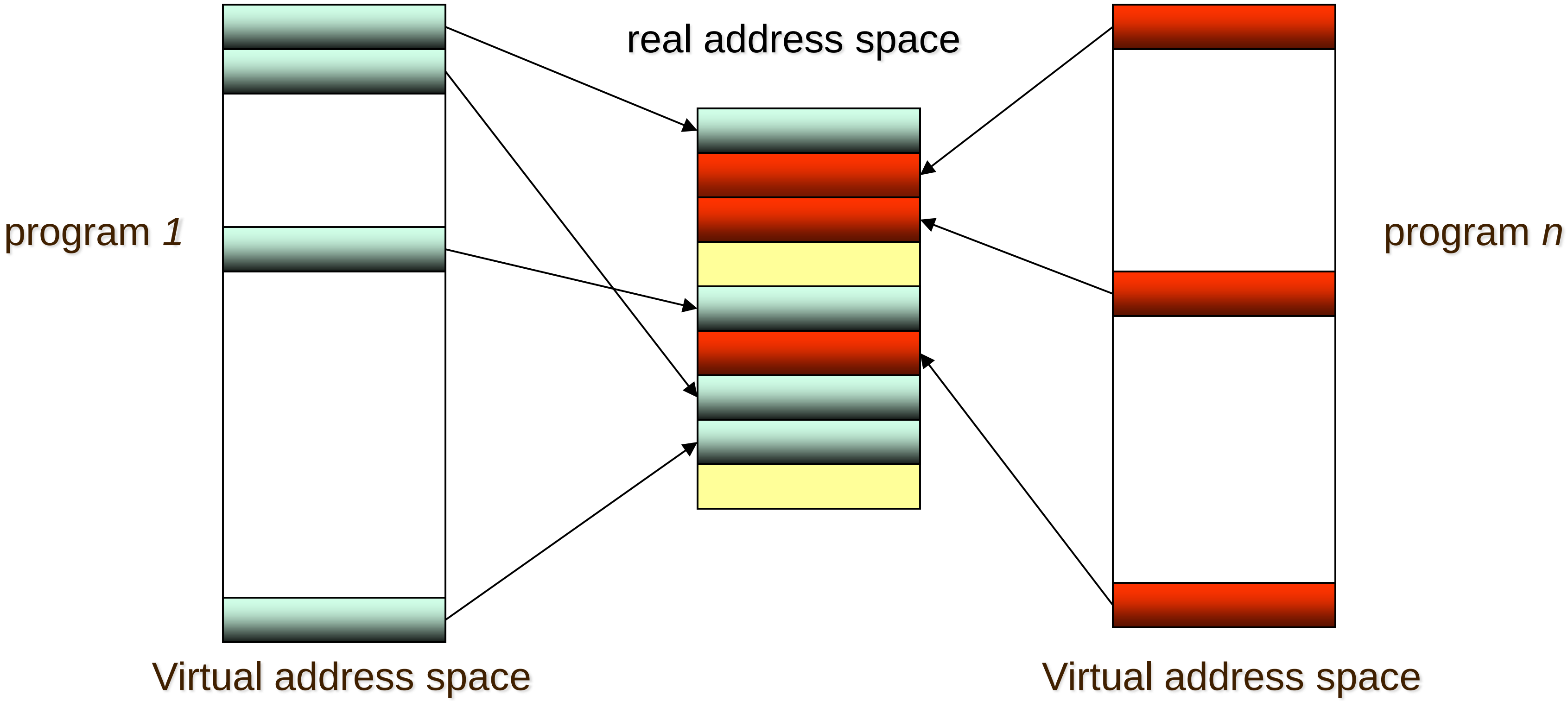
That means, programs see the virtual address space

- While the system sees the physical address space
- Virtual address space is as big as a register can address

Ideally, physical address = mapping_function(virtual address)

- Mapping function can be implemented as a table in system memory
- But, if we map at word or block level, the table is too big!
- Instead, split the virtual and physical address space into pages
  - A typical page size is 4Kbytes
  - Then, the mapping is between virtual and physical pages

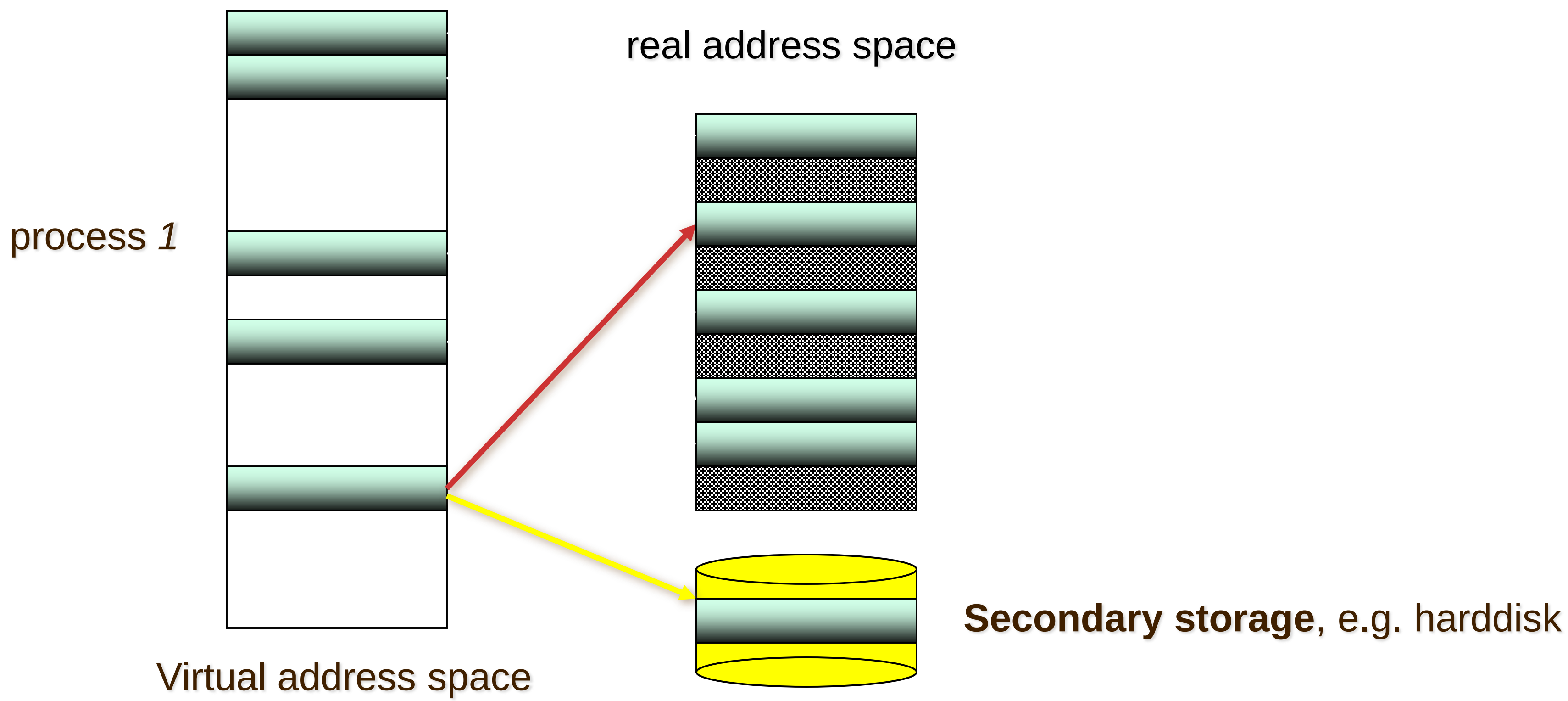
Virtual Memory: Basic Concepts



- The "*mapping function*" is implemented as a *table*

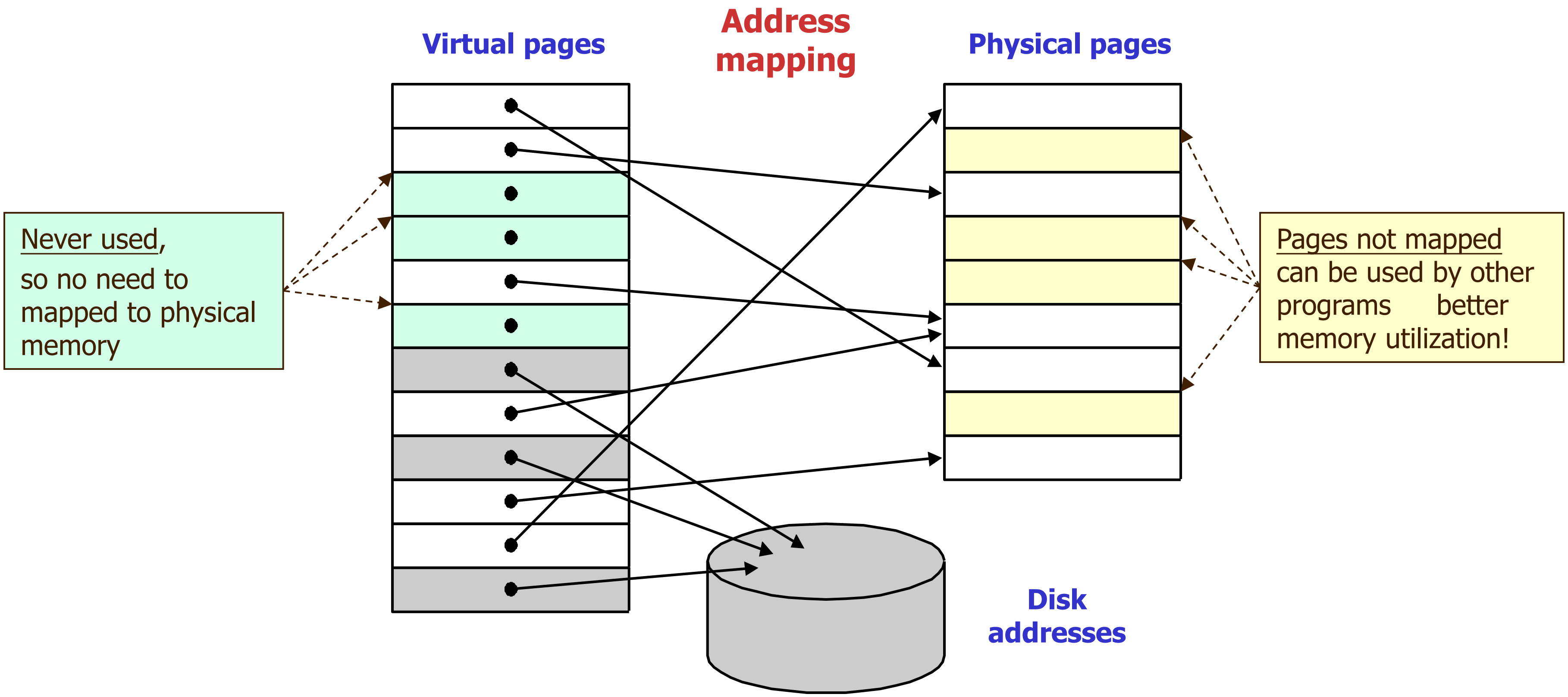
More precisely

- Portion of V, when not currently in use, is stored in **secondary storage**
- When it is requested later, OS shuttles it into the memory, replacing other portions not currently in use (replacement policy answers this part)



How Does Virtual Memory Work?

- Virtual page size = physical page size

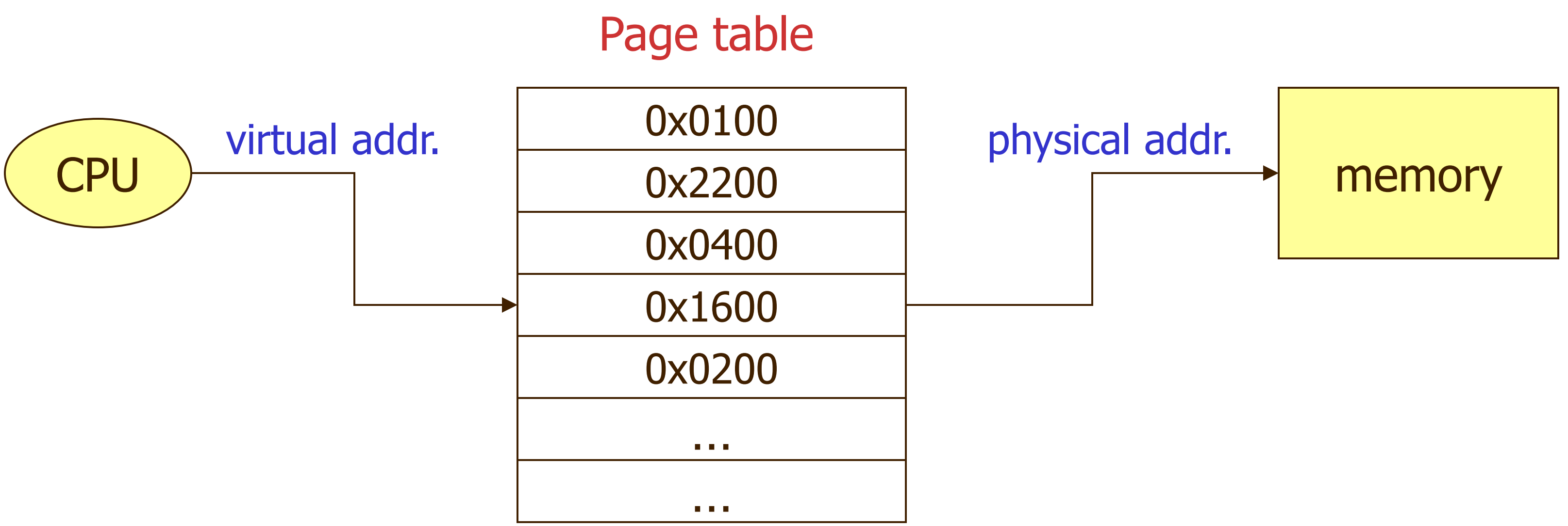


Page Table

Implementation of mapping function: page table

- Page table maps virtual pages to physical pages

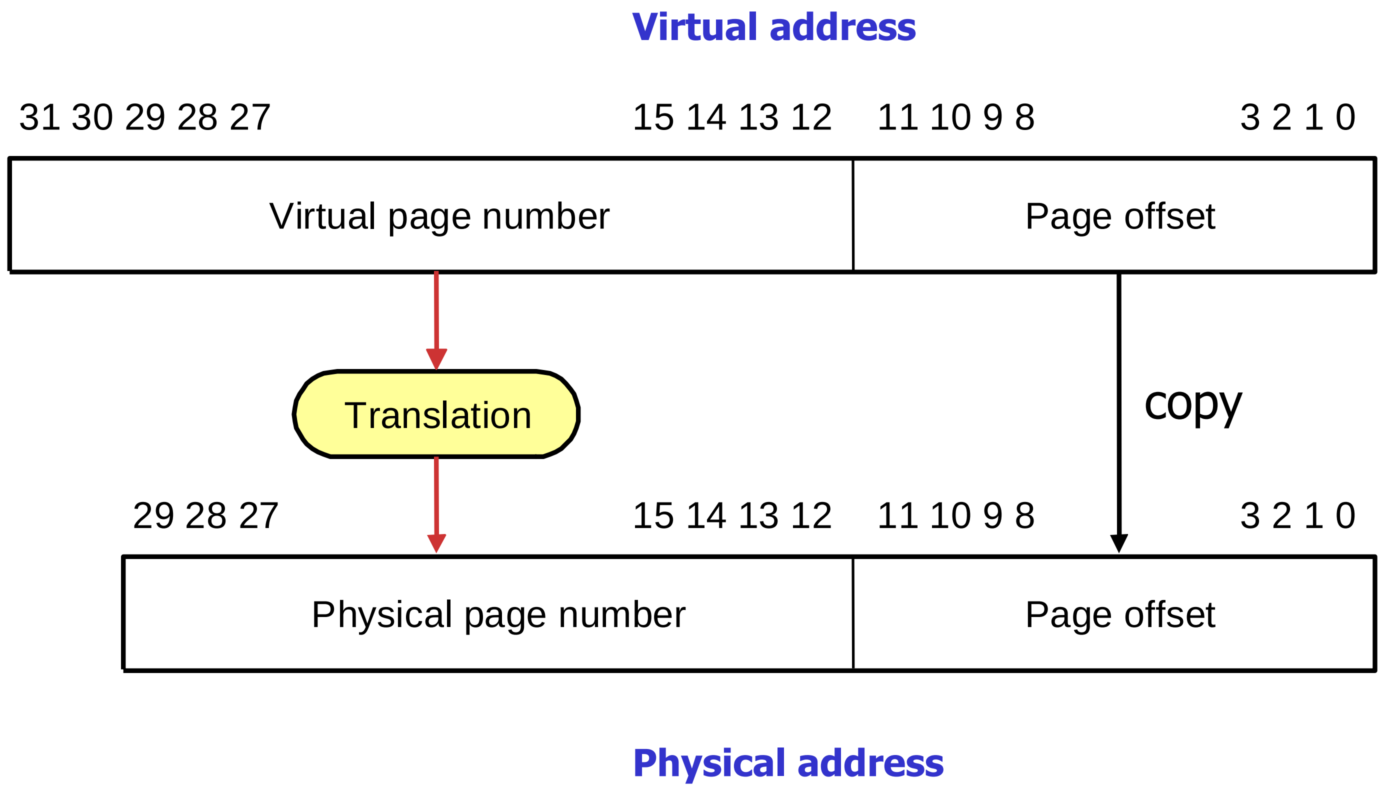
Example



Question: How to convert a virtual address into a virtual page number?

Address Translation Example

- Virtual address space: 4 GB (2^{32})
- Maximum main (physical) memory size: 1 GB (2^{30})
- Page size: 4 KB (2^{12})



Page Faults

- It is possible that a mapping does not exist upon first access
  - Try to map through the page table results in a page fault

- Operating system is invoked to resolve the page fault
  - Resolve means find the mapping or setup appropriate mapping

- A page fault usually has an enormous penalty
  - Page faults are handled in software
  - It can take millions of clock cycles to process
  - Dominated by the time to get the first word for typical page sizes
  - Program execution is stalled until page fault is resolved

So, pages should be large enough

- Too small the page size, too often we see page faults
- Too large the page size, higher chances of page fragmentation

Page Table: Put All Together

Mapping virtual addresses to physical addresses through a **page table**

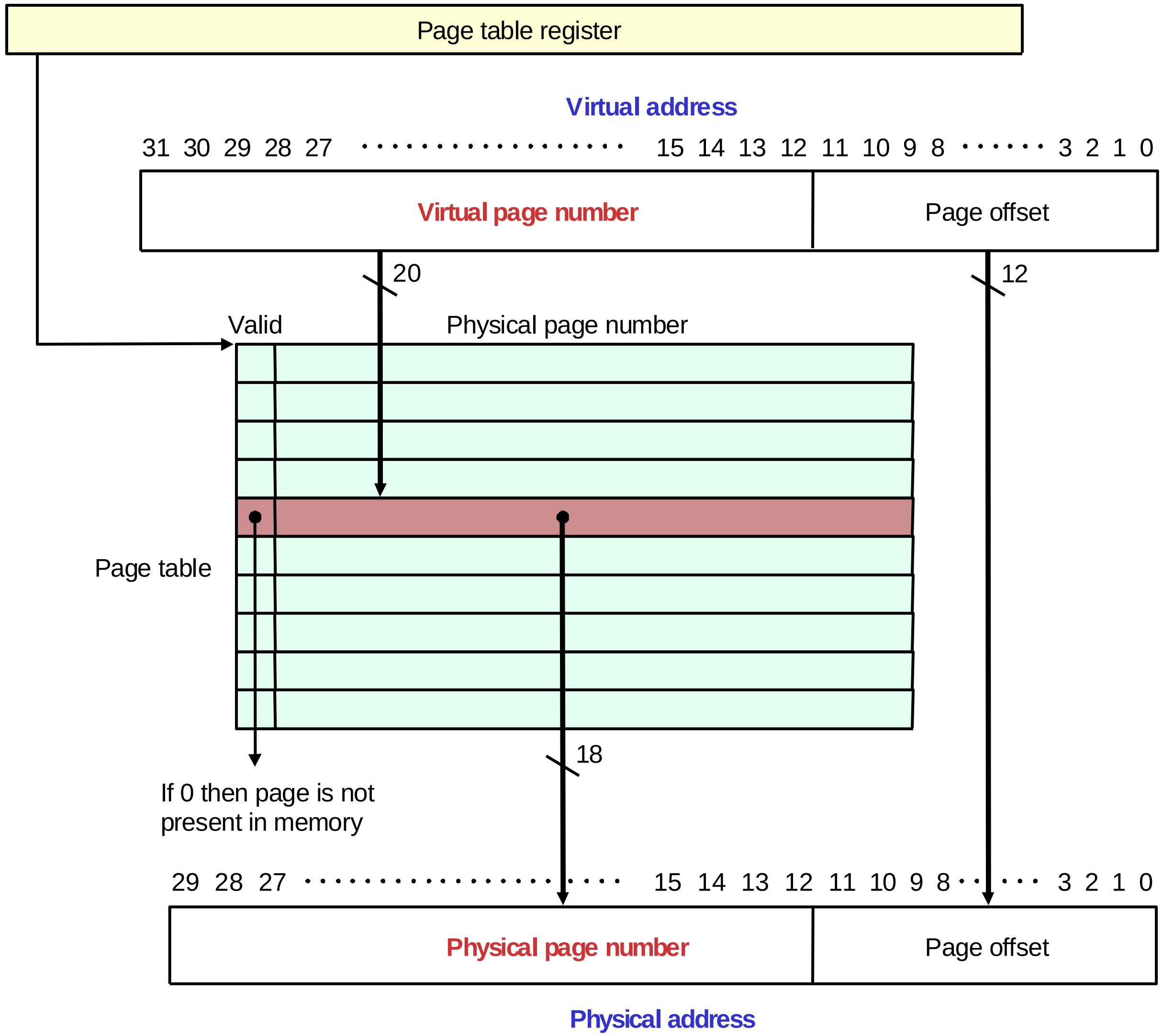
- Page table is a structure that resides in the memory
- Starting address of the page table is stored in the page table register
- Each page table entry stores

  a valid bit to indicate if the mapping exists, and

  the corresponding physical page number
- Since every possible virtual page is represented in the page table,

➢ There is no need to have a tag field

Page Table: Put All Together (cont'd)



Example of Looking Up Page Table

- Page size = 4Kbyte (2^{12})
- Virtual address space = 2^{32}
- Physical address space = 2^{28}
- What are the sizes of the virtual and physical page number?

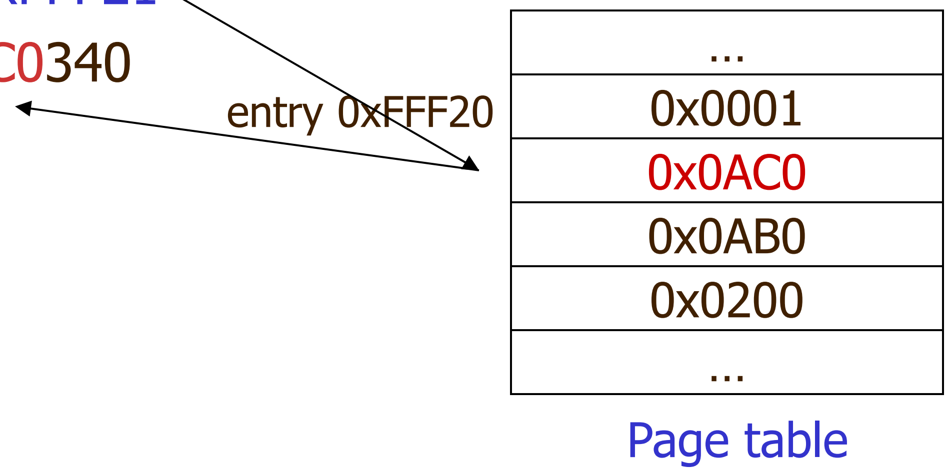
  Size of virtual page number = 32 – 12 = 20

  Size of physical page number = 28 – 12 = 16
- What is the physical address for 0xFFF21340 using page table below?

  0xFFF21340 = 1111 1111 1111 0010 0001 0011 0100 0000_2

  Virtual page number = 0xFFF21

  Physical address = 0x0AC0340



Fast Address Translation

Problem with pure page table approach

- Page tables are in main memory
- Every memory access by a program can take at least **twice as long**
  - One memory access to obtain the physical address
  - The second access to get the data
- **Bad performance**!

Solution

- Translation-lookaside buffer (TLB), a cache copy of the page table
  - TLB relies on the locality of reference to the page table
  - When a translation for a virtual page number is used, it will probably be needed again in the near future as the references to the words on that page have both temporal and spatial locality
- i.e. TLB is a special cache keeping track of recently used translations
- TLB is usually a small fully-associative cache, (e.g. 16~64 entries)

Working with TLB and Page Table

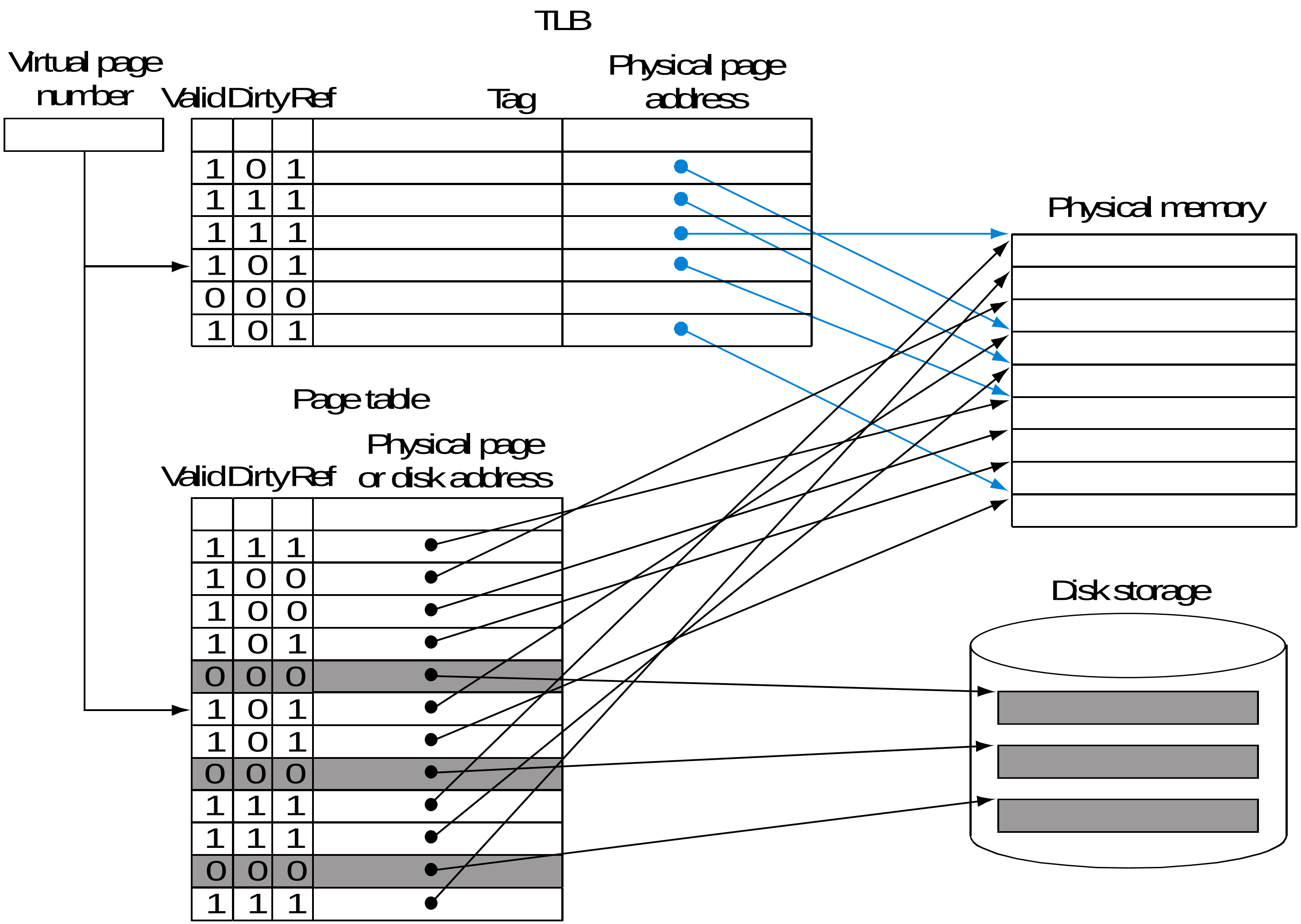
Upon each memory access

- Mapping for virtual address generated by CPU is first looked up in TLB
- If found, do the translation and done
- If not found, then looked up in the page table residing in memory

If mapping (i.e. translation) not found in the page table,

- Page fault!
- OS is invoked to handle the page fault
- After OS resolve the page fault, the memory access is restarted

Translation-Lookaside Buffer (TLB)



Key Concepts to Remember

- Ordinary programs exhibit two different notions of locality
  - **Temporal locality** and **spatial locality**
- Multilevel memory organizations achieve cost/performance tradeoff by exploiting the **principle of locality**
- **Cache**
  - Direct-mapped, set-associative, or fully-associative
  - Data are transferred in blocks from main memory to cache upon misses
  - Block replacement uses either random or least recently used (LRU)
  - The write strategy for caches is either write-through or write-back
- **Virtual memory**
  - The technique to seamlessly map virtual addresses to physical addresses
  - Needed for protection and efficient sharing of memory among programs
  - Mapping is implemented via a page table
  - TLB is cache copy of page table for the sake of performance