System Architecture

18 Virtual Memory (1)

Motivation, Concept, Paging

January 19 2009

Winter Term 2008/09

Gerd Liefländer

Agenda

- Review Address Space
- Motivation
- Concept and Implication
- Paging
  - Address Space Layout
  - Page Tables
    - Linear
    - Multi-level
    - Inverted
    - Virtual
  - Transaction Look-Aside Buffer (TLB)
  - Super Pages

Review Address Space (AS)

- Program is executed within a logical AS (LAS)
    - global and local data have (logical) addresses
    - instructions have (logical) addresses, e.g. the target address of a jump is a logical address
- Multiprogramming implies n ≥ 1 LAS
    - each application program has its own LAS
        - ... implemented via software and
        - ... supported additionally by HW (MMU)
    - OS kernel has its own KAS

Review: Valid & Invalid Addresses

- Most LAS are only sparsely filled, i.e. they have many "address space holes"
- Accessing an invalid logical address → exception
- A valid logical address references a valid program entity, e.g.
  - instruction: MOVE, ADDI, …
  - simple data: byte, word, pointer, …
  - compound data: struct, union, array, object, …

Addres Space & Address Region

AS is unit of protection at a coarse granularity

- Threads of AS_1 can not corrupt data in another AS_2 (unless sharing is explicitly allowed)

AR is unit of protection at a finer granularity

- An AR has specific access rights

How does the HW help to implement ARs?

- explicitly via the segment concept
- implicitly via an extra kernel data structure mapping a set of pages, called a region

focus

Address-Space Management

Traditional kernels establish and manage ASes and do their bookkeeping

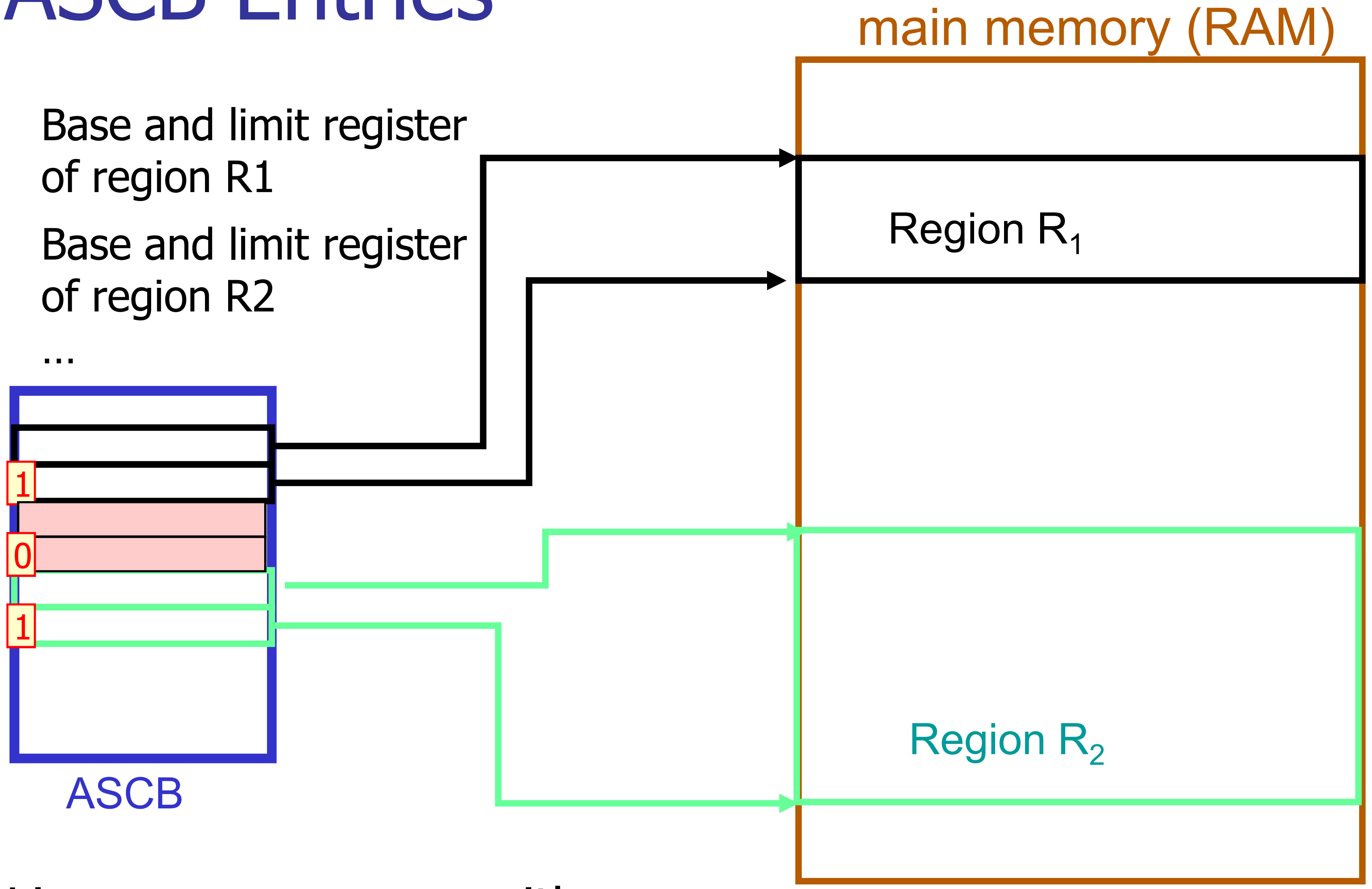
Where?

Analogous to bookkeeping of threads the kernel holds an ASCB (or PCB or TaskCB) to establish an AS (or process or task)

Typical content of an ASCB?

ASCB Entries*

- Base and limit register of region R1
- Base and limit register of region R2
- …



*In Linux ∃ `mm_struct` with `vm_areas`

Motivation VM

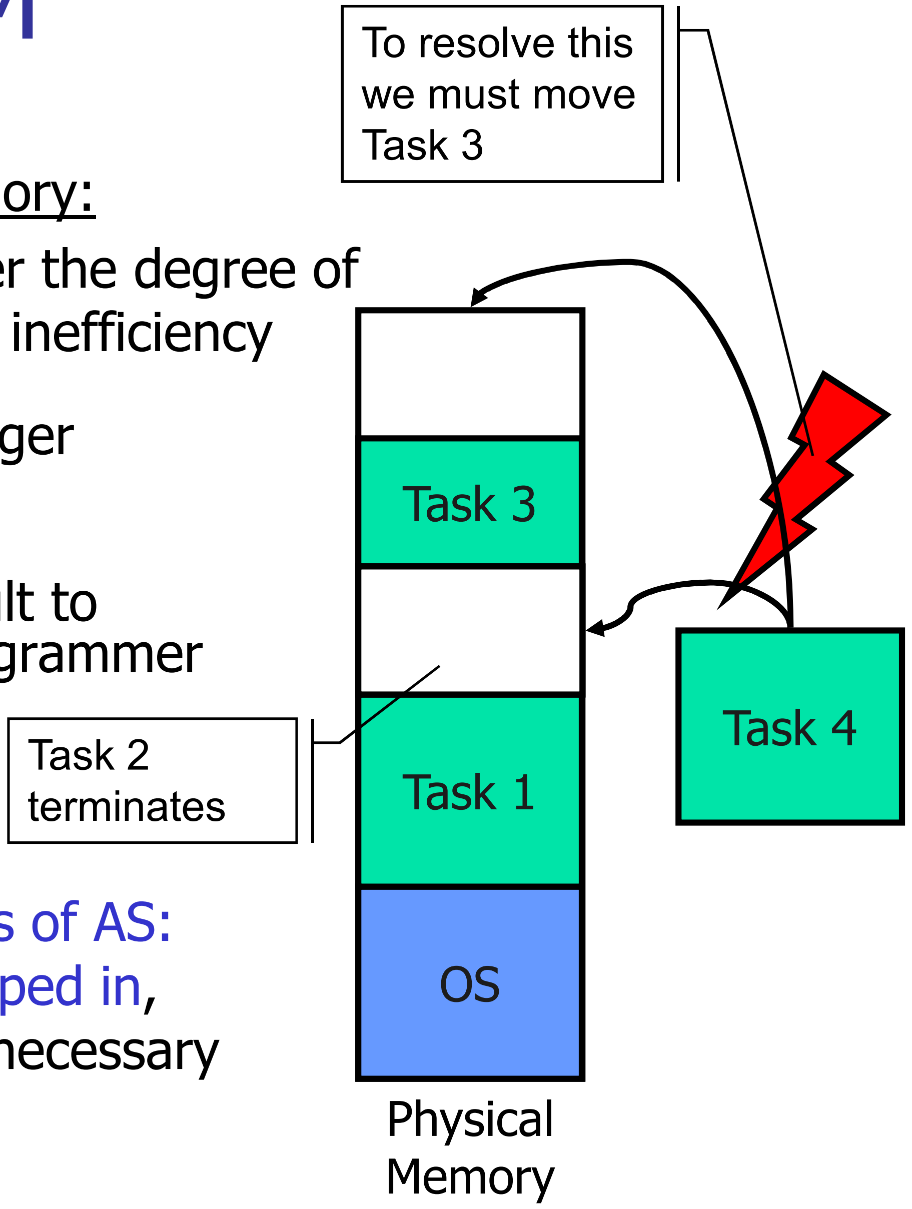
Any time you see the term “virtual”
think of an additional level of indirection
Program uses virtual (logical) addresses

Virtual addresses are converted to physical addresses
Physical address indicate real location of program entity
Physical location can be RAM or disk

Motivation for VM*

Shortcomings of non-virtual memory:

1. Completely mapped ASs lower the degree of multiprogramming ⇒ system inefficiency
2. What to do with ASs even larger than the physical memory?
3. Partly mapped ASs are difficult to handle for an application programmer (see overlay technique)



Goal: Replace automatically parts of AS: the needed parts are swapped in, others are swapped out if necessary

Virtual Memory Principle

Key idea:

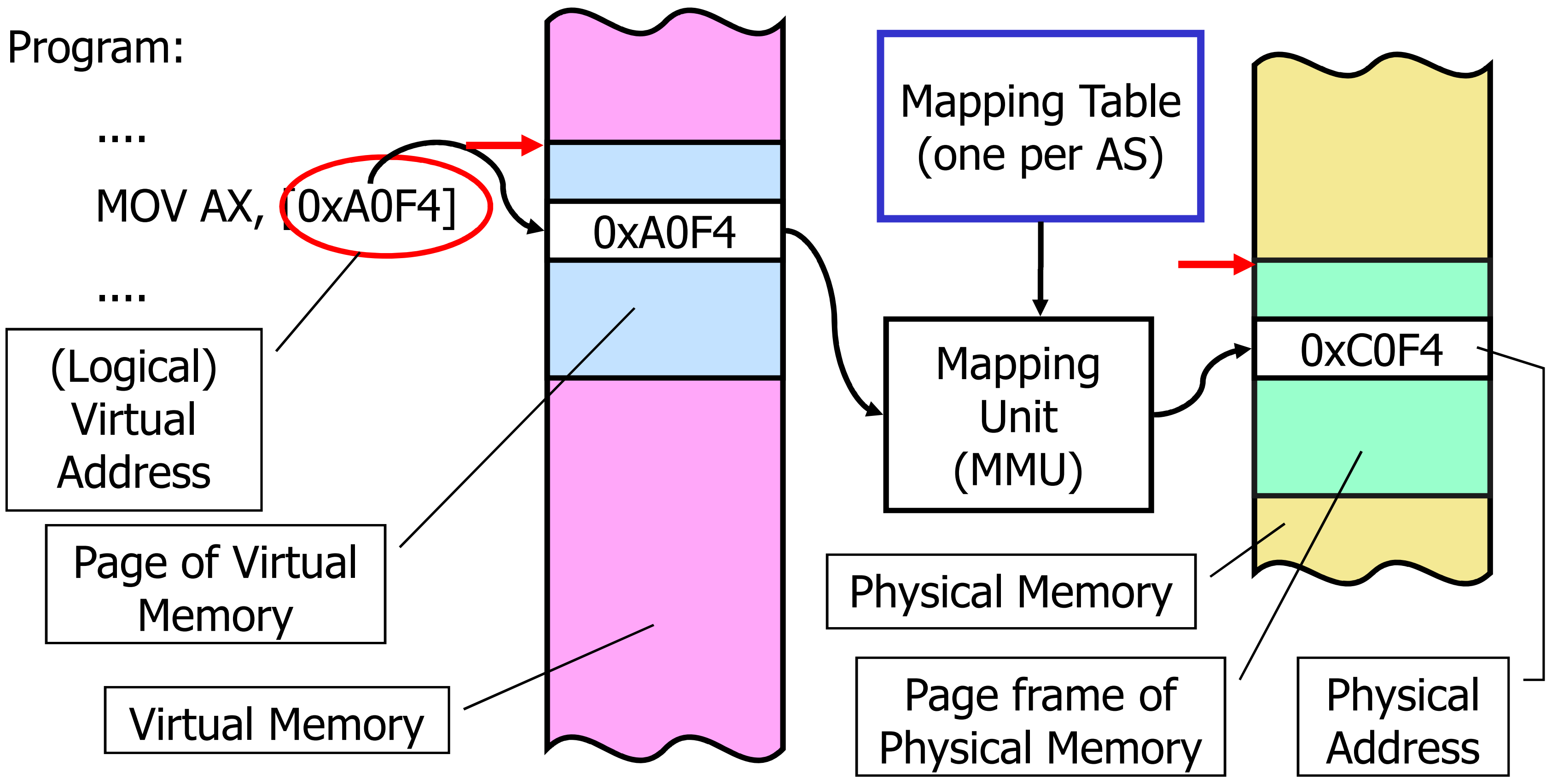
Instead of swapping a complete AS, in a VM system the OS automatically maps

- the currently needed LAS units (e.g. pages) to fitting memory units (e.g. page frames).

$\Rightarrow$

Application programmers are no longer involved in overlaying, a concept that was really cumbersome, error-prone, and thus hard to maintain

Virtual Memory?



Open Question: *How can we tell the HW that the **blue page** has been mapped to the **green page frame**?*

Note: In *many* cases it does not matter where a page of VM is mapped to.

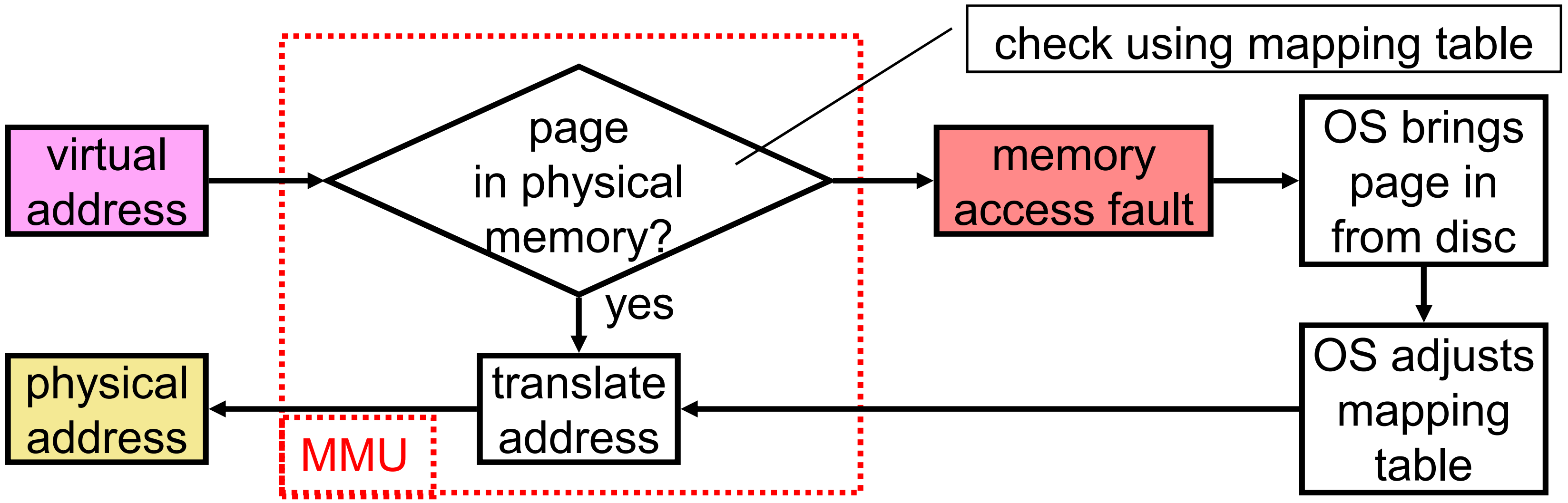
Mapping Process in Principle

Not every page of VM has to be present in memory:

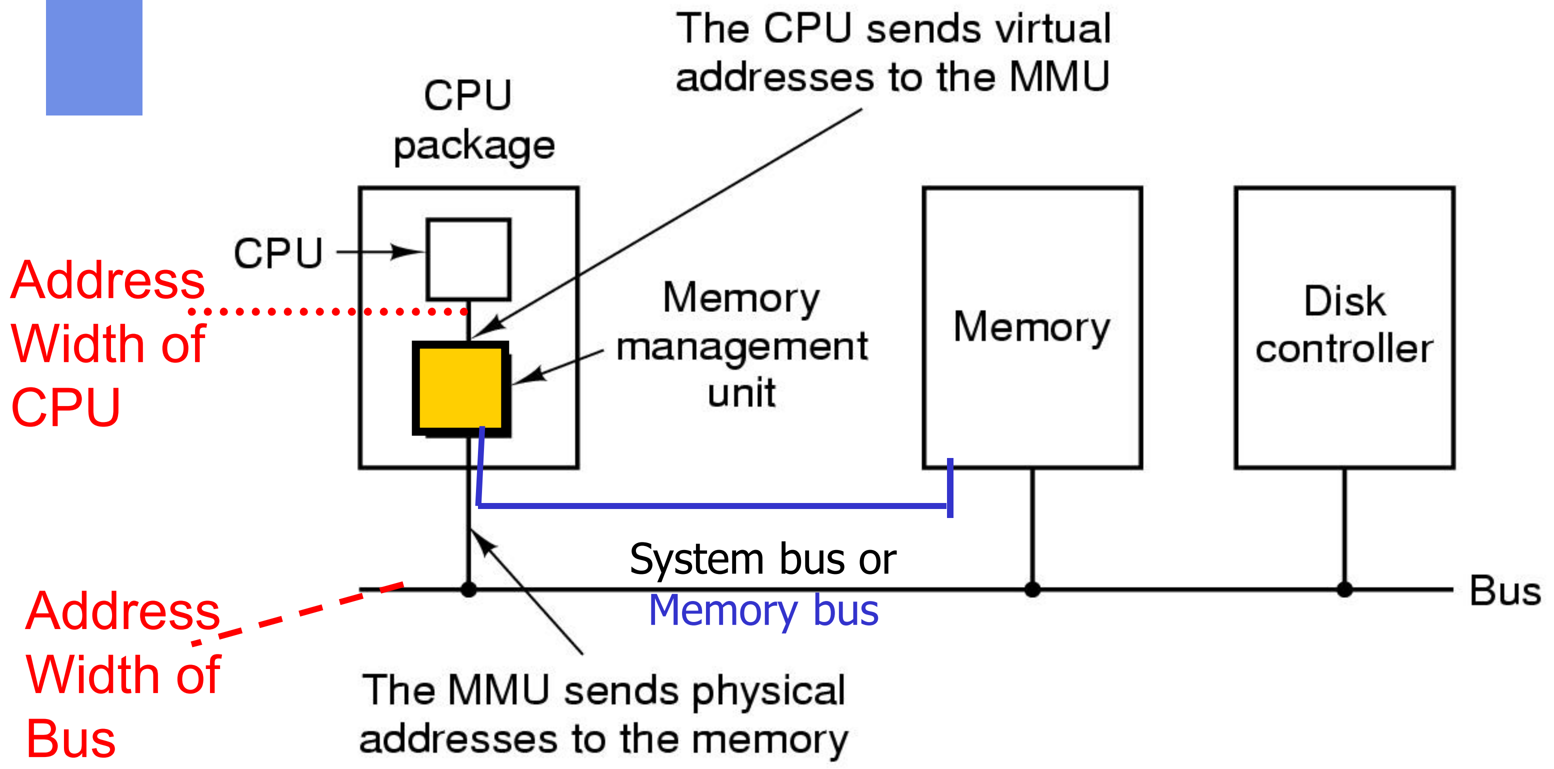
- Pages may be loaded (paged-in or swapped in) from disk when they are referenced
- No longer used or rarely used pages can be discarded (over-paged) or written out to disk (paged-out or swapped out)
  (→ paging or swapping)

Usually every task has its own "mapping table"

⇒ a virtual address space per task (LAS per task)

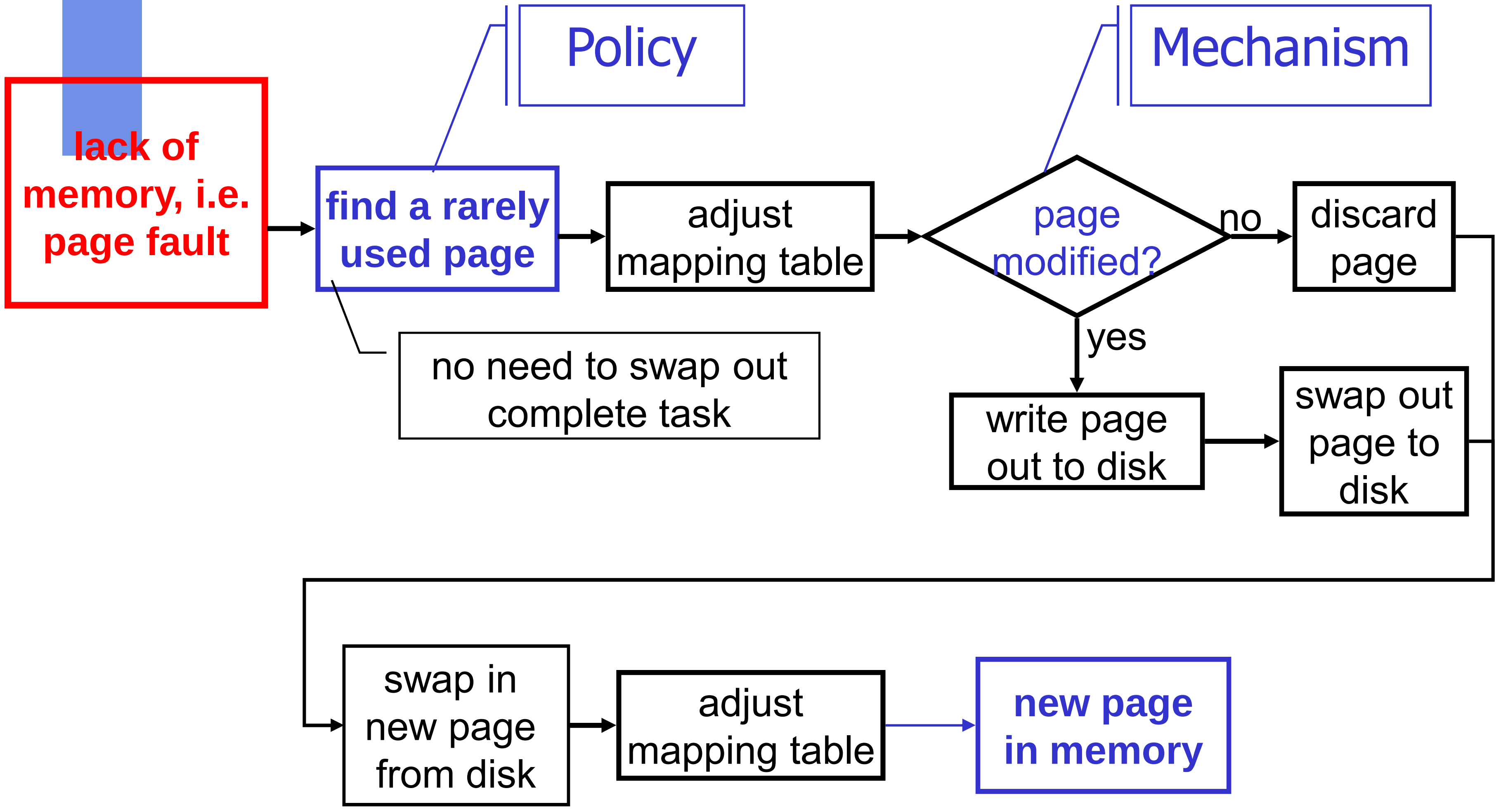


Memory Management Unit (MMU)



Location and function of the MMU

Principle of Paging



Terms of Virtual Memory (1)

- Virtual Memory (VM)
  - Not a physical device but an abstract concept
  - Comprises all current virtual address spaces
- Virtual Address Space (VAS) (per task)
  - Set of valid (defined or predefined) virtual addresses
- *(Single Address Space Systems (see Mungi at UNSW) use a single VAS for all tasks)*

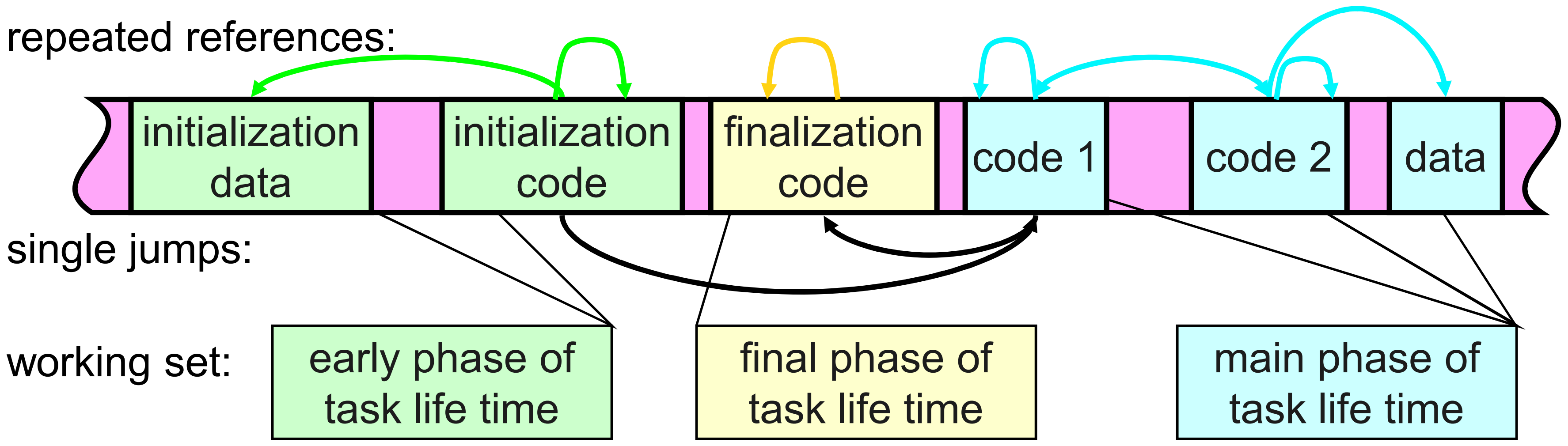
Terms of Virtual Memory (2)

- Resident Set (RS)
  - Set of pages of a task that are currently mapped to physical memory
- Working Set (WS)
  - Set of pages that a task currently really needs, i.e. those pages „should" be in main memory (precise definition later)

Principle of Locality

Memory references of a running activity tend to cluster.
Working set should be part of the resident set
to operate efficiently (otherwise: frequent page faults)

⇒ honor the principle of locality to achieve this



Terms of Paging

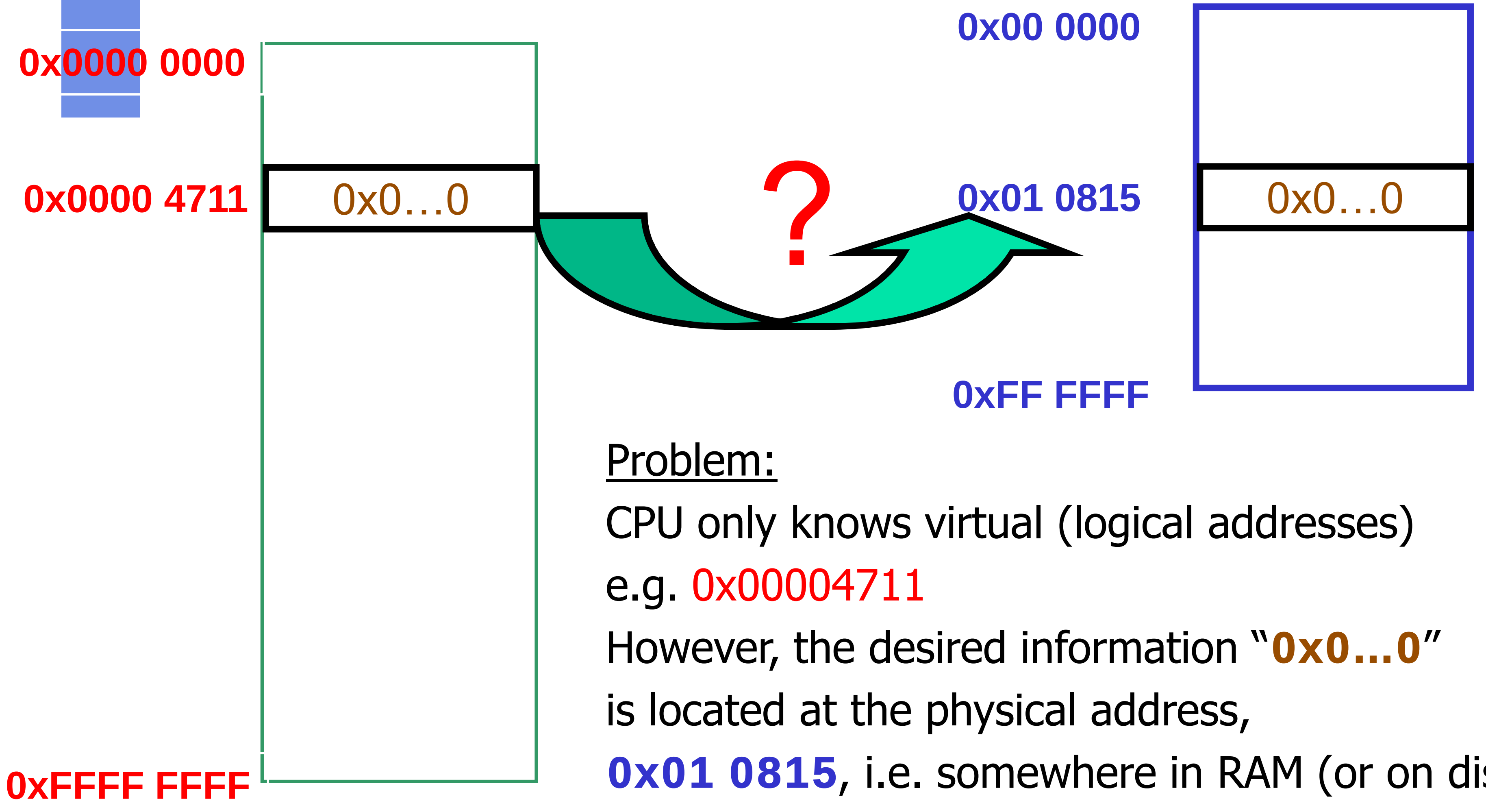
Definition:

Page =	2^i sized block* of virtual address space
Page frame =	2^i sized block of physical address space (=RAM)

typically $i \in [10, 16]$

*Modern systems offer more than one page size. *Why?*

Gap between AS and Memory



Problem:

CPU only knows virtual (logical addresses)

e.g. 0x00004711

However, the desired information "**0x0…0**"

is located at the physical address,

0x01 0815, i.e. somewhere in RAM (or on disk)

MMU contains Mapping Info

Observation:

Inside the MMU we need information how to map

a logical (virtual) address of the LAS

to a physical address of RAM,

where the desired information

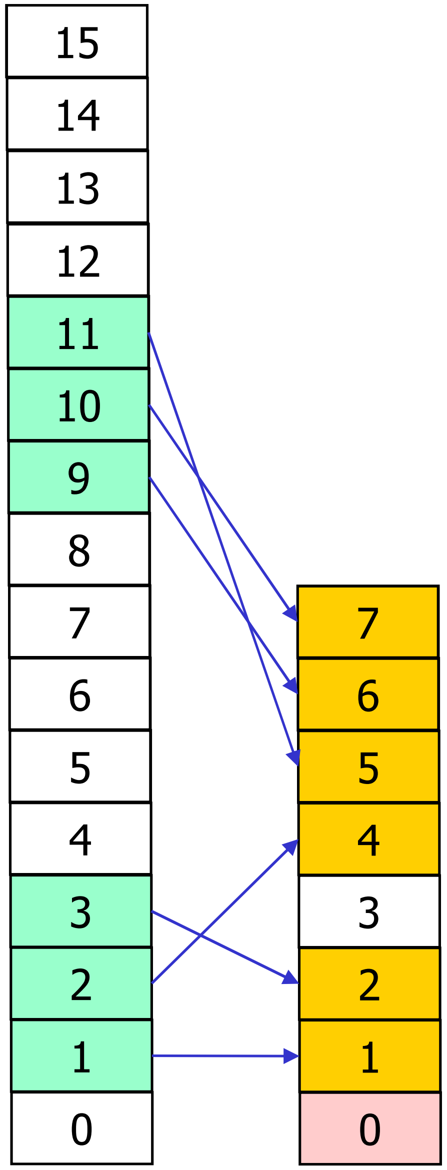
- code instruction
- stack entity
- global data entity

is actually located

Mapping

Virtual Memory

- Divided into equal sized pages
- Mapping is a translation between
  - Page and page frame
  - Page and Null
- Mappings are defined at **run time**
  - they can change
- AS can have holes
- Task/process does not have to be contiguous in physical memory
- Task might be mapped only partially

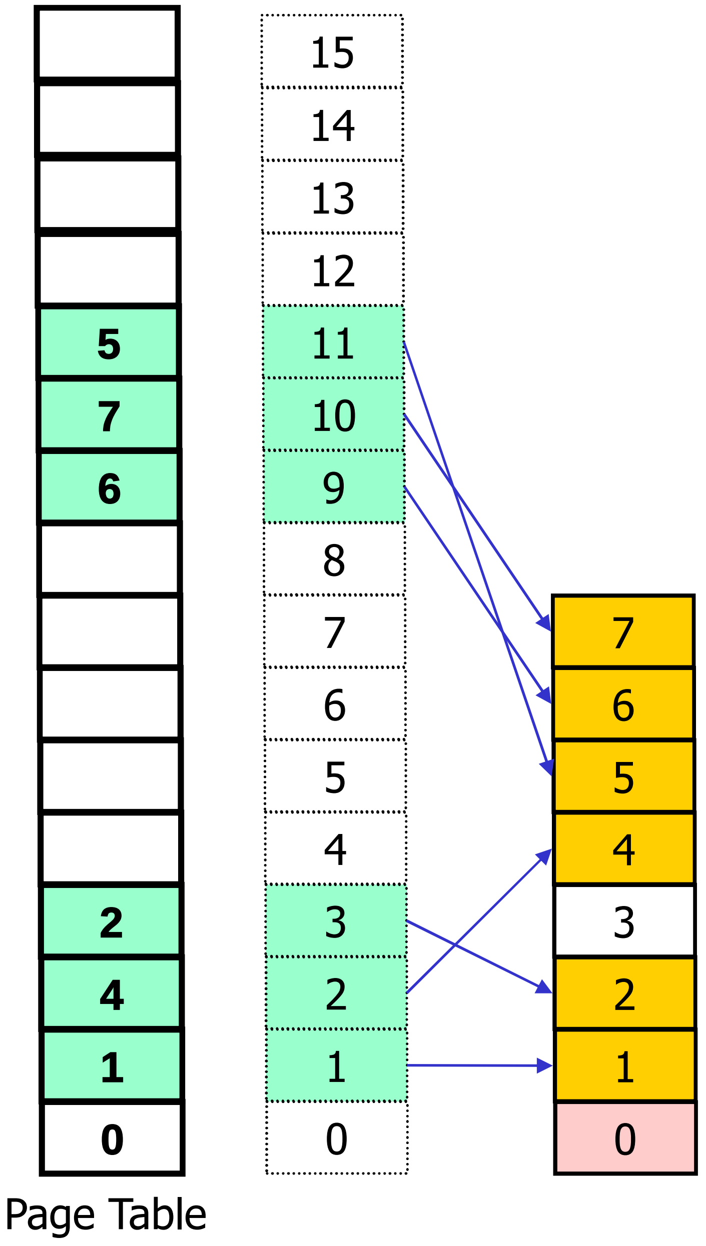


Physical Memory

- Divided into equal sized page frames
- Some page frames can be reserved for special purpose

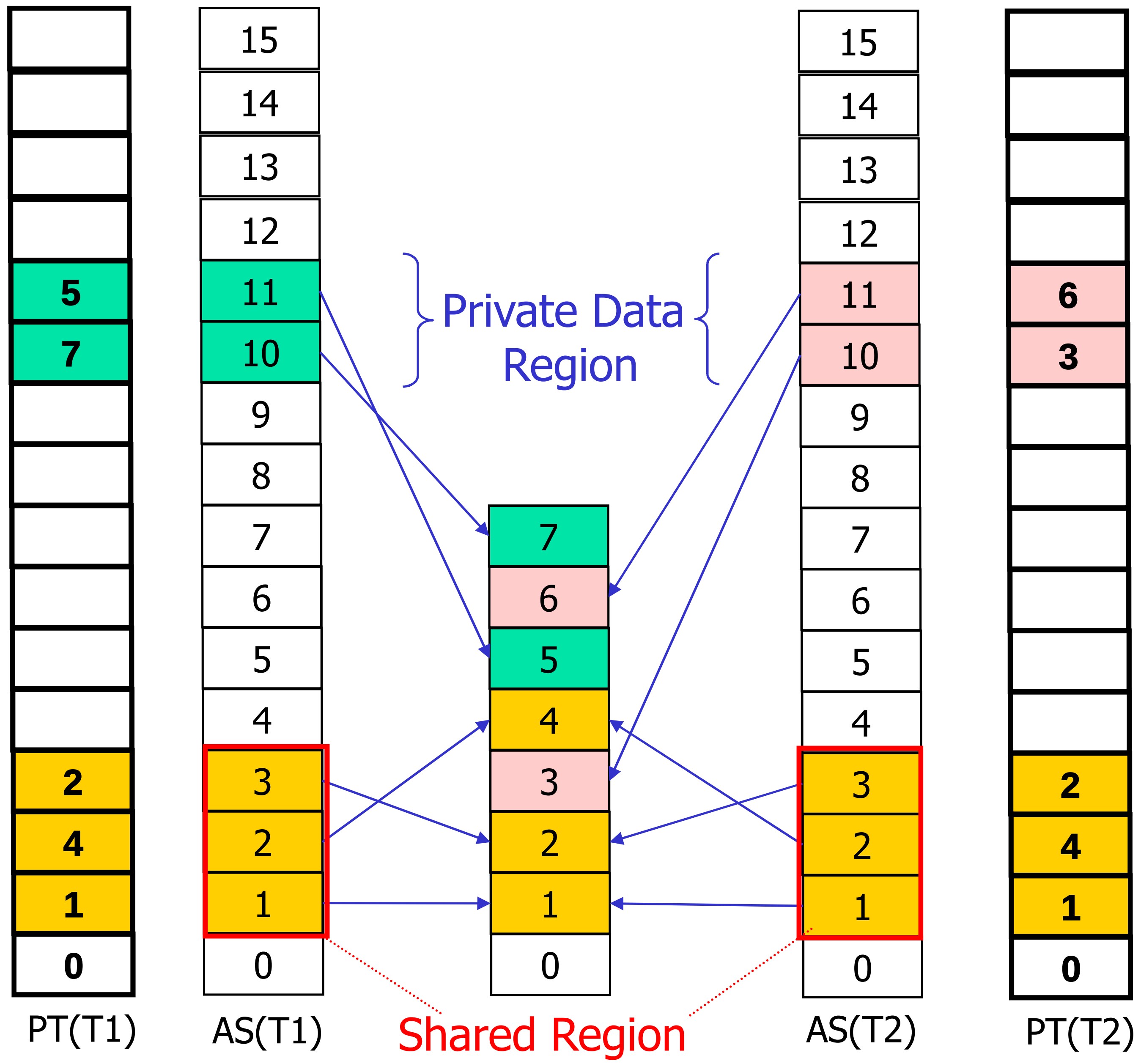
Page Table

- Page table contains at least the page frame numbers for all mapped pages to support address translation
- Page table can contain additional control information



Sharing Pages

- Private data/code
  - Each task has its own copy of data and code
  - Data and code can appear anywhere in the address space
- Shared code
  - Single copy of code shared between all tasks executing the same code
  - Code must be reentrant, i.e. not self modifying
  - Code must appear at the same address in all tasks



15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
5
7
2
4
1
0
6
3
Private Data Region
Shared Region
PT(T1)
AS(T1)
AS(T2)
PT(T2)

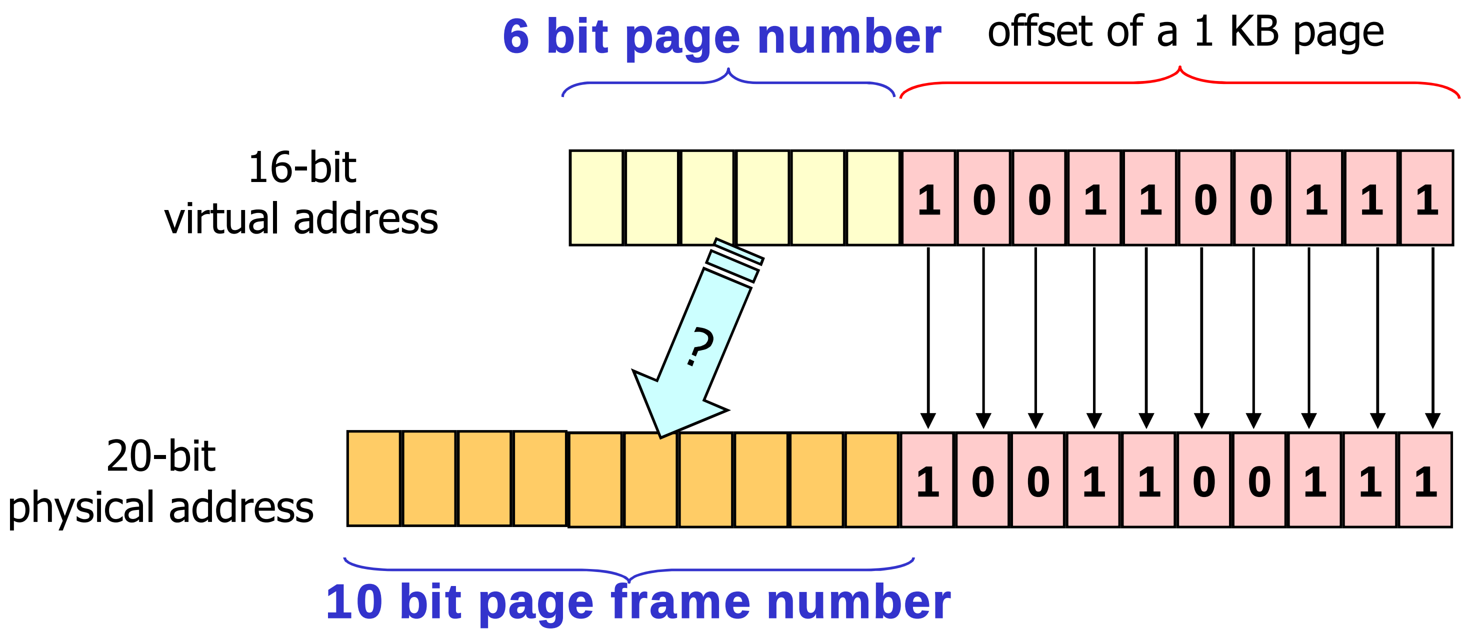
Address Translation

- Every (virtual) address issued by the CPU must be translated to a physical address of main memory
  - Every **`load & store`** instruction
  - Every operand of an **xyz**-operation
  - Every instruction **fetch**

⇒ needs HW support, otherwise too slow

- Address translation involves replacing page no by page frame no (in paging systems)

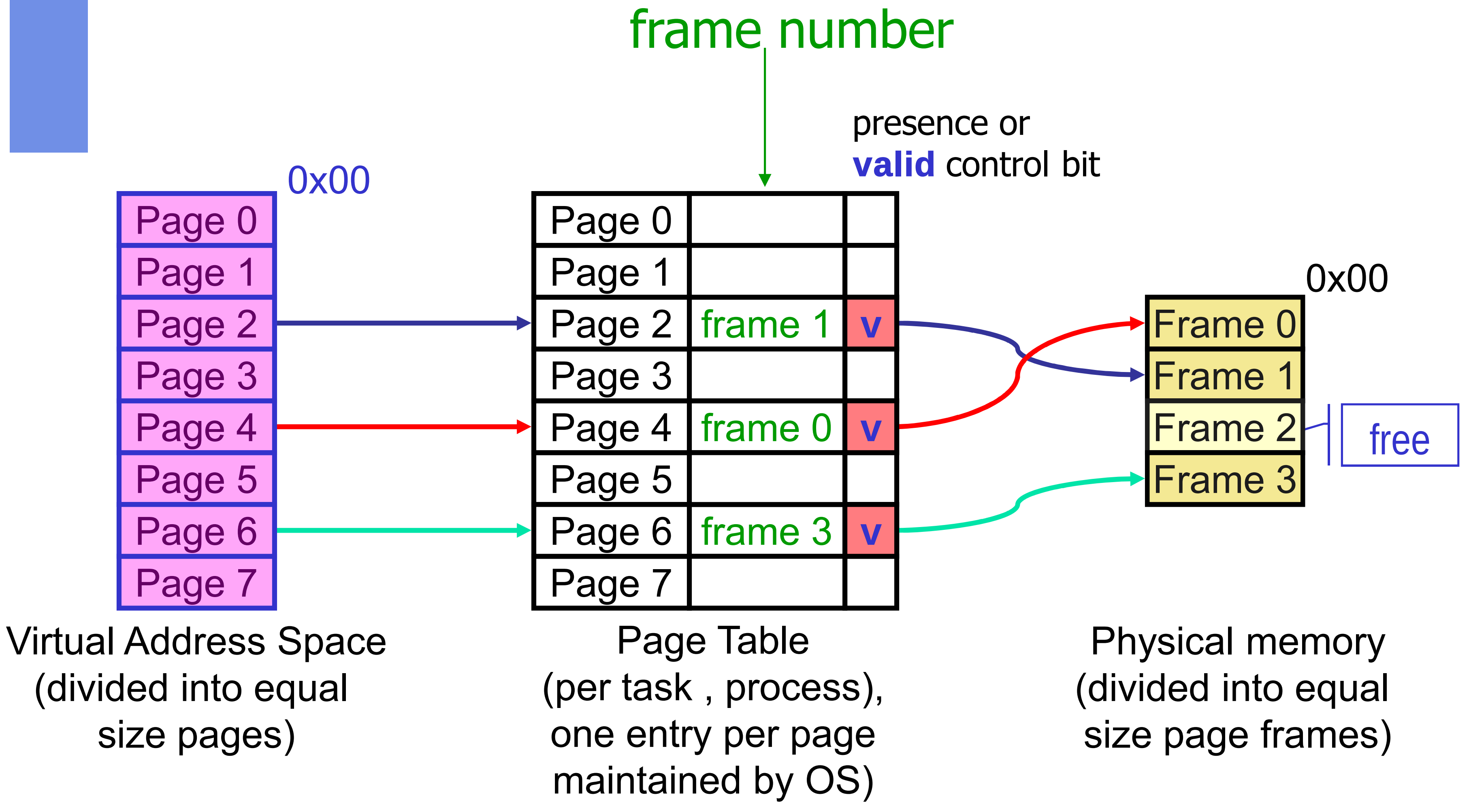
Easy: Page Number Substitution



Potential mapping of a 16-bit virtual address to a 20-Bit physical address

Hint: Typically the virtual AS is larger than the physical AS.
However, some early control systems used the above scheme

Usual Paging via Page Tables



Page Table Structure

- Page table = array of PTEs
- Hint: Adapt page table size to standard page size
- PTE often contains additional control bits
- Page table of current running task must be in main memory (~ role as the base register did in a non-virtual memory system)
- A single register holds the physical starting address of the page table of the currently running task

Control Bits of Page Table (1)

- Valid* bit: whether the page is mapped or not
- If it is in main memory, the PTE holds the frame number of this page in main memory
- If it is not in main memory, the PTE contains
  - either the "background" address of that page on disk, e.g. offset
    - in executable file or
    - in swap area
  - or nothing, because this information is stored somewhere else (where?)

*Again not the best term (mapped or presence bit)

Control Bits (2)

- Modified bit (dirty bit): whether the page has been modified since the page has been swapped in
  - If no changes have been made, this page is still identical with the corresponding image on the disk
- Defined bit: whether page belongs to LAS
- Pinned bit: whether page is resident
- Read-only/read-write bit: how to access the page
- Protection level bit: kernel or user page

Role of Defined Bit

Possible reaction to a reference on a non-defined page:

1.) Abort thread and/or task due to address violation

2.) Extend the surrounding region if the surrounding region has been specified as “extensible”

Example: Page Table Entry

r	w	x	v	re	m	s	c	su	tid	p	other	Page Frame #

r	read	s	shared
w	write	c	don’t cache
x	execute	su	super-page
v	valid	tid	task/thread id
re	referenced	p	pinned
m/d	modified/dirty		

Questions: Who is authorized to access these control bits?
What is done by hardware, what is done by software?

Implementation of Page Tables

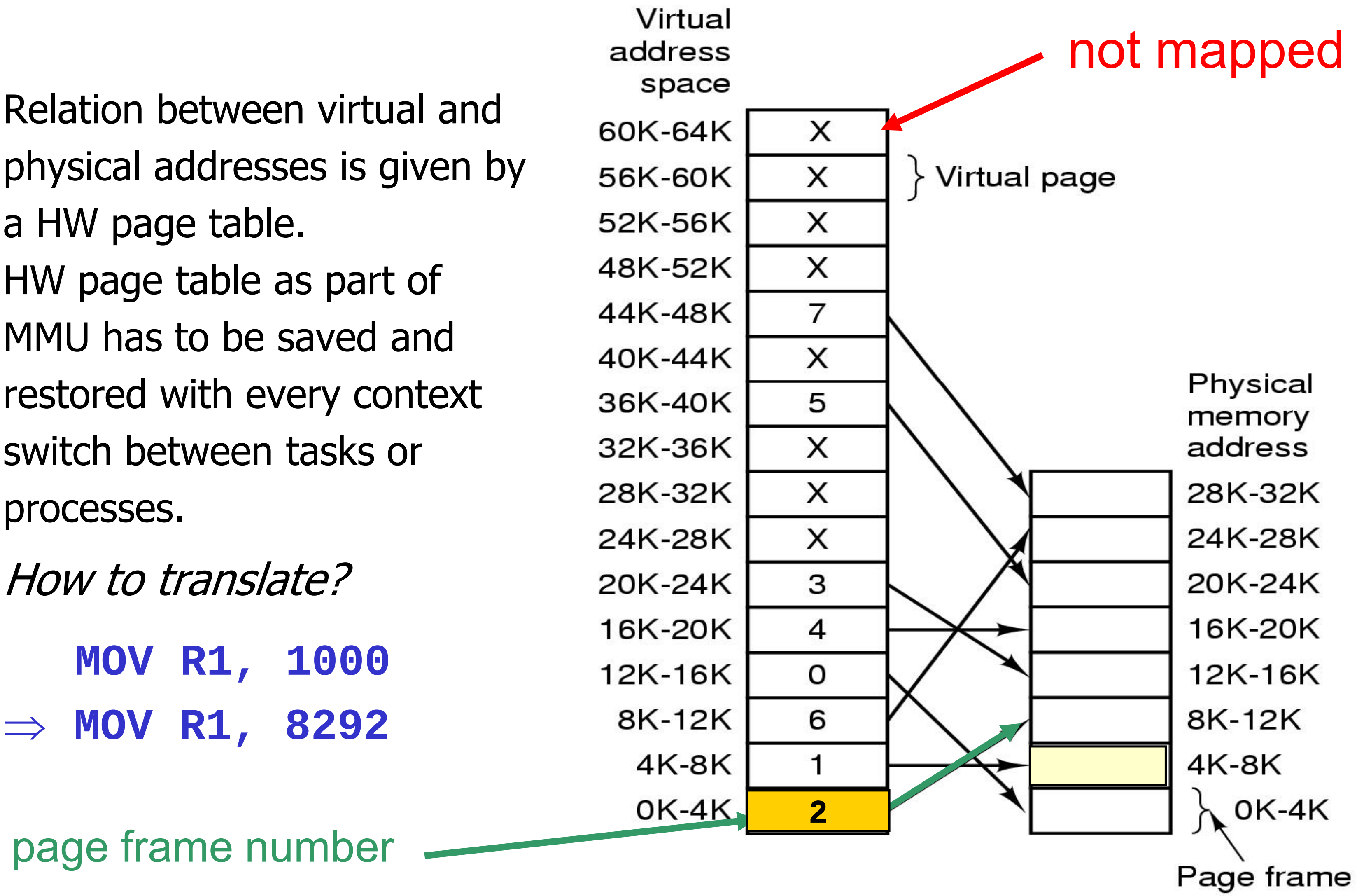
- Assume we have
  - 32-bit virtual address ($\rightarrow$ 4 GB AS)
  - Page size = 4 KB $\Rightarrow$
  - 2^{20} PTEs !!!!
- Problems
  - Size
    - Page table can be too large
  - Speed
    - Access must be fast, lookup for every memory reference
- *How and where to implement page table*
  - HW (e.g. part of MMU) or SW (as data structure in RAM)

Hardware Page Table

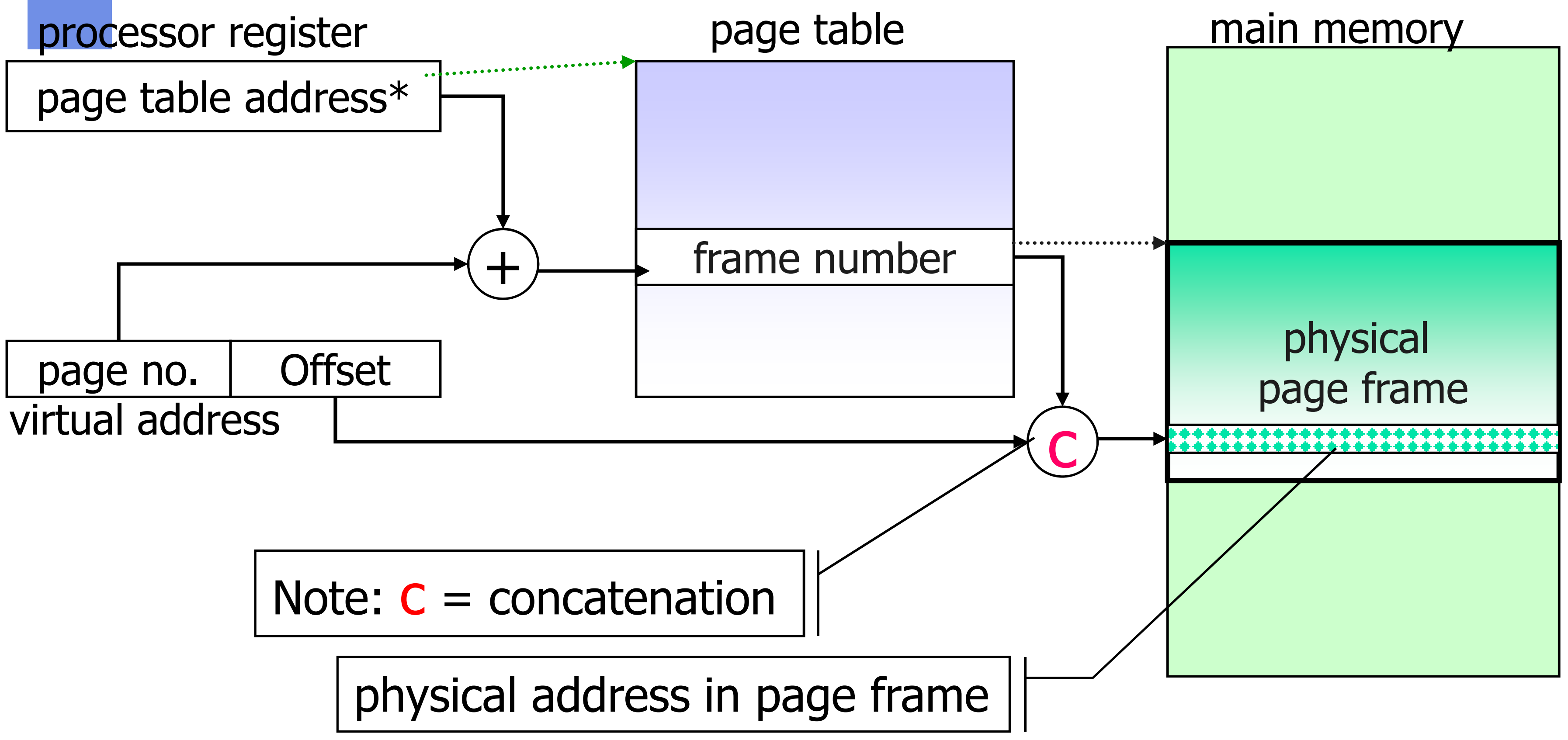
Relation between virtual and physical addresses is given by a HW page table.
HW page table as part of MMU has to be saved and restored with every context switch between tasks or processes.

How to translate?

```
   MOV R1, 1000
⇒  MOV R1, 8292
```



Address Translation Scheme



HW Page Tables Analysis

Drawbacks of a contiguous HW page table

In a multi-programming system we need per activated task/process a page table in RAM

Only the PT of current running process/task is mapped into the MMU

- Very expensive
- Not suited for large and sparsely occupied AS
- Huge overhead per task-/process switch, i.e. each time 2 copies between MMU and RAM

SW Page Tables Analysis

- Can slow down address translation too much
  - Has to be accessed per reference, i.e. suppose you have an instruction with 2 operands
  - *How many accesses to the page able do you need?*
- In HW you only need 1 register containing the start address of the current page table
- Task/process switch is much faster

<u>Conclusion:</u> Take the SW page table and try to speed up its slow address transformation

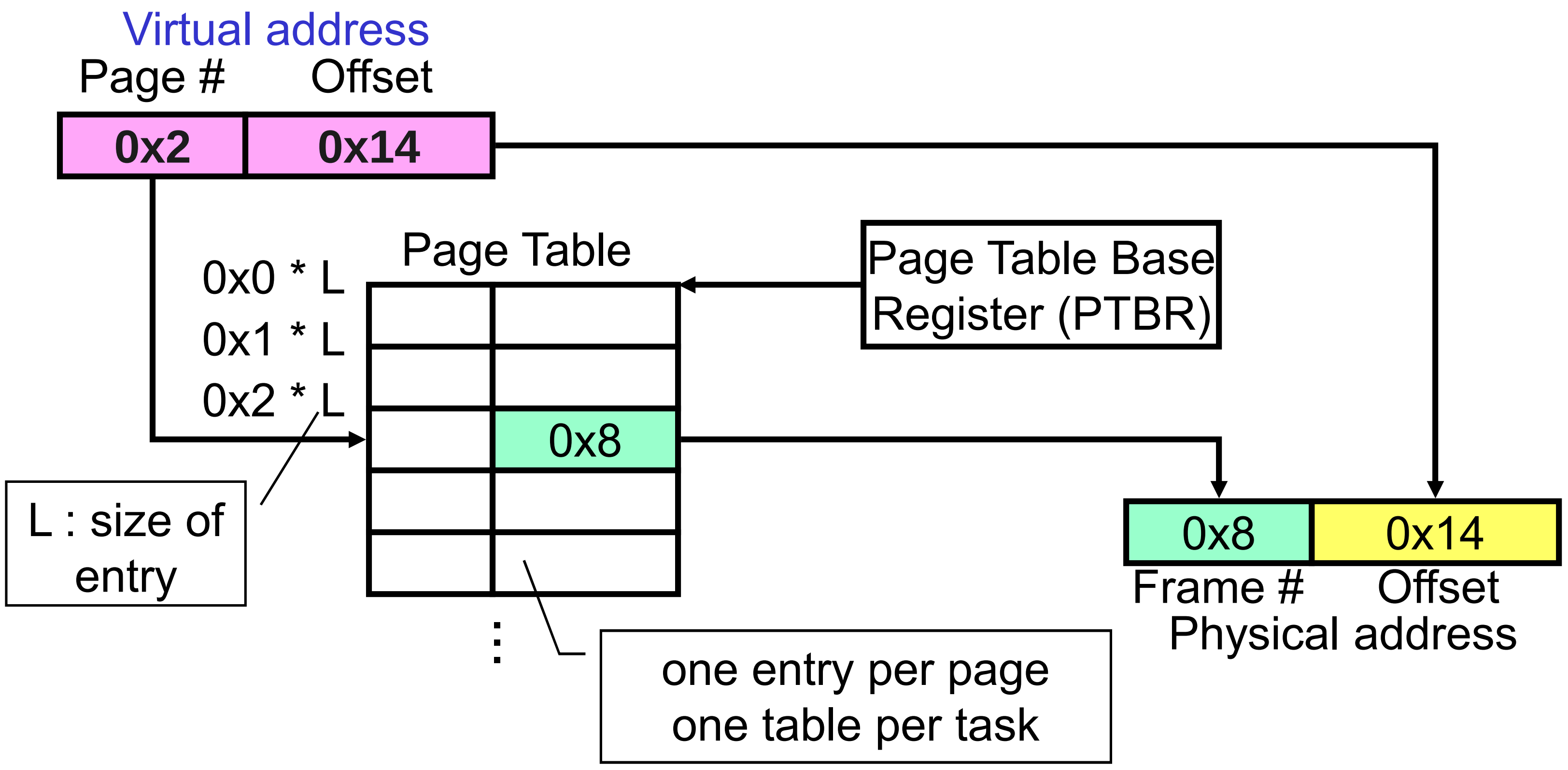
Page Tables for 32 Bit Machines

- Many tasks do not use the full a≤3 GB AS, e.g.
  - 0.1 – 1 MB code, 0.1 – 10 MB data, 0.1 MB stack
- We need a compact representation of this mapping situation that does not waste kernel space, but still guarantees fast lookups
- Three basic schemes:
  - Data structures reflecting the sparsity of a LAS
  - Data structures only representing resident pages
  - Use VM techniques for page tables

Implementation of Page Tables

- Multi-level Page Tables
  - Two-level
  - Three-level
  - …
- Inverted Page table
- Virtual Linear Page Table

Review: Linear Page Table



<u>Problem:</u>

Page tables can get very large, e.g. with a 32 bit AS & 4KB pages

$\Rightarrow 2^{20}$ entries per task $\Rightarrow$ **4MB** (if one PTE needs 4 bytes)

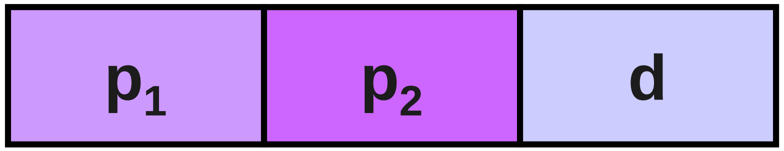
Multilevel Page Tables

Linear page table requires n pages to be stored

$\Rightarrow$ use a tree in order to

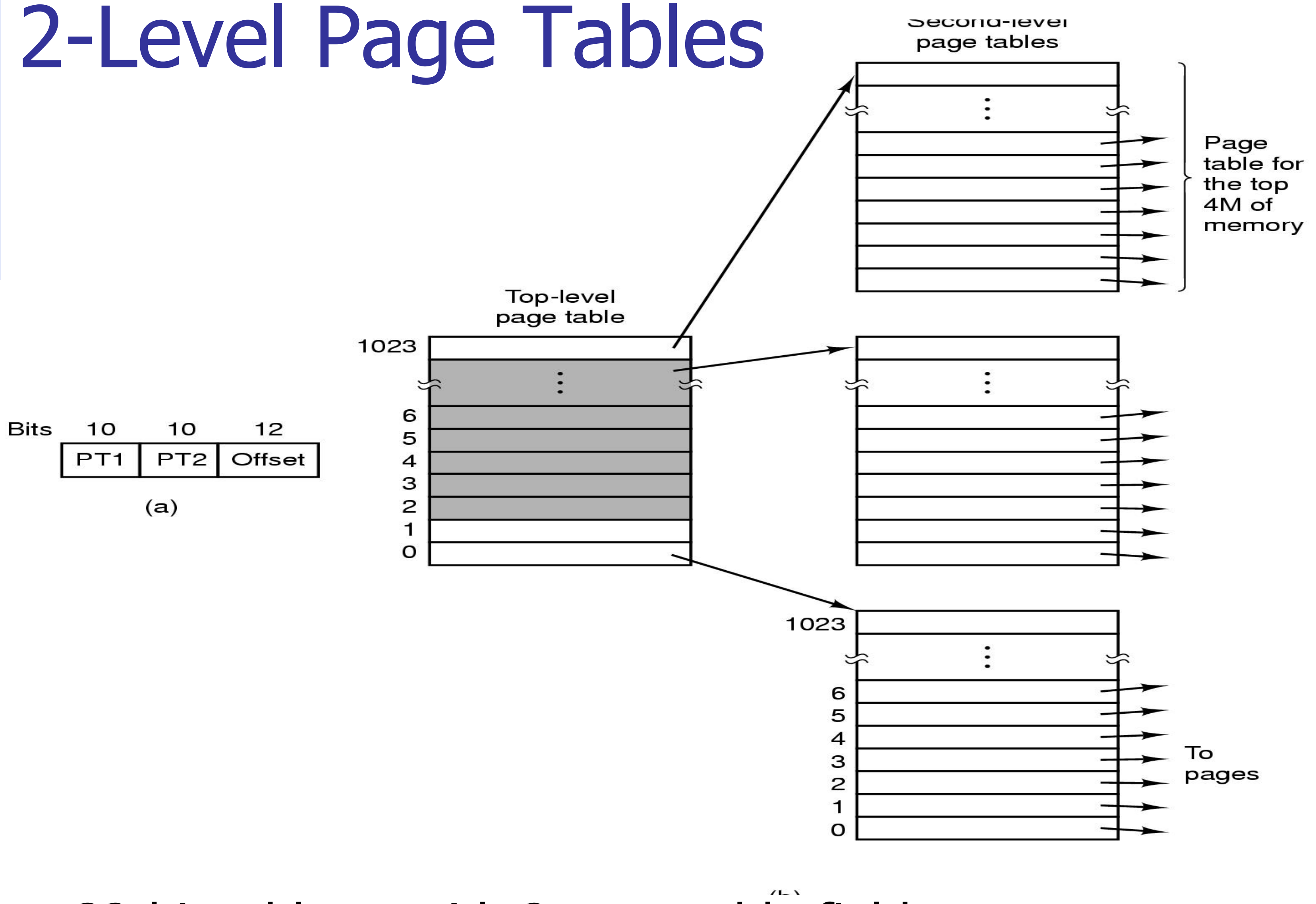
- cut down space requirements of the needed page tables of an AS (*in case of a sparse AS*)

- p_1 ~ an index to a page directory
- p_2 ~ an index to a page



d is the displacement (offset) within the page

2-Level Page Tables



- 32-bit address with 2 page table fields

Address Translation: 3-Level PT

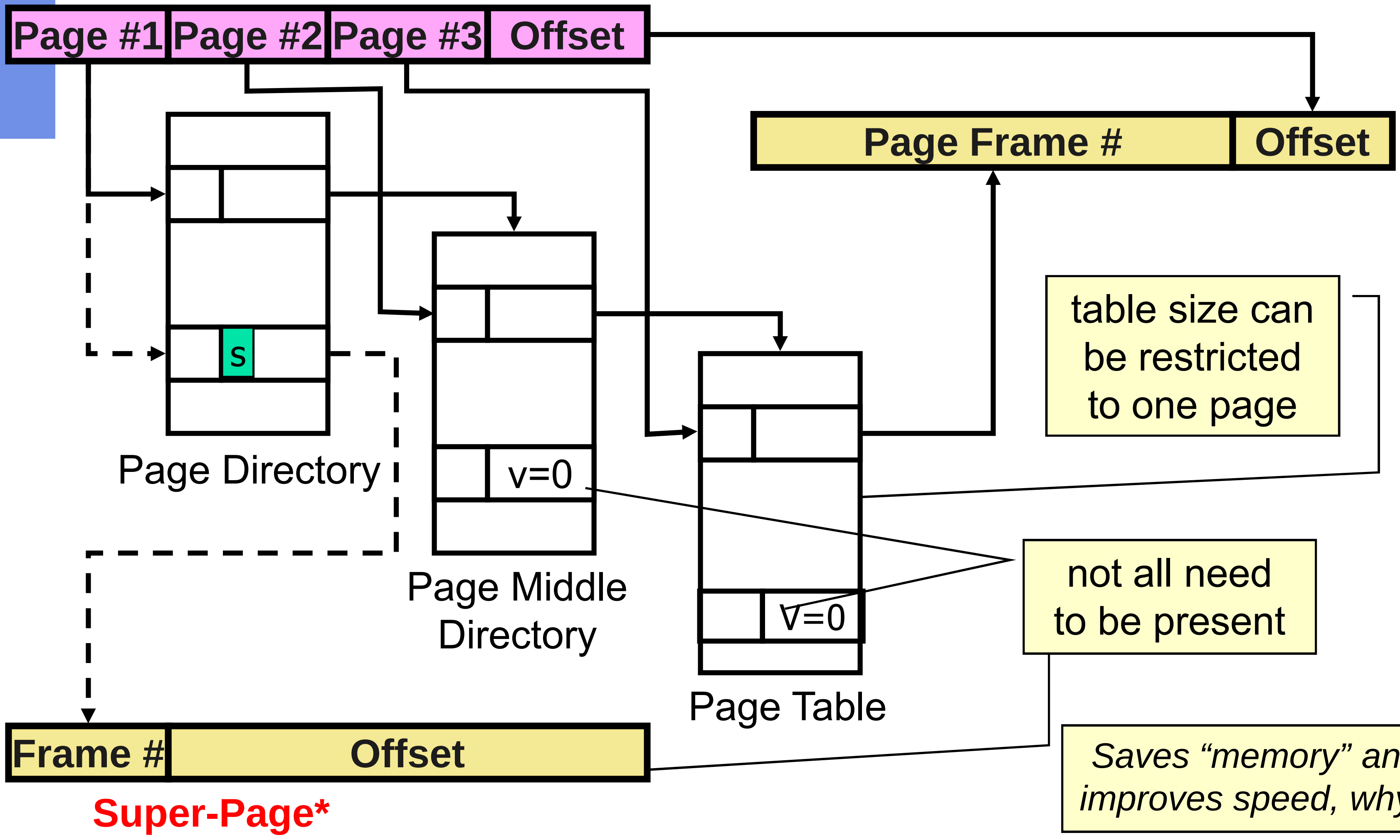


table size can be restricted to one page

not all need to be present

Saves "memory" and improves speed, why?

Analysis of Large AS

1. The larger the potential AS, the more levels are needed

   $\Rightarrow$ the slower the address translation

   $\Rightarrow$ further HW support needed ($\rightarrow$ TLB)

2. New concepts for implementing page tables

   (see: Talluri et al: "A New Page Table for 64- Bit Address Spaces", SOSP 1995 and Jan Oberländer: Seminar Talk on this Topic)

Inverted Page Table (1)

Instead of maintaining large page tables some machines (i.d. PowerPC, IBM Risk 6000) use

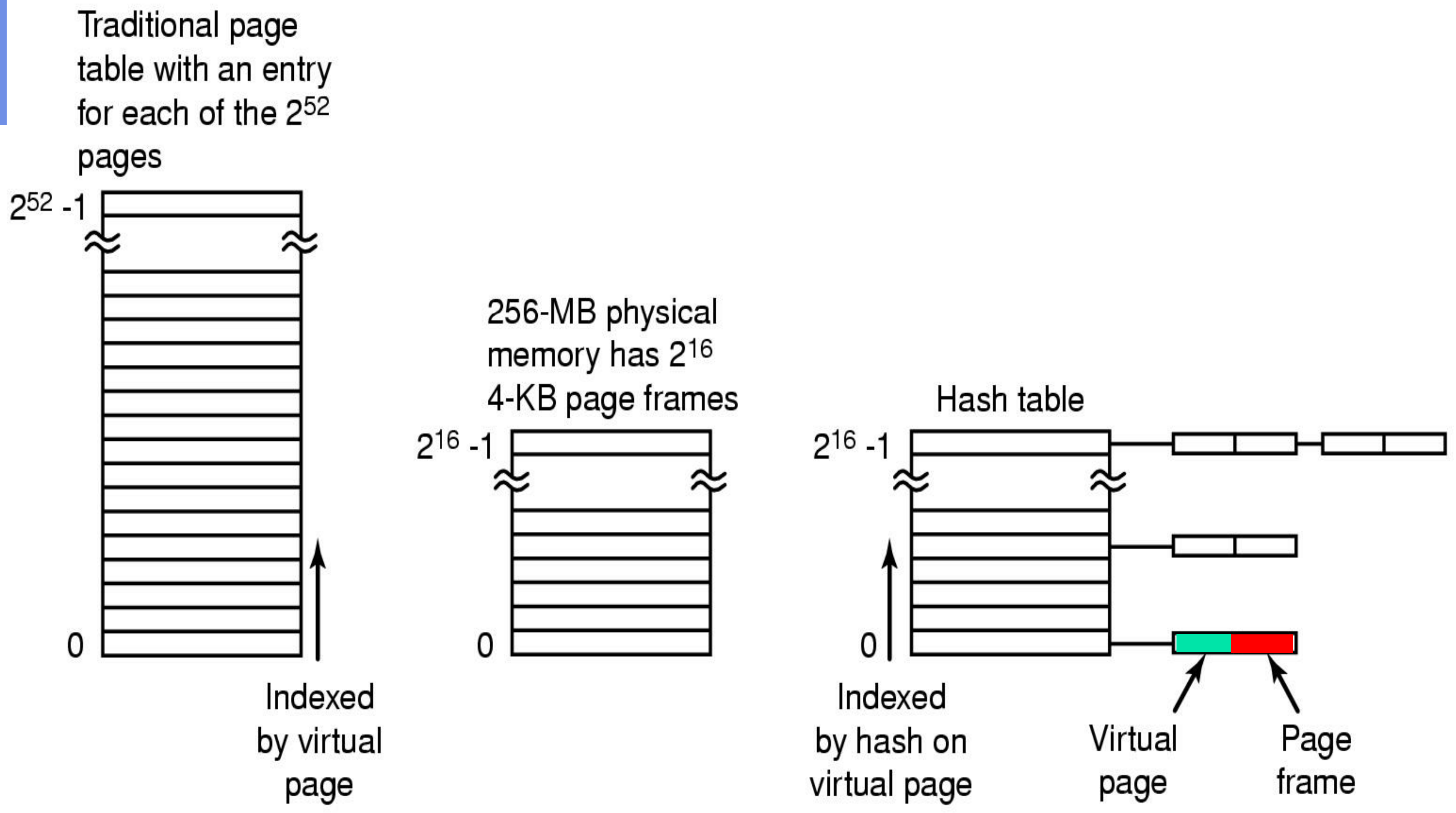
an Inverted Page Table (IPT)

Length of IPT depends on the number of page frames, i.e. space needed for the IPT is fixed

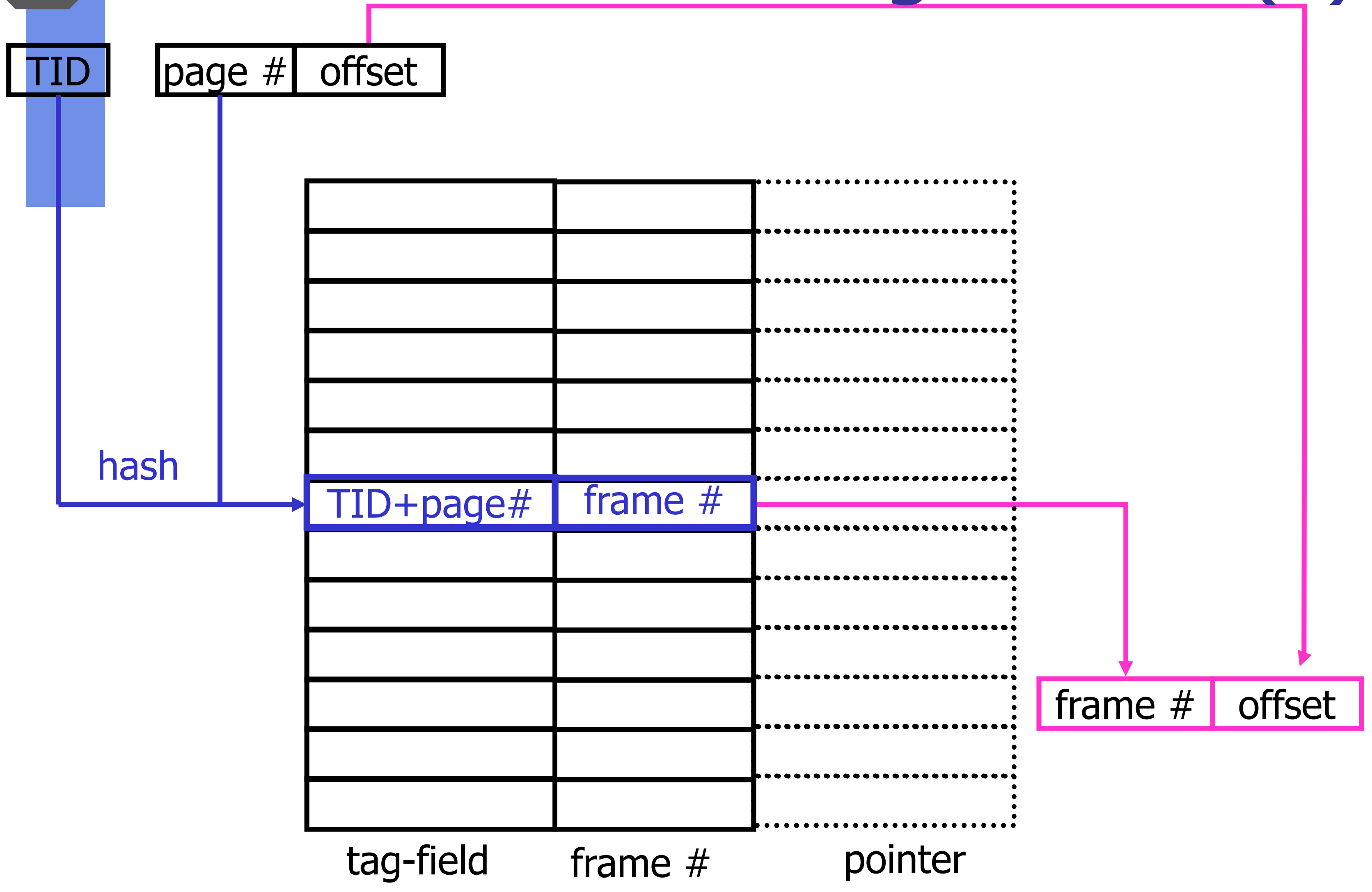
Question:

How to map virtual to physical addresses, i.e. how to find the appropriate pair of page number and page frame number?

Inverted *Hashed* Page Table (2)


Traditional page table with an entry for each of the 2^{52} pages
2^{52} -1
0
Indexed by virtual page
256-MB physical memory has 2^{16} 4-KB page frames
2^{16} -1
0
Hash table
2^{16} -1
0
Indexed by hash on virtual page
Virtual page
Page frame

Inverted Hashed Page Table (3)



Inverted Hashed Page Table (4)

Task ID + virtual page number is used
to look up in IPT to get desired frame number

For better performance, hashing is used to obtain a hash table entry which points to an IPT entry

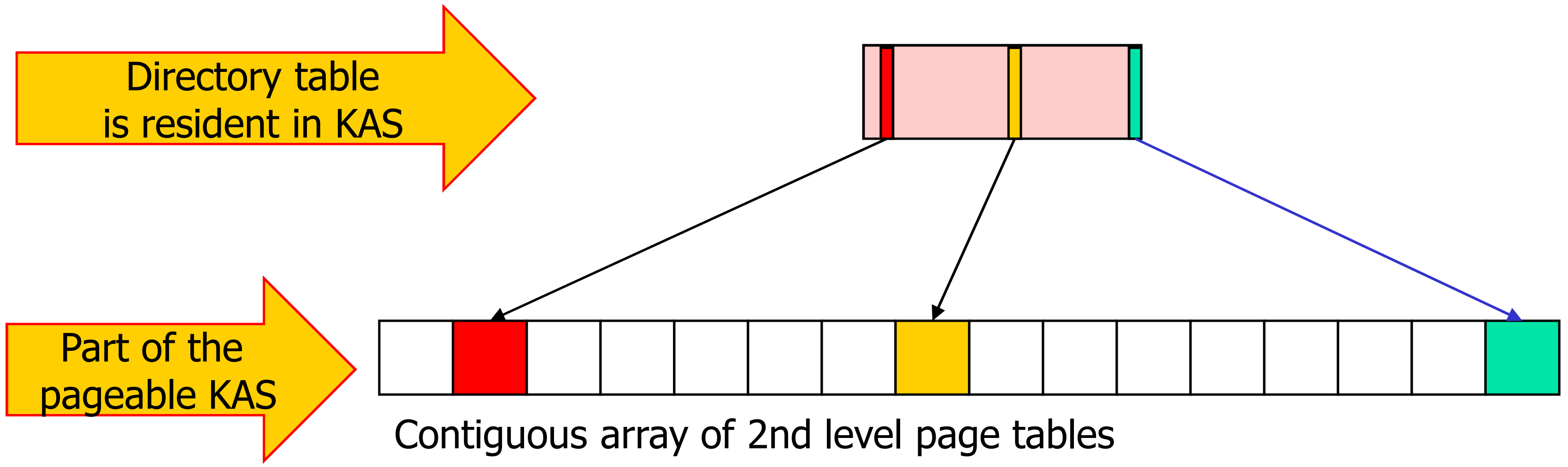
- A page fault occurs if no match is found
- Chaining is used to manage hashing conflicts

Used in some IBM and HP work stations

Problem:
Where and how to place information about pages currently not mapped?

Virtual Linear Page Table

- Assume a two-level page table structure
- Assume 2nd-level PT nodes in virtual memory
- Assume all 2nd-level PT nodes are allocated ⇒ 2nd-level PT nodes form a contiguous array indexed by page numbers


Directory table is resident in KAS
Part of the pageable KAS
Contiguous array of 2nd level page tables

Virtual Linear Page Table (2)

- Unused parts of 2nd-level page table unmapped
- Index into 2nd level page table without referring to page directory table
- Simply use full page number as PT index
- If reference is made to unmapped part of 2nd level page table, a secondary page fault is triggered

  $\Rightarrow$To load the missing 2nd level page table, use information of page directory table

Virtual Linear Page Table (3)

- Observation:
    - Each virtual memory reference can cause two physical memory accesses
        - 1 to fetch the page table entry
        - 1 to fetch/store the data
- Solution (also used for MLPT and IPT):
    - Use a high-speed cache for currently used mapping info
    - Associative cache, called TLB

Translation Look Aside Buffer (1)

Given a virtual address va', the MMU examines the TLB

If corresponding PTE' is in TLB =TLB HIT, its frame number is retrieved ⇒ construct physical address pa

If PTE' is not found in the TLB =TLB MISS

⇒ lookup the related PT and fill TLB with required PTE iff page is mapped, otherwise raise a page fault

⇒ therefore TLB always contains the

“hottest” page table entries

Translation Look Aside Buffer (2)

- Page table = array of page frame numbers
- TLB holds a (recently used) subset of PTEs
  - Each TLB entry is identified (tagged) by the **page number**
  - Access is by associative lookup:
    - All TLB entries' tags are compared with page number
    - TLB is associative (or content-addressable) memory
    - TLB entry additionally contains some control bits

page number	frame number	v	w
…	…	…	…
…	…	…	…
…	…	…	…

Example: TLB-Entries

Valid	Virtual page	Modified	Protection	Page frame	
1	140	1	RW	31	data
1	20	0	R X	38	code
1	130	1	RW	29	data
1	129	1	RW	62	data
1	19	0	R X	50	code
1	21	0	R X	45	code
1	860	1	RW	14	stack
1	861	1	RW	75	stack

Besides the above necessary control bits a TLB entry can contain:

- TID or PID as a tag (so called tagged TLB)
- Reference-Bit (set by MMU)
- Dirty-Bit (set by MMU)
- Cache Disabled Bit

Implementation of a TLB

- TLB size: typically 64 – 128 entries
- Separate TLBs for instruction and data
- TLB needed for
  - Multi level PTs (especially for 64 Bit machines)
  - Inverted page tables
- 2 TLB organizations:
  - Hardware-controlled TLB
  - Software-controlled TLB

HW-Controlled TLB

- On a TLB miss MMU scans the PT ($\Rightarrow$ fast)
  - MMU loads new TLB if page is mapped
    - Need to write TLB entries back if TLB is full
    - PT layout is fixed (e.g. Pentium)
  - MMU generates a page fault if page is not mapped
    - Software performs page fault handling …
    - Restart the faulting instruction of page faulting KLT

faster, but inflexible

- On a TLB hit MMU checks control bits
  - If valid & allowed, it performs address translation
  - If V-Bit is not set or if access is not allowed, MMU generates an exception

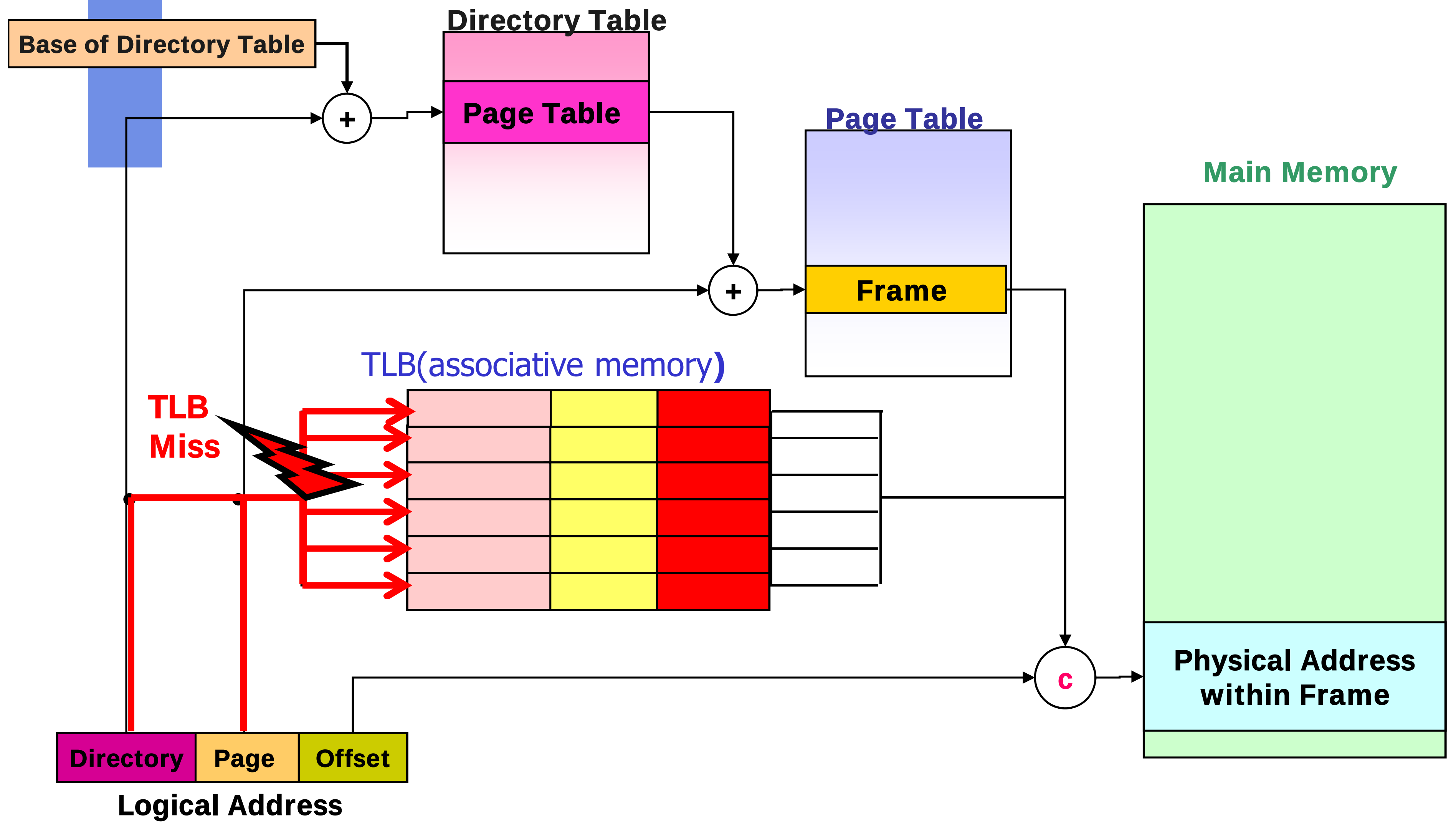
SW-Controlled TLB

- On a TLB miss, HW raises an exception
  - Exception handler scans the PT and loads corresponding TLB from PT if page is mapped
    - Need to write back if TLB is full
    - PT layout is free (e.g. MIPS)

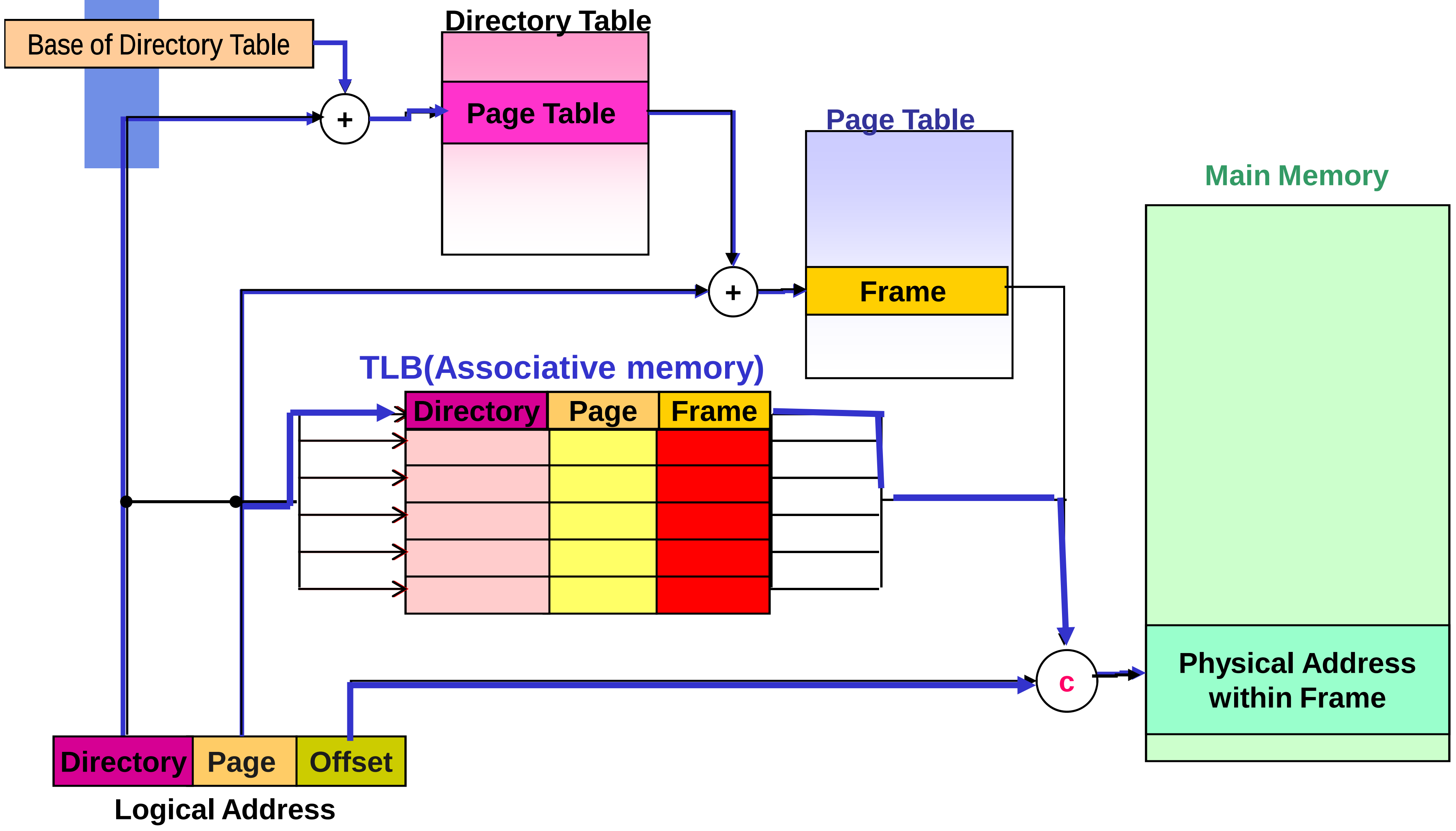
Slower, but more flexible

  - If page not mapped call page fault handler, i.e.
    - Software performs page fault handling …
    - Restart the faulting instruction of page faulting KLT
- On a TLB hit MMU checks the control bits …

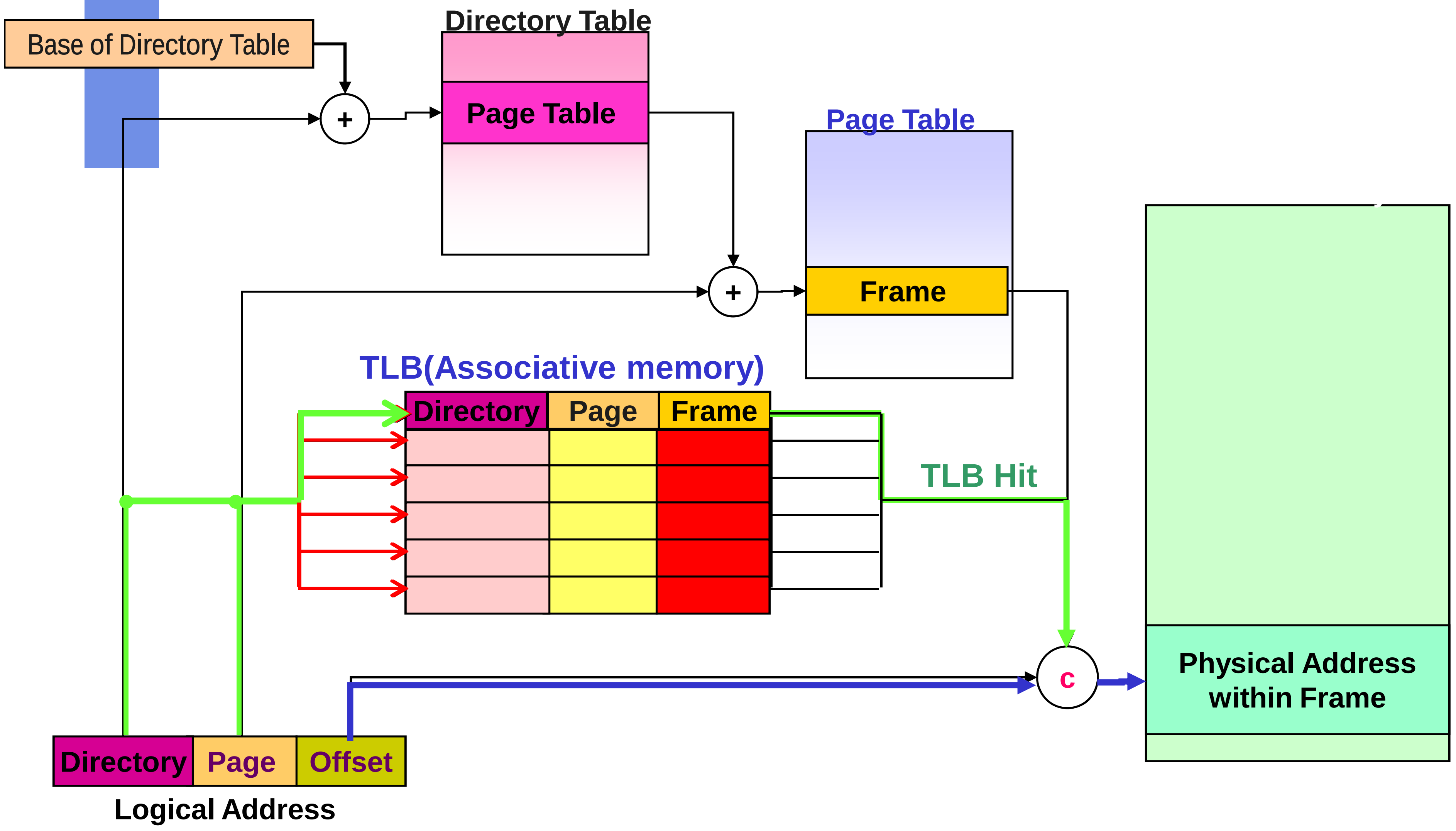
2-Level Address Translation & TLB



2-Level Address Translation & TLB



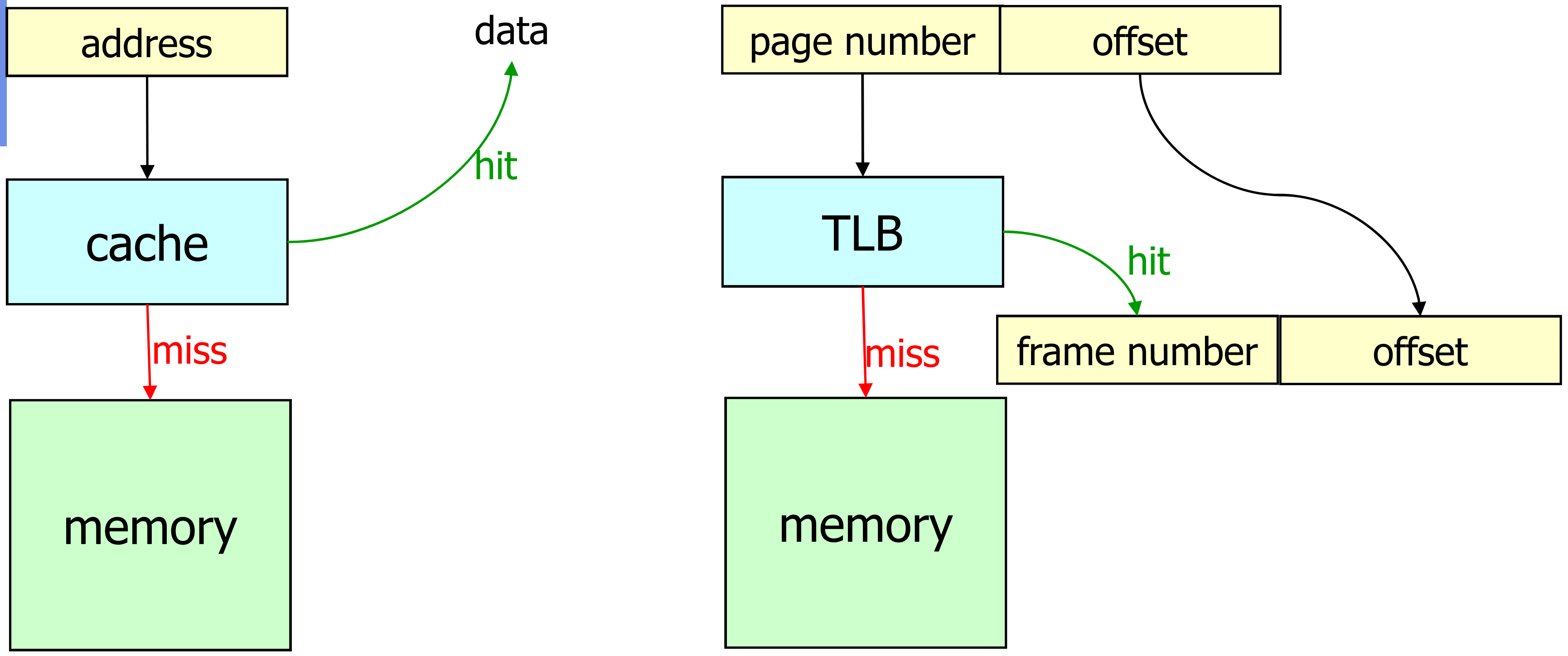
2-Level Address Translation & TLB



HW versus SW Controlled TLB

- HW Controlled TLB
  - Efficient (i.e. fast)
  - Inflexible
  - Need more space for page table
- SW Controlled TLB
  - Not as fast as HW
  - Flexible
  - Can deal with large virtual address spaces

Cache versus TLB



- Similarities
  - cache a portion of memory hierarchy
  - write back on a miss
- Differences
  - Associativity
  - Consistency

Consistency Issues

- "Snoopy" cache protocols can maintain consistency with memory, even when DMA happens
- No hardware maintains consistency between memory and TLBs, you need to flush related TLBs whenever changing a page table entry in memory
- On multiprocessors, when you modify a page table entry, you might need to do "TLB shoot-down" to flush all related TLB entries on **all processors**

TLB and Thread Switch

- TLB is a shared HW object ⇒
- Page tables are per address space, i.e. either per task or per process ⇒
  - TLB must be flushed each time a thread of another task/process has to run next
  - High context switching overhead (x86) or we use
- Tagged TLBs, i.e. ASID (or TaskID or PID) is part of TLB entry
  - Used in some modern architectures
  - TLB entry: ASID, page number, frame number, ….

TLB Effect

- Without TLB (and linear PT)
  - Average number of physical memory references per virtual reference = 2
- With TLB (assuming 99% hit ratio)
  - Average number of physical memory references per virtual reference

    = .99 * 1 + 0.01*2 = 1.01

Performance Analysis of Paging

<u>Given:</u> p_{pf} = page fault probability
at_m = access time to main memory
et_{pf} = execution time for a page fault

What is the average access time eat to virtual memory?

$$\boxed{\mathbf{eat_{vm} = (1 - p_{pf}) * at_m + p_{pf} * et_{pf}}}$$

Example: $p_{pf} = 0.001$, at_m = 20 nsec
then et_{pf} = 20 msec

$\Rightarrow eat_{vm} \sim 10^6\ at_m$

<u>Conclusion:</u> Minimize number of page faults,
i.e. replace very carefully,
i.e. use sophisticated replacement policy (see later)

Page Size Issue?

Base page size = 2^{size} may be dictated by hardware

Which size to use is an open question?

- Advantages of large pages:
  - Short page tables (decreases # of PTEs)
  - Increases TLB coverage
  - Few TLB entries $\Rightarrow$ increases TLB hit ratio
  - Increases swapping throughput (disk and RAM)
- Advantage of small pages:
  - Reduces "internal fragmentation"
  - Decreases page fault latency (but only a little bit)

Page Size Issue

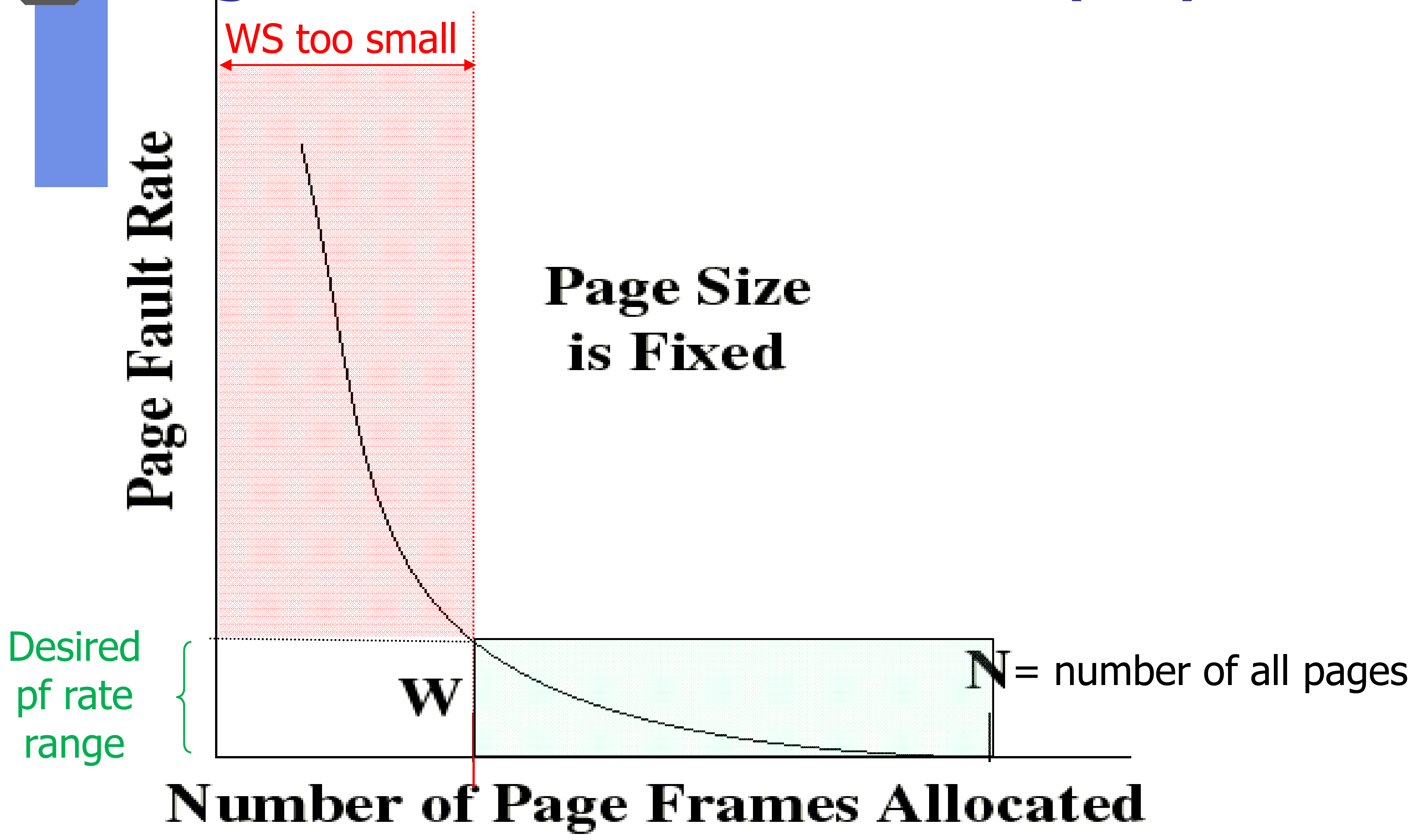
- Multiple page sizes provide flexibility to improve usage of a TLB, e.g. some suggest
  - Large pages for code (especially if shared)
  - Smaller pages for thread stacks

- OSes often support only 1 page size

Page Sizes in Practice

Computer	Page Sizes
Atlas	0.5 K 48-bit words
Honeywell-Bull/Multics	1 K 36-bit words
IBM 370/XA	4 KB
DEC VAX	0.5 KB
IBM AS/400	0.5 KB
Intel Pentium	4 MB, 4 KB
ARM	64 KB, 4 KB
MIPS R4000	16 MB – 4 KB in powers of 4
DEC Alpha	4 MB – 8KB in powers of 8
Ultra Sparc	4 MB – 8KB in powers of 8
PowerPC	4 KB
Intel IA-64	256MB – 4KB in powers of 4

Page Fault Behavior as a f(RS)


WS too small
Page Fault Rate
Page Size is Fixed
Desired pf rate range
W
N = number of all pages
Number of Page Frames Allocated