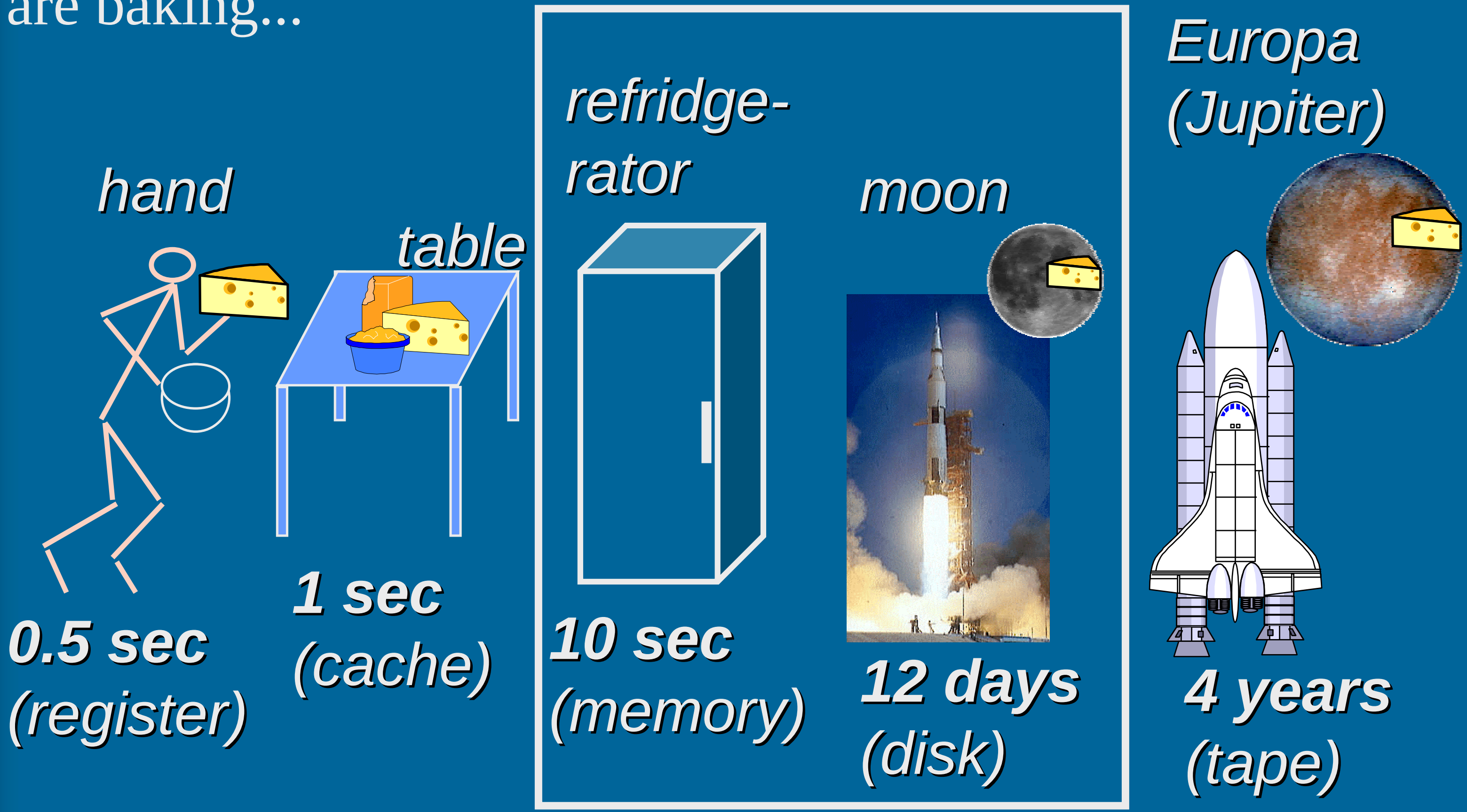
Virtual Memory (VM)
Ch 8.3

Memory Management
Address Translation
Paging
Hardware Support
VM and Cache

Teemu's Cheesecake

Register, on-chip cache, memory, disk, and tape speeds relative to times locating cheese for the cheese cake you are baking...



Virtual Memory

(virtuaalimuisti)

- Problem: How can I make my (main) memory as big as my disk drive?
- Answer: Virtual memory
  - keep only most probably referenced data in memory, and rest of it in disk
    - disk is much bigger and slower than memory
    - address in machine instruction may be different than memory address
    - need to have efficient address mapping
    - most of references are for data in memory
  - joint solution with HW & SW

Other Problems Often Solved with VM (3)

- If you must want to have many processes in memory at the same time, how do you keep track of memory usage?
- How do you prevent one process from touching another process' memory areas?
- What if a process needs more memory than we have?

Memory Management Problem (4)

- How much memory for each process?
  - is it fixed <u>amount</u> during the process run time or can it vary during the run time?
- Where should that memory be?
  - in a continuous or discontinuous area?
  - is the <u>location</u> the same during the run time or can it vary dynamically during the run time?
- How is that memory managed?
- How is that memory referenced?

Partitioning (3)

- How much physical memory for each process?
- Static (fixed) partitioning (staattiset tai kiinteät partitiot)
  - amount of physical memory determined at process creation time
  - continuous memory allocation for partition
- Dynamic partitioning (dynaamiset partitiot)
  - amount of physical memory given to a process varies in time
    - due to process requirements (of this process)
    - due to system (I.e., other processes) requirements

Static Partitioning

- Equal size - give everybody the same amount
  - fixed size - big enough for everybody
    - too much for most
  - need more? Can not run!
- Unequal size
  - sizes predetermined
- Variable size
  - Size determined at process creation time

Fig. 8.13 (a)
(Fig. 7.14 (a) [Stal99])

Fig. 8.13 (b)
(Fig. 7.14 (b) [Stal99])

Fig. 8.14 (Fig. 7.15 [Stal99])

Fragmentation

- Internal fragmentation (sisäinen pirstoutuminen)
  - unused memory inside allocated block
  - e.g., equal size fixed memory partitions

    Fig. 8.13 (a)
    (Fig. 7.14 (a) [Stal99])
- External fragmentation (ulkoinen pirstoutuminen)
  - enough free memory, but it is splintered as many un-allocatable blocks

    Fig. 8.13 (b)
    (Fig. 7.14 (b) [Stal99])
  - e.g., unequal size partitions or dynamic fixed size (variable size) memory partitions

    Fig. 8.14
    (Fig. 7.15 [Stal99])

Dynamic Partitioning (3)

- Process must be able to run with varying amounts of main memory
  - all of memory space is **not** in physical memory
  - need some minimum amount of memory
- New process?
  - reduce amount of memory for some (lower priority) processes
- Not enough memory for some process?
  - reduce amount of memory for some (lower priority) processes
  - kick (swap) out some (lower priority) process

Address Mapping (4) (osoitteen muunnos)

Pascal, Java:

```
while (....)
        X := Y+Z;
```

Symbolic Assembler:

```
loop:  LOAD    R1, Y
       ADD     R1, Z
       STORE   R1, X
```

Textual machine language:

```
1312:  LOAD    R1, 2510
       ADD     R1, 2514
       STORE   R1, 2600
```

(addresses relative to 0)

Execution time:

```
101312:  LOAD   R1,102510
         ADD    R1,102514
         ADD    R1,102600
```

(real, actual!)

Address Mapping (2)

logical address

Textual machine language:

1312: LOAD R1, 2510

+100000?

Execution time:

101312: LOAD R1,102510 or

101312: LOAD R1, 2510 ??

physical address (constant?)

logical addr

- Want: R1 ← Mem[102510] or Mem[2510] ?
- Who makes the mapping? When?

Address Mapping (2)

- At program load time
  - loader (lataaja)
  - static address binding (staattinen osoitteiden sidonta)
- At program execution time
  - cpu
  - with every instruction
  - dynamic address binding (dynaaminen osoitteiden sidonta)
  - swapping
  - virtual memory

Swapping (4)

(heittovaihto)

- Keep all memory areas for all running and ready-to-run processes in memory
- New process
  - find continuous memory partition and swap the process in
- Not enough memory?
  - Swap some (lower priority) process out
- Some times can swap in only (runnable) portions of one process
- Address map: add base address

VM Implementation (2)

- Methods
  - base and limit registers
  - segmentation
  - paging
  - segmented paging, multilevel paging
- Hardware support
  - MMU - Memory Management Unit
    - part of processor
    - varies with different methods
  - Sets limits on what types of virtual memory (methods) can be implemented using this HW

Base and Limit Registers (2)

- Continuous memory partitions
  - one or more (4?) per process
  - may have separate base and limit registers
    - code, data, shared data, etc
    - by default, or given explicitly in each mem. ref.
- *BASE* and *LIMIT* registers in MMU
  - all addresses logical in machine instructions
  - address mapping for address (x):
    - check: x < *LIMIT*
    - physical address: *BASE*+x

Segmentation (4)

- Process address space divided into (relatively large) logical segments
  - code, data, shared data, large table, etc
  - object, module, etc
- Each logical segment is allocated its own continuous physical memory segment
- Memory address has two fields

011001 1010110000

segment byte offset

(lisäys)

Segment. Address Mapping (3)

- Segment table
  - maps segment id to physical segment base address and to segment size
- Physical address
  - find entry in segment table
  - check: byte offset < segment size
  - physical address: base + byte offset
- Problem: variable size segments
  - External fragmentation, lots of memory management

Paging (4)

- Process address space divided into (relatively small) equal size pages
  - address space division is not based on logical entities, only on fixed size chunks designed for efficient implementation
- Each page is allocated its own physical page frame in memory
  - any page frame will do!
- Internal fragmentation
- Memory addresses have two fields

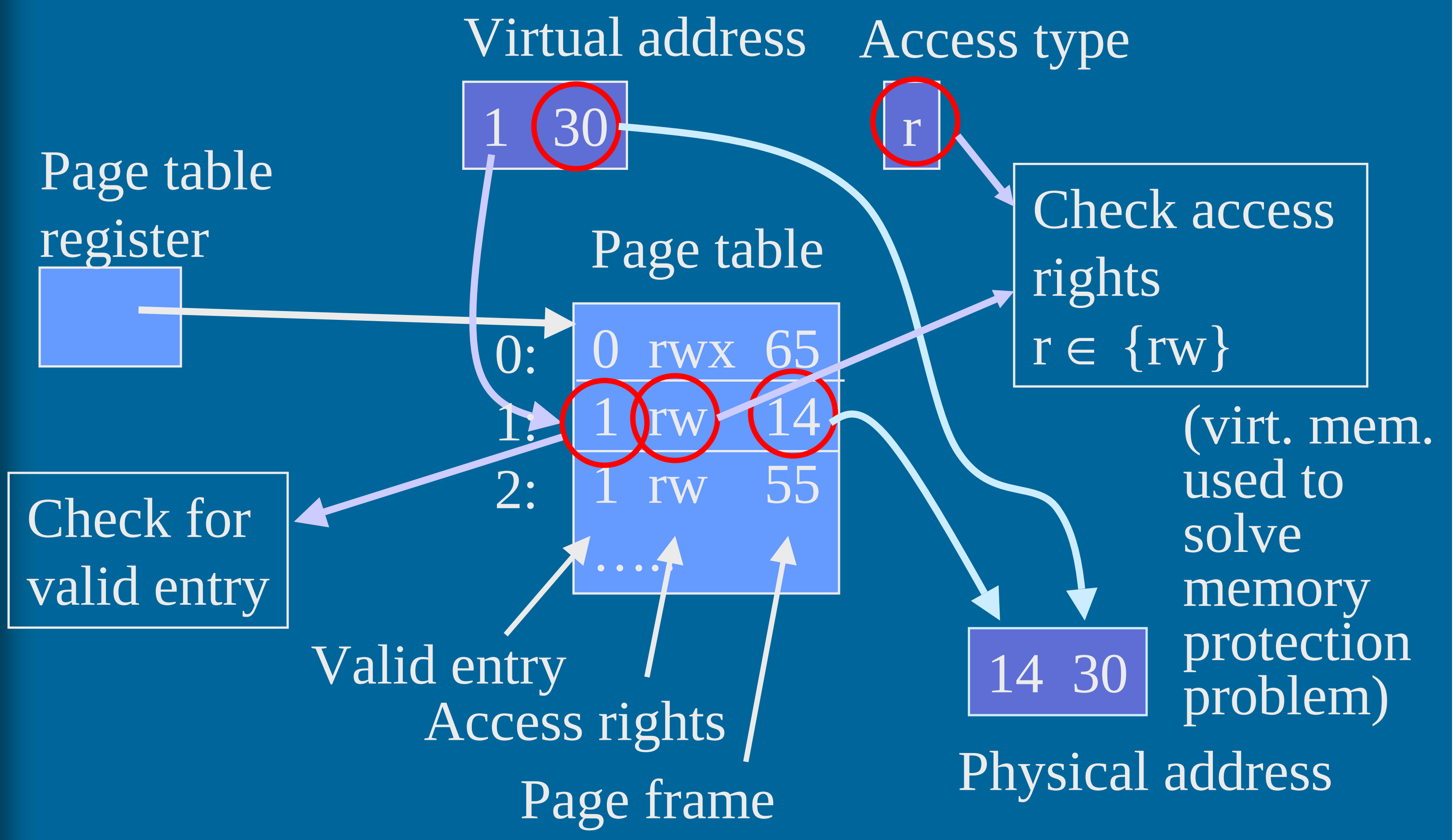
01100110 10110000

page byte offset (lisäys)

Paged Address Mapping

- Page table
  - maps page nr to physical page frame
- Physical address
  - find entry in page table (large array in memory)
  - get page frame, I.e., page address
  - physical address: page address + byte offset

Paged Address Translation (4)



Page Fault (12)

Page fault interrupt

Stop execution

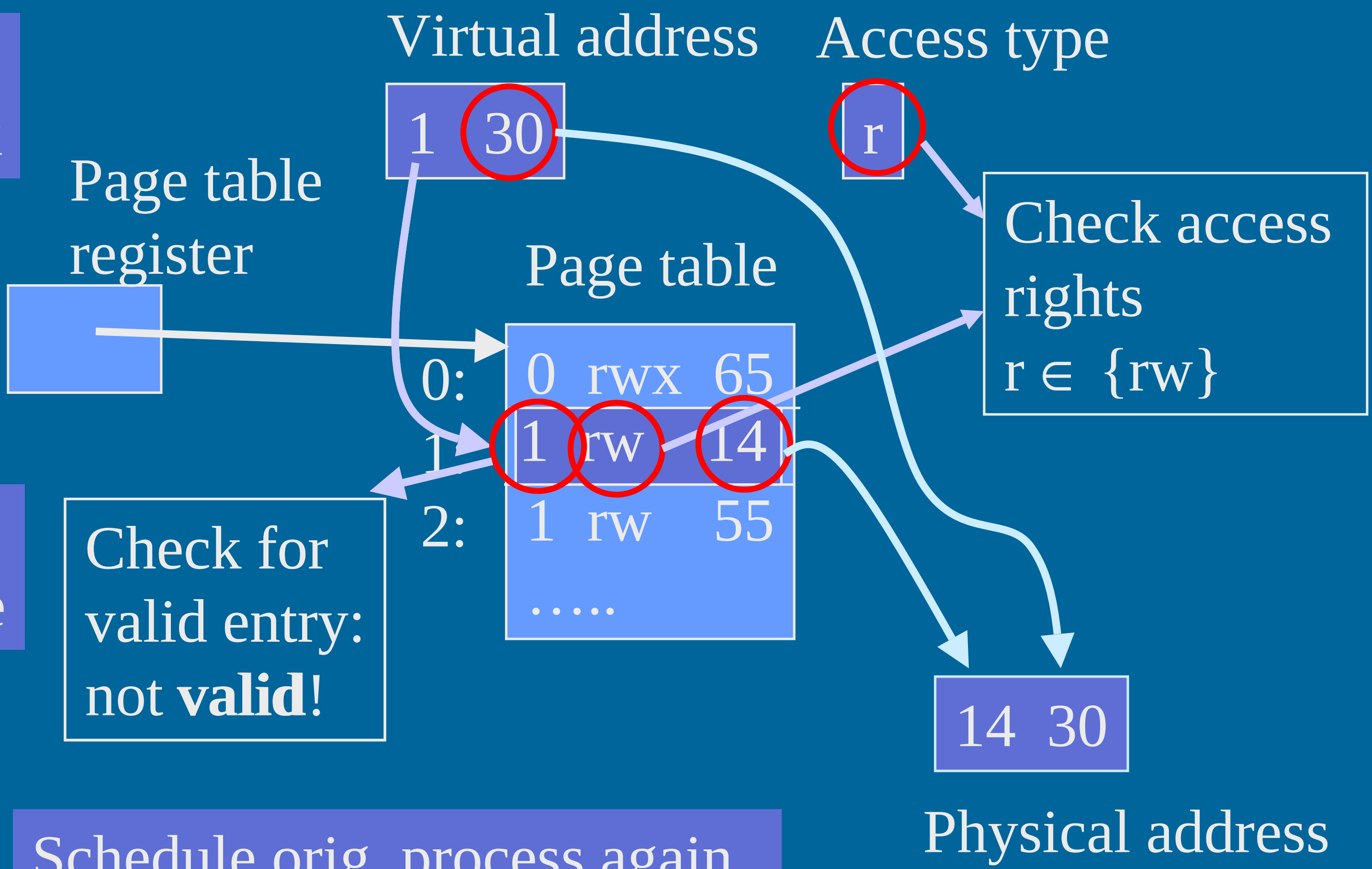
Initiate reading page 1 from disk

Schedule next process to run

I/O interrupt

Page 1 read, update page table

Make orig. process ready-to-run



Schedule orig. process again, at the same instruction

Paging (3)

- Physical memory partitioning
  - discontinuos areas

Fig. 8.15

(Fig. 7.16 [Stal99])

- Page tables
  - located in memory
  - can be very big, and each process has its own
    - entry for each page in address space
- Inverted page table
  - entry for each page in memory
  - less space, more complex hashed loc

(Fig. 7.18 [Stal99])

Fig. 8.17

Address Translation (3)

- MMU does it for every memory access
  - code, data
  - more than once per machine instruction!
- Can not access page tables in memory every time - it would be too slow!
  - too high cost to pay for virtual memory?
- MMU has a “cache” of most recent address translations
  - TLB - Translation Lookaside Buffer
  - 99.9% hit ratio?

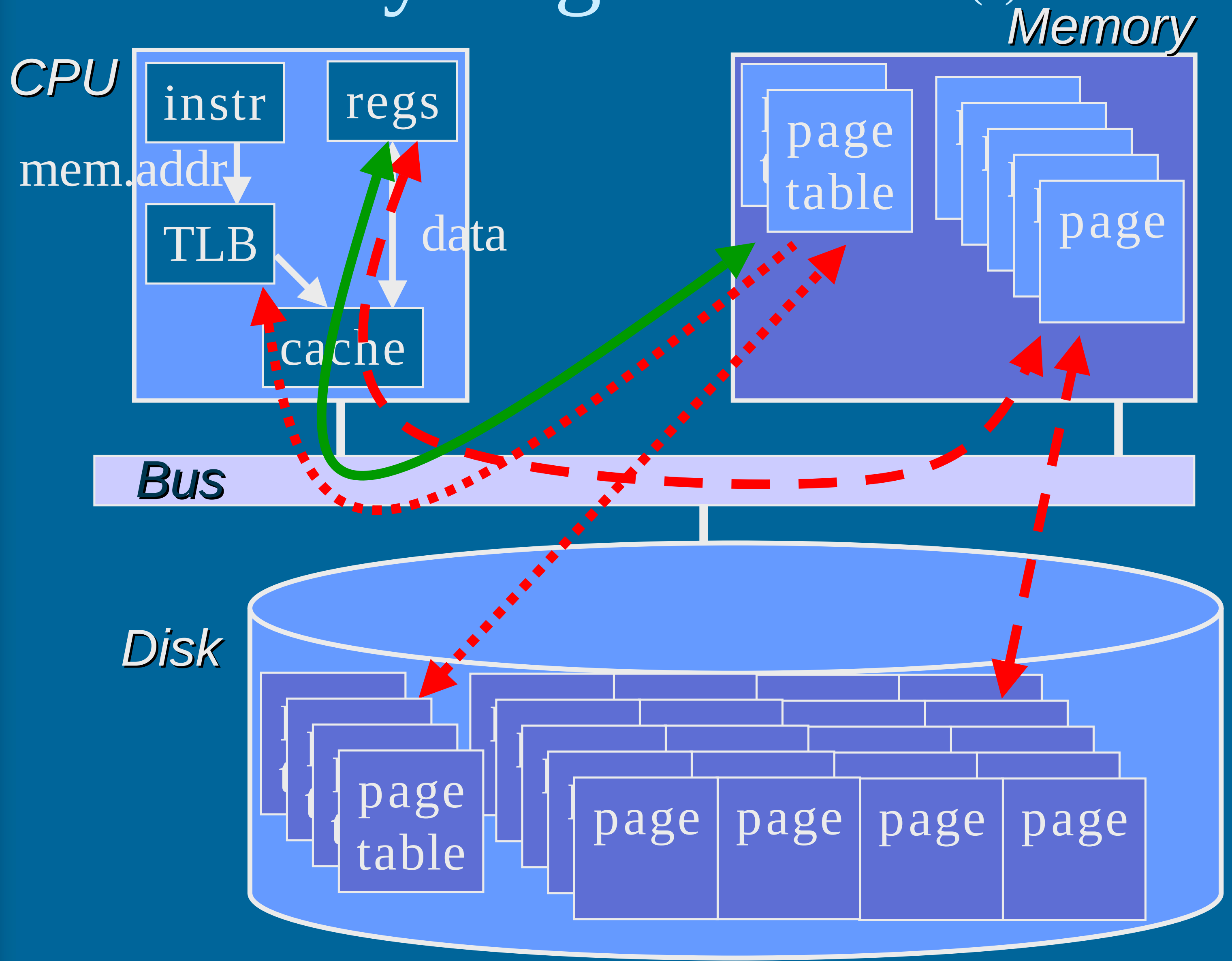
(osoitteen-muunnos-taulukko)

Translation Lookaside Buffer (3)

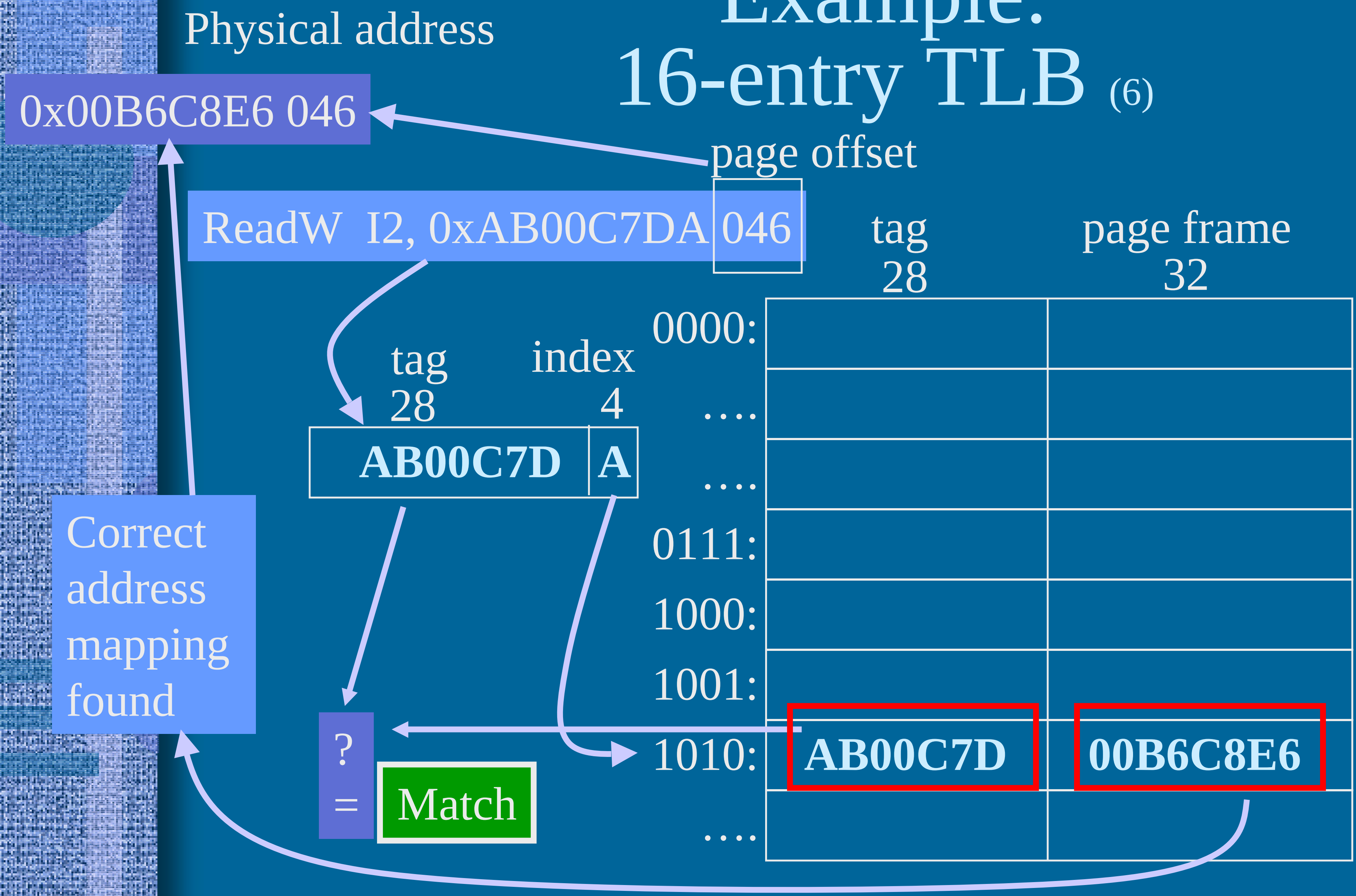
Fig. 8.18 (Fig. 7.19 [Stal99])

- "Hit" on TLB?
  - address translation is in TLB - real fast
- "Miss" on TLB?
  - must read page table entry from memory
  - takes time
  - cpu waits idle until it is done
- Just like normal cache, but for address mapping
  - implemented just like cache
  - instead of cache line data have physical address
  - split TLB? 1 or 2 levels?

Memory Organisation (3)



Example: 16-entry TLB (6)



TLB and Cache (3)

Fig. 8.19

(Fig. 7.20 [Stal99])

- Usually address translation first and then cache lookup
- Cache can be based on virtual addresses
  - can do TLB and cache lookup simultaneously
  - faster
- Implementations are very similar
  - TLB often fully associative
    - optimised for temporal locality (of course!)

TLB vs. Cache

TLB Miss

- CPU waits idling
- HW implementation
- Invisible to process
- Data is copied from memory to TLB
  - from page table data
  - from cache?
- Delay 4 (or 2 or 8?) clock cycles

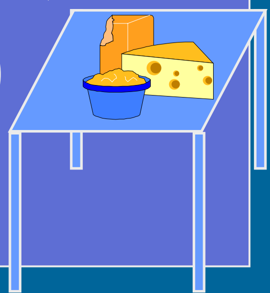
Cache Miss

- CPU waits idling
- HW implementation
- Invisible to process
- Data is copied from memory to cache
  - from page data
- Delay 4 (or 2 or 8?) clock cycles

TLB Misses vs. Page Faults

TLB Miss

- CPU waits idling
- HW implementation
- Data is copied from memory to TLB (or from cache)
- Delay 1-4 (?) clock cycles



Page Fault

- Process is suspended and cpu executes some other process
- SW implementation
- Data is copied from disk to memory
- Delay 30 ms (?)

Virtual Memory Policies (3)

- Fetch policy (noutopolitiikka)
  - demand paging: fetch page only when needed 1st time
  - working set: keep all needed pages in memory
  - prefetch: guess and start fetch early
- Placement policy (sijoituspolitiikka)
  - any frame for paged VM
- Replacement policy (poistopolitiikka)
  - local, consider pages just for this process for replacement
  - global, consider also pages for all other processes
  - dirty pages must be written to disk (likaiset, muutetut)

Page Replacement Policy (2)

- Implemented in SW
- HW support
  - extra bits in each page frame
  - M = Modified
  - R = Referenced
    - set (to 1) with each reference to frame
    - reset (to 0) every now and then
      - special (privileged) instruction from OS
      - automatically (E.g., every 10 ms)
  - Other counters?

Page Replacement Policies (6)

(sivunpoisto-algoritmit)

- OPT - optimal
- NRU - not recently used
- FIFO - first in first out
  - 2nd chance
  - clock
- Random
- LRU - least recently used
  - complex counter needed
- NFU - not frequently used

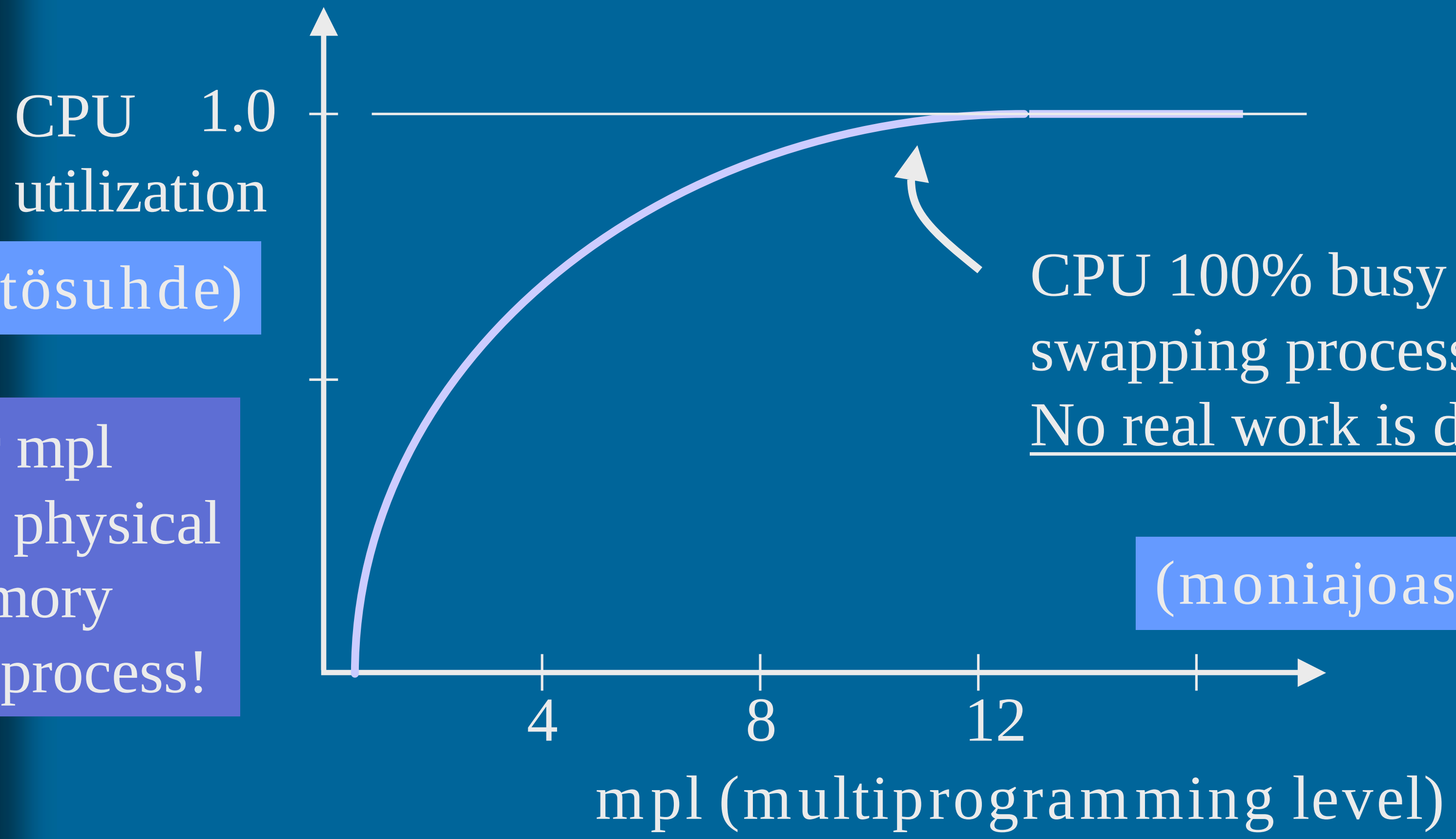
OS
Virtual Memory
Management

Thrashing

- Too high mpl
- Too few page frames per process
  - E.g., only 1000? 2000?
  - Less than its working set
- Once a process is scheduled, it will very soon reference a page not in memory
  - page fault
  - process switch

Trashing

(ruuhkautuminen)



(käyttösuhde)

CPU 100% busy
swapping processes!
No real work is done!

Higher mpl
⇒ less physical memory per process!

- How much memory per process?
- How much memory is needed?

Page Fault Frequency (PFF) Dynamic Memory Allocation

- Two bounds: L=Lower and U=Upper
- Physical memory split into fixed size pages
- At every page fault
  - T=Time since previous page fault
  - if $T<L$ then give process more memory
    - 1 page frame? 4 page frames?
  - if $U<T$ then take some memory away
    - 1 page frame?
  - if $L<T<U$ then keep current allocation

Multi-level paging/segmentation

- Segmented paging
  - address logically split into segments and then physically into pages
  - protection may be at segment level

01101 01100110 10110000

segm page byte offset

- Multiple level paging
  - large address space may result in very large page tables
  - solution: multiple levels of page tables

Fig. 5.43 [HePa96]

  - VM implementation may not utilize them all
  - VM implementation may seem to use more levels than there are (e.g., Linux 3 levels on 2-level Intel arch.)
    - nr of actual levels in mem. management macros

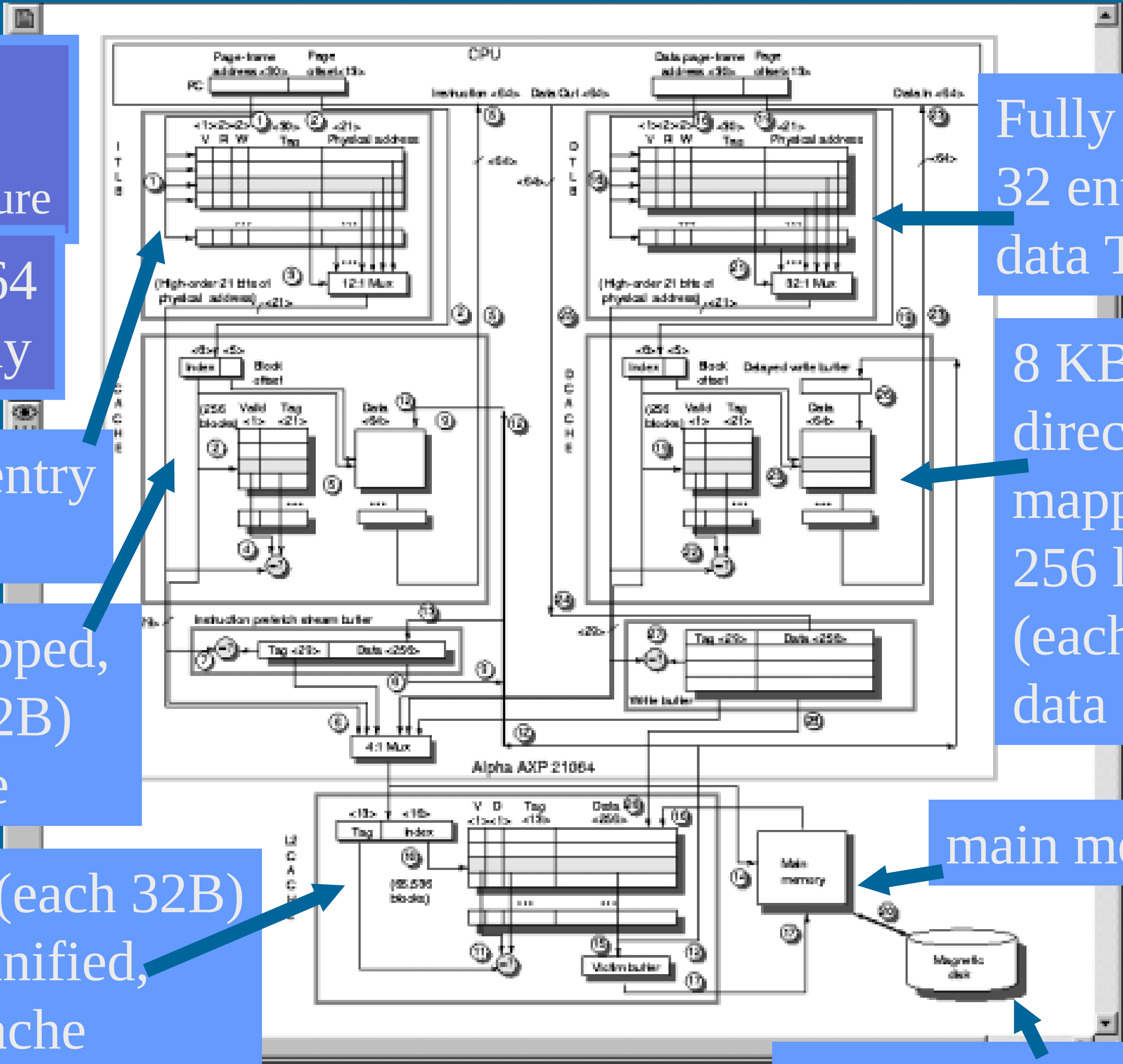
VM Summary

- How to partition memory?
  - Static or dynamic size (amount)
- How to allocate memory
  - Static or dynamic location
- Address mapping
- HW help (TLB) for address translation
  - before or concurrently with cache access?
- VM policies
  - fetch, placement, replacement

-- End of Chapter 8.3:Virtual Memory --

Fig. 5.47 from Hennessy-Patterson, Computer Architecture

Alpha AXP 21064 memory hierarchy



Fully assoc, 12 entry instruction TLB

8 KB, direct mapped, 256 line (each 32B) instruction cache

2 MB, 64K line (each 32B) direct mapped, unified, write-back L2 cache

Fully assoc, 32 entry data TLB

8 KB, direct mapped, 256 line (each 32B) data cache

main memory

paging disk (dma)