EEL-4736/5737 Principles of Computer System Design

Lecture Slides 11

Textbook Chapter 5

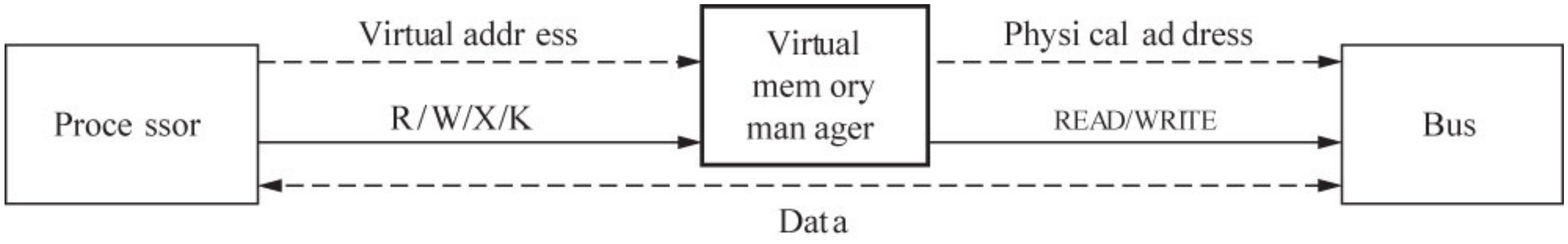
Virtual Memory

Introduction

- Memory domains restrict management
  - How to grow them if memory is limited?
    - Copying is expensive
    - Re-locating and re-computing address is cumbersome
- Virtual memory adds a layer of abstraction for better management
  - Applications use virtual addresses as names in the context of its own virtual address space
  - Bus sees physical address names
  - Re-mapping vs. copy/relocation

Virtualizing addresses

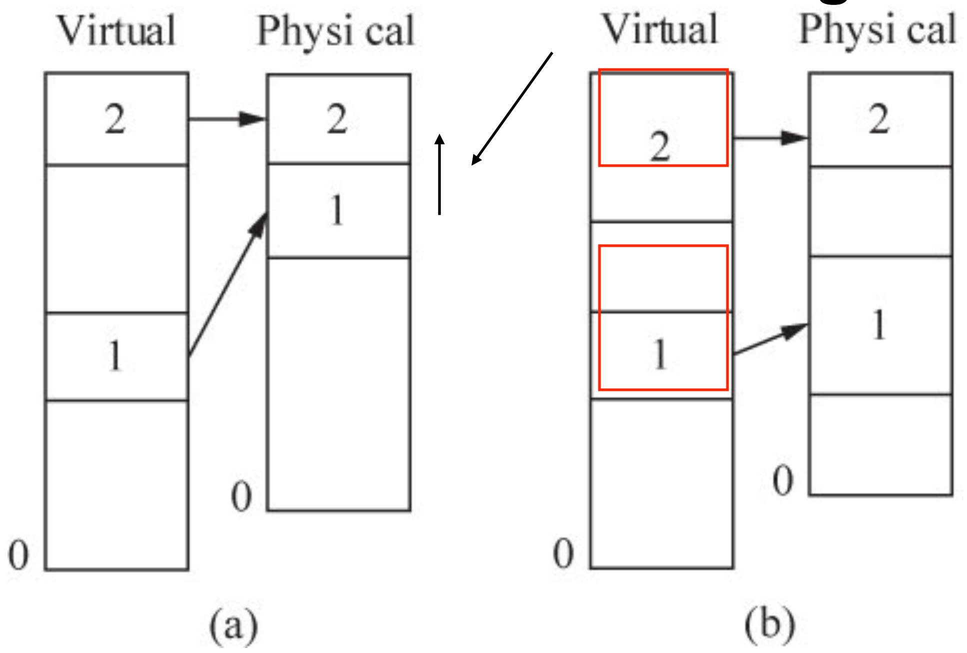
- Threads use *virtual addresses*
- Bus sees *physical addresses*
- Memory manager: *permission checks and translation*
  - – Decouple modules with indirection



Example

- Large set of virtual addresses (2^64) but a smaller set of physical addresses
- Still use domains, with virtual addresses
- What if need domain 1 to grow?

No room to grow



Copy and grow
Thread unaware of relocation

Domains with virtual address

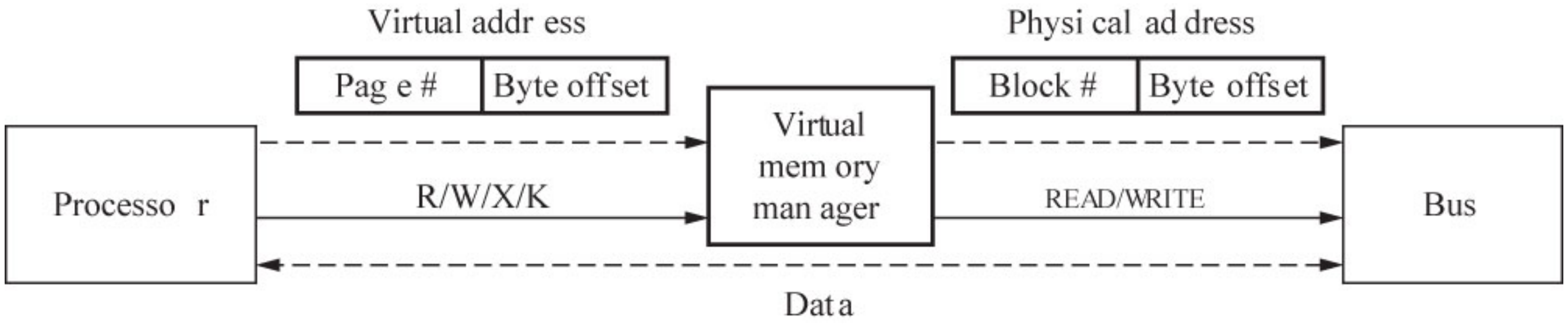
- Pros:
  - Relocation is application-transparent
- Cons:
  - Translation is in the path of every memory reference
  - Domain growth may still incur copying
  - Fragmentation
  - Need additional modules to implement memory manager functions

Using a page map

- Mapping an entire domain
  - Coarse grain: fragmentation, expensive copy
- Mapping individual addresses
  - Possible conceptually, but keeping track of mapping would require huge tables
- Design approach: page maps
  - An array of page map entries
  - Each entry maps for a *fixed-size range of consecutive bytes of virtual addresses* (page) to a range of physical addresses (block, or frame)

Page map

- Virtual addresses now is a name overloaded with a structure to translate on a per-page basis
  - Upper bits: page number
  - Lower bits: page offset

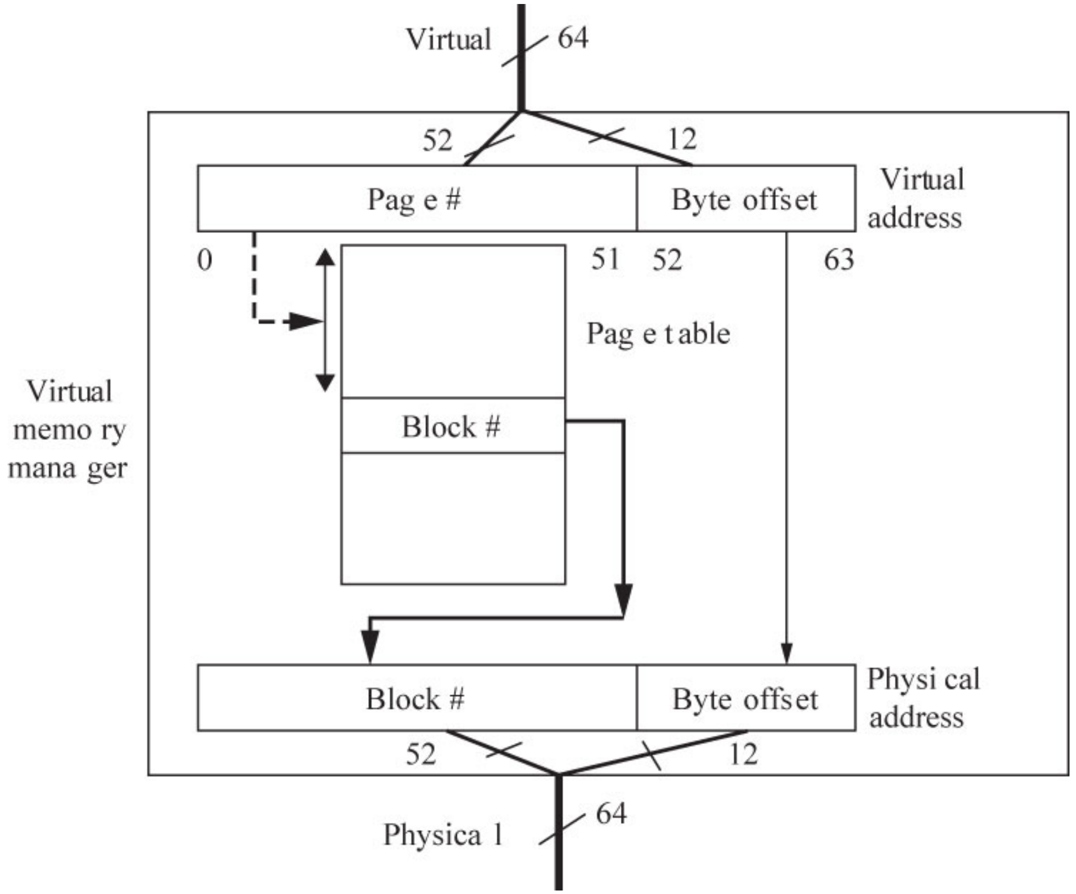


Memory management

- One page map per allocated page
- Free to allocate/move any block in physical memory to hold a page
  - Insert/update appropriate map
  - Transparent to the thread
  - Unit of management: same size blocks instead of varying-size domains

Implementing page maps

- Page tables - only one field of the virtual address needs translation



Page tables

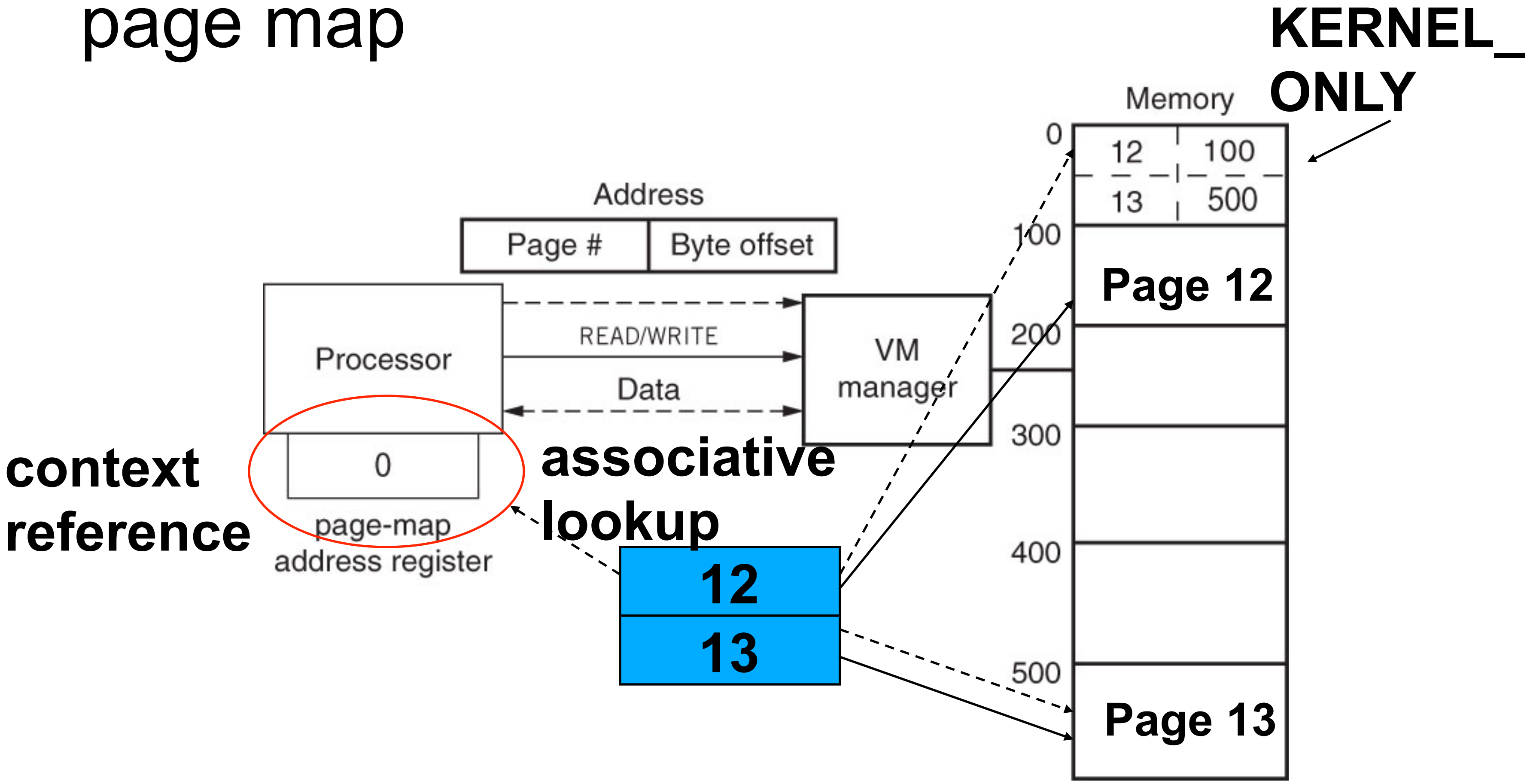
- Mapping table size still a concern
  - 4KBytes per page, 64-bit virtual addresses – 2^(52) * 52 bits at a minimum
    - 52: 64 bits – 12 offset
  - Can be addressed with techniques such as multi-level page tables
  - We'll assume for simplicity a single-level table

Implementing page tables

- Even with techniques to reduce size, page tables are larger than what's typically available within a processor
- Hence, they are themselves stored in main memory
- Implications:
  - Need to be able to locate the page table
  - Need to protect the page table from non-kernel threads
  - Performance optimizations are needed

Implementing page maps

- Example – assume processor has a register pointing to the address of the page map



Virtual address spaces

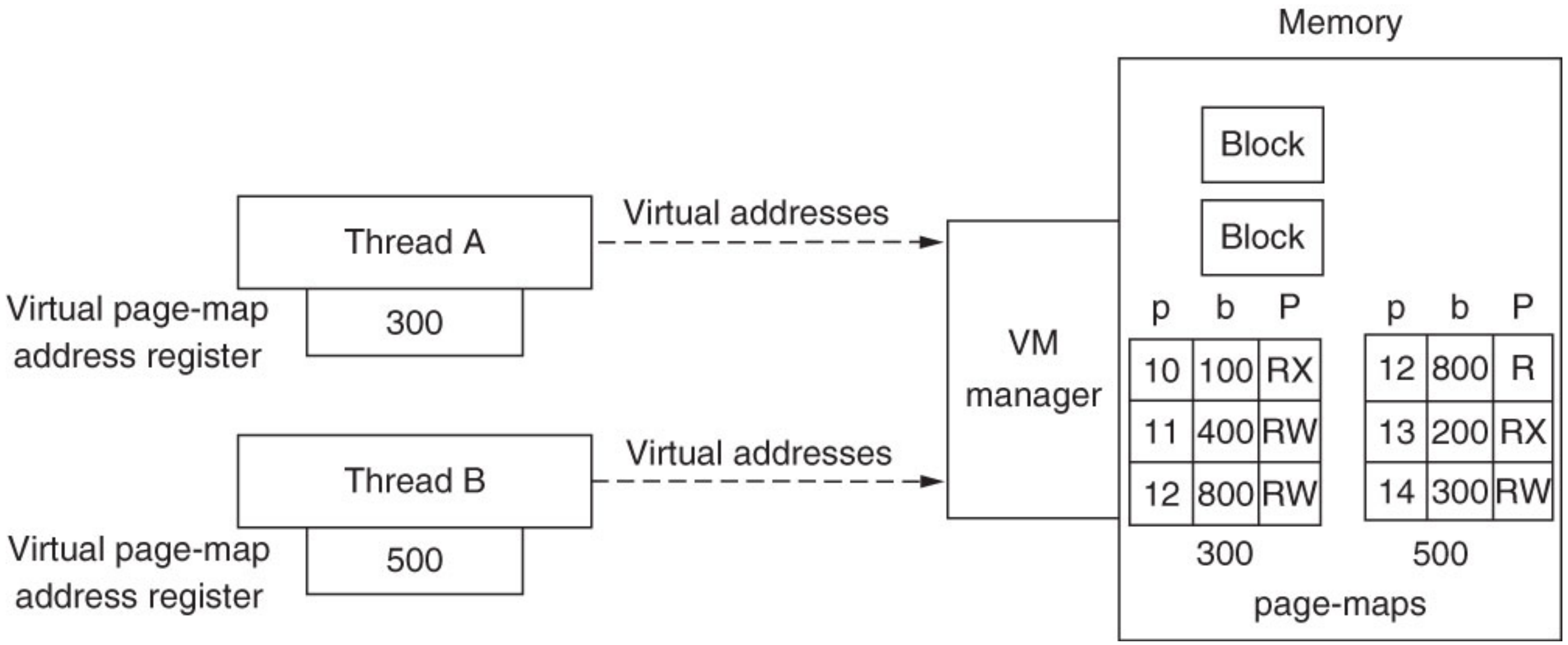
- A single, large set of virtual addresses could be shared by all threads
  - Practicality depends on address bits
    - E.g. 64-bit virtual addresses; might have 4 billion threads each using 4GB; 32-bit – 8 512MB threads?
    - Another problem: relocation needed depending on base address allocated
    - E.g. multiple instances of the same program
- Can eliminate this assumption with per-application virtual address spaces

Virtual address spaces

- An application has the illusion that it has a complete address space to itself
  - 32-bit virtual address space: *each* thread can use 2^32 virtual addresses
- Key feature to support this:
  - *Each thread has its own page map*
- Often, operating system *process* is used to refer to 1+ threads that share a virtual address space

Virtual address spaces

- Virtual memory manager now keeps a map per virtual address space



Address translation

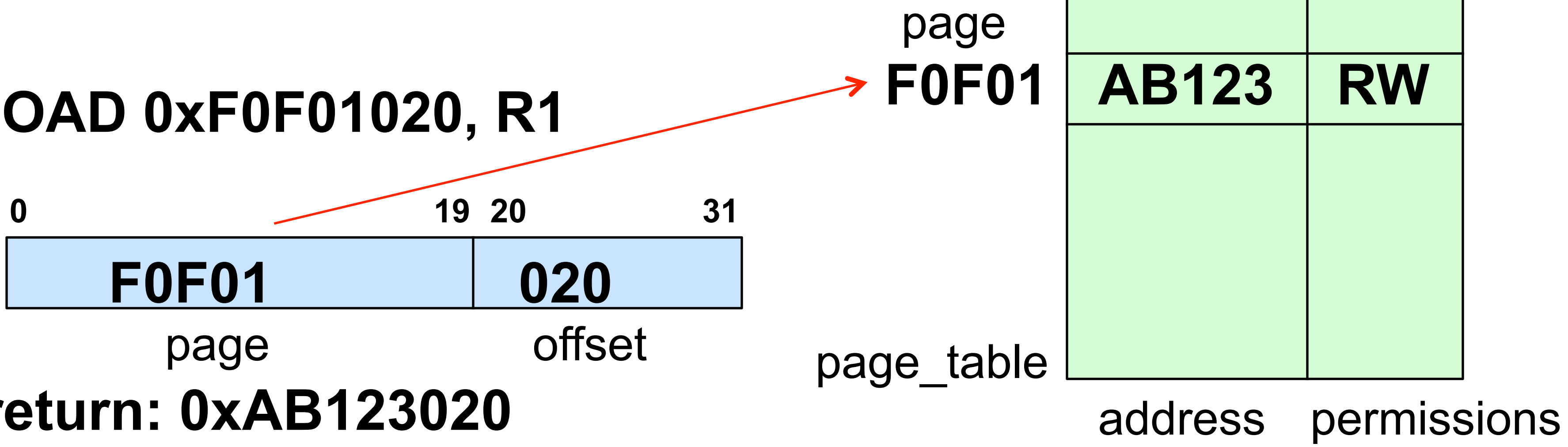
```
procedure TRANSLATE (integer virtual, perm_needed)
returns physical
   page <- virtual[0..19]     // 32-bit address space
   offset <- virtual[20-31]   // 4KB (12-bit offset) pages
   page_table <- PMAR
   perm_page <- page_table[page].permissions
   if PERMITTED (perm_needed,perm_page)
     block <- page_table[page].address
     physical <- block + offset
     return physical
   else return error
```

Address translation

```
procedure TRANSLATE (integer virtual, perm_needed) returns physical
   page <- virtual[0..19]      // 32-bit address space
   offset <- virtual[20-31]   // 4KB (12-bit offset) pages
   page_table <- PMAR
   perm_page <- page_table[page].permissions
   if PERMITTED (perm_needed,perm_page)
     block <- page_table[page].address
     physical <- block + offset
     return physical
   else return error
```

perm_needed
LOAD: R
STORE: W
Exec: X
Kernel: K

LOAD 0xF0F01020, R1

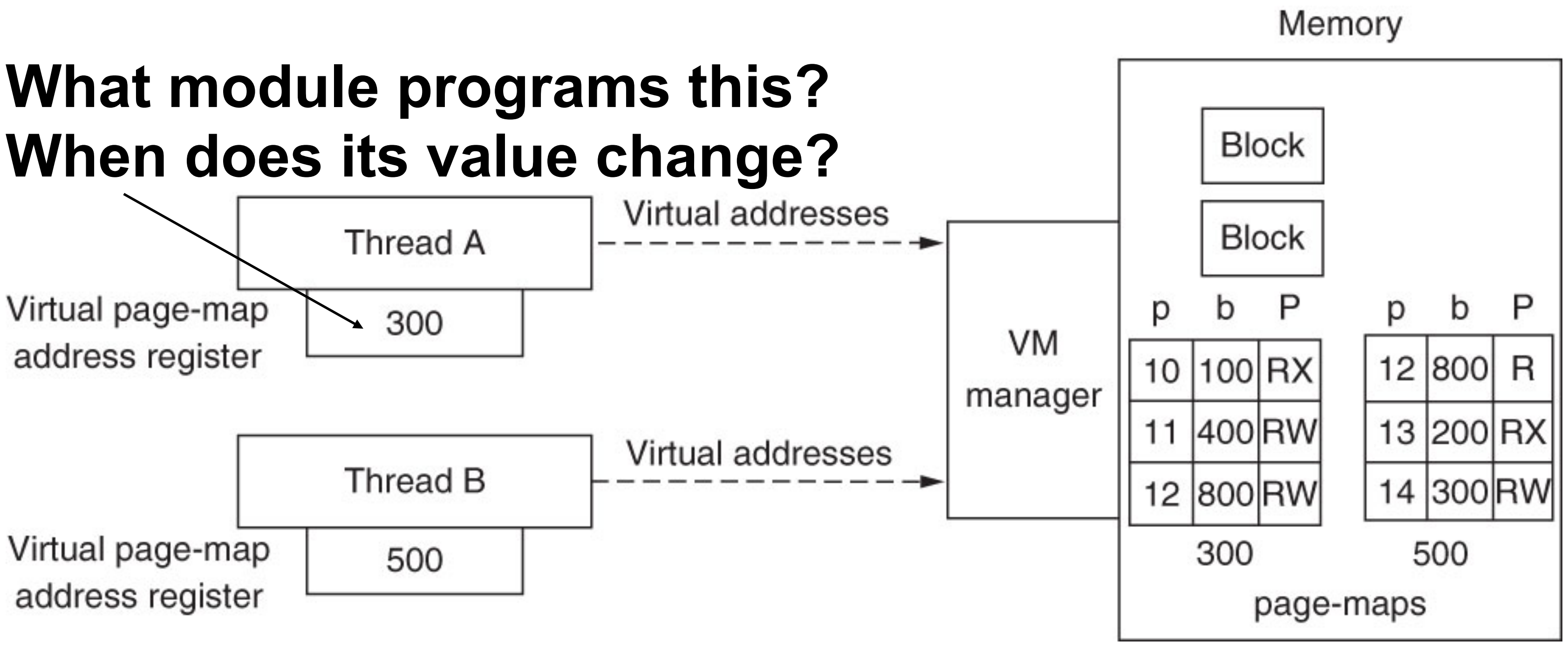


return: 0xAB123020

Example

- What happens if:
  - A issues a read reference to VA within page 10?
  - B issues a read reference to VA within page 10?
  - A issues a write reference to VA within page 12?
  - B issues a write reference to VA within page 12?

What module programs this?
When does its value change?



Primitives

- id <- CREATE_ADDRESS_SPACE()
- block <- ALLOCATE_BLOCK()
- MAP (id, block, page_number, permission)
- UNMAP (id, page_number)
- FREE_BLOCK (block)
- DELETE_ADDRESS_SPACE(id)

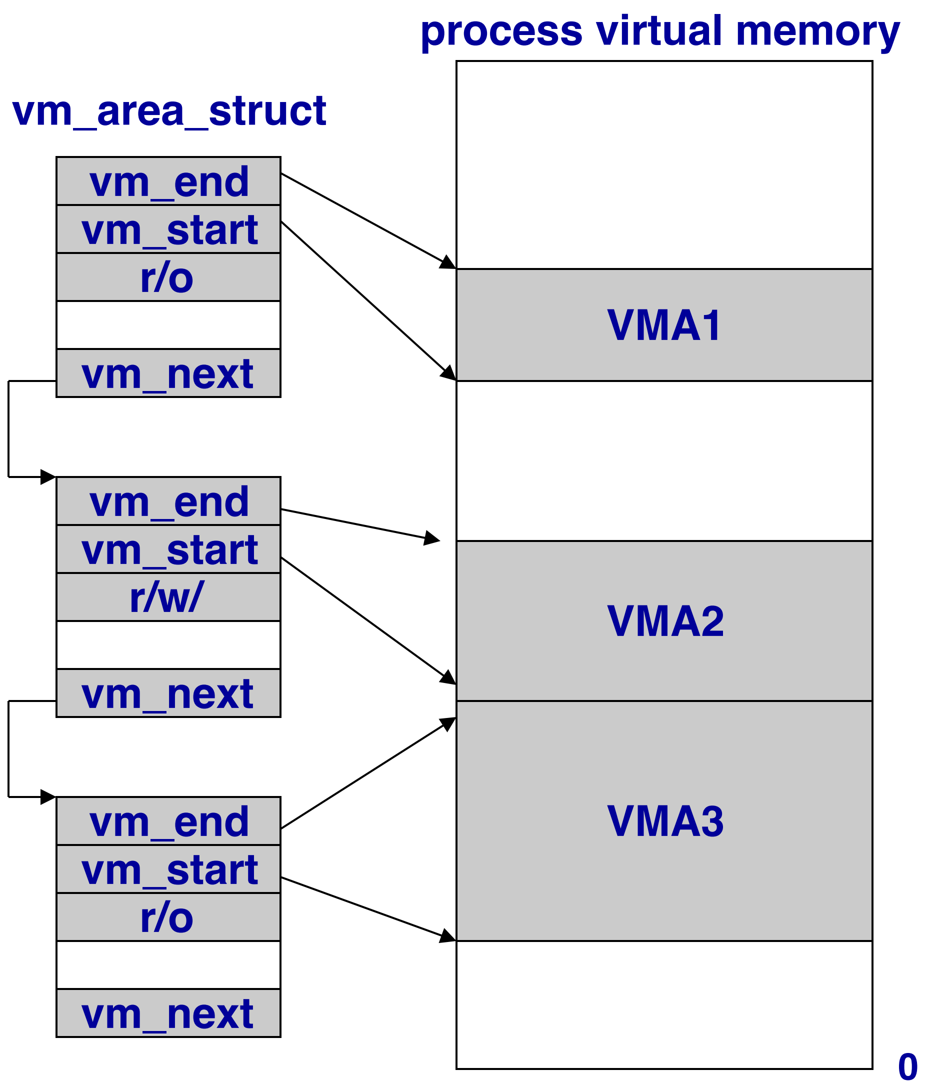
SW/HW interface

- Virtual memory manager in the kernel
  - Allocate/map blocks
  - Maintain data structures for each virtual address space in memory
  - Program hardware structures to put appropriate address space “in context”
- Memory management unit in hardware
  - Translation/permission checks for individual memory accesses, at high speed

Example: Linux VM “areas”

- Linked vm_area_struct structures
- A VM area: a part of the process virtual memory space that has a special rule for the page-fault handlers (i.e. a shared library, the executable area etc).
- These are specified in a vm_area_struct
  - Start and end VM address of area
  - Protection information, flags

Linux fault handling

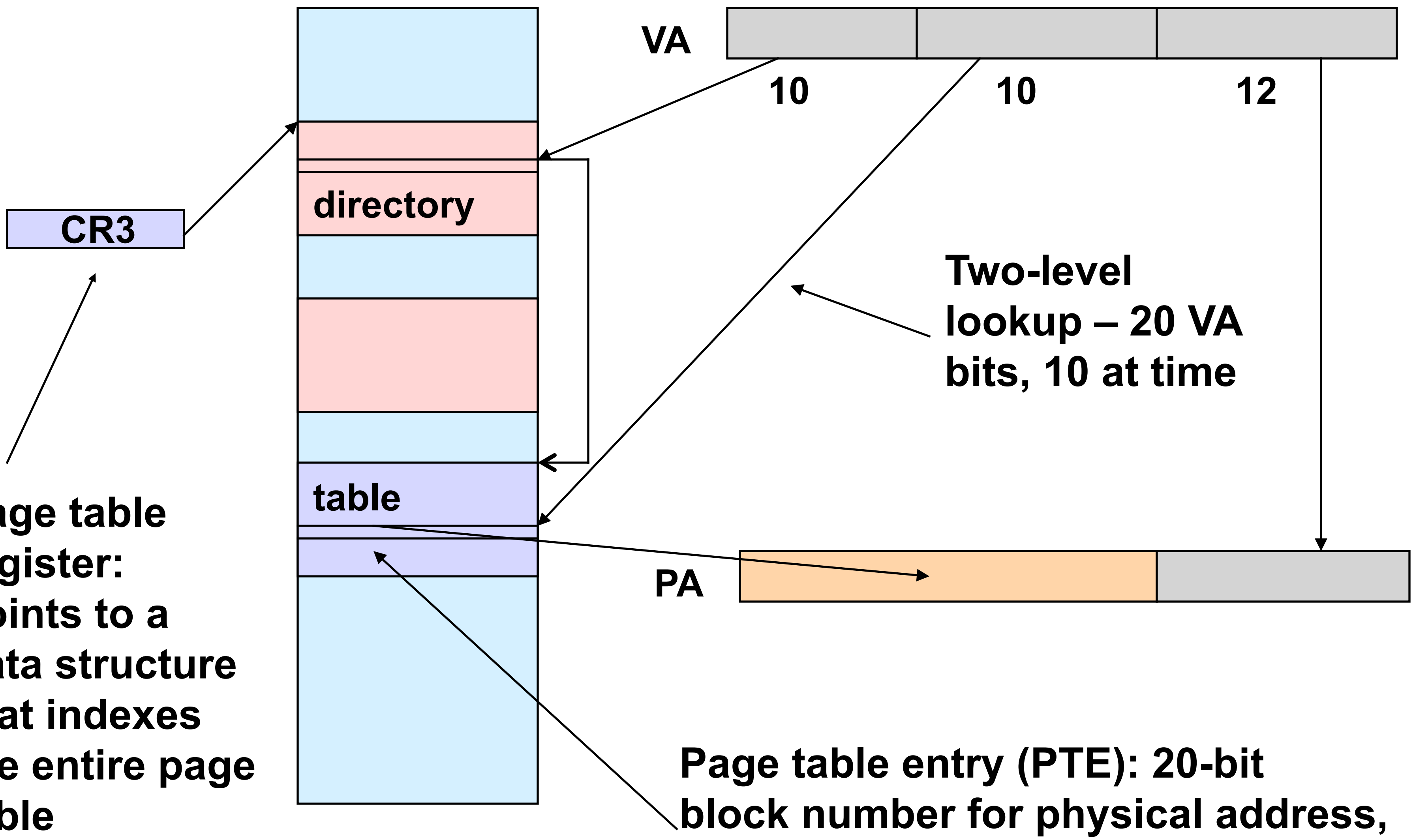


- Exception triggers O/S handling if address out of bounds or protection violated
- Traverse vm_area list, check for bounds
  - If not mapped, it is a segmentation violation – signal to process
- If mapped, check protection of vm_area_struct
  - E.g. r/o, r/w
  - Signal protection violation to process if access not allowed
  - Otherwise, handle fault and bring page to memory

Kernel and address spaces

- Can separate the kernel and user address spaces in different ways
  - They share the same address space, but kernel pages are KERNEL_ONLY
    - E.g. top bit of virtual address divides what's available to kernel vs. user
  - Or, kernel has different page map altogether
    - SVC instruction and return to user mode must also change the page map register

E.g.: x86 2-level page tables



Two-level lookup – 20 VA bits, 10 at time

Page table register: points to a data structure that indexes the entire page table

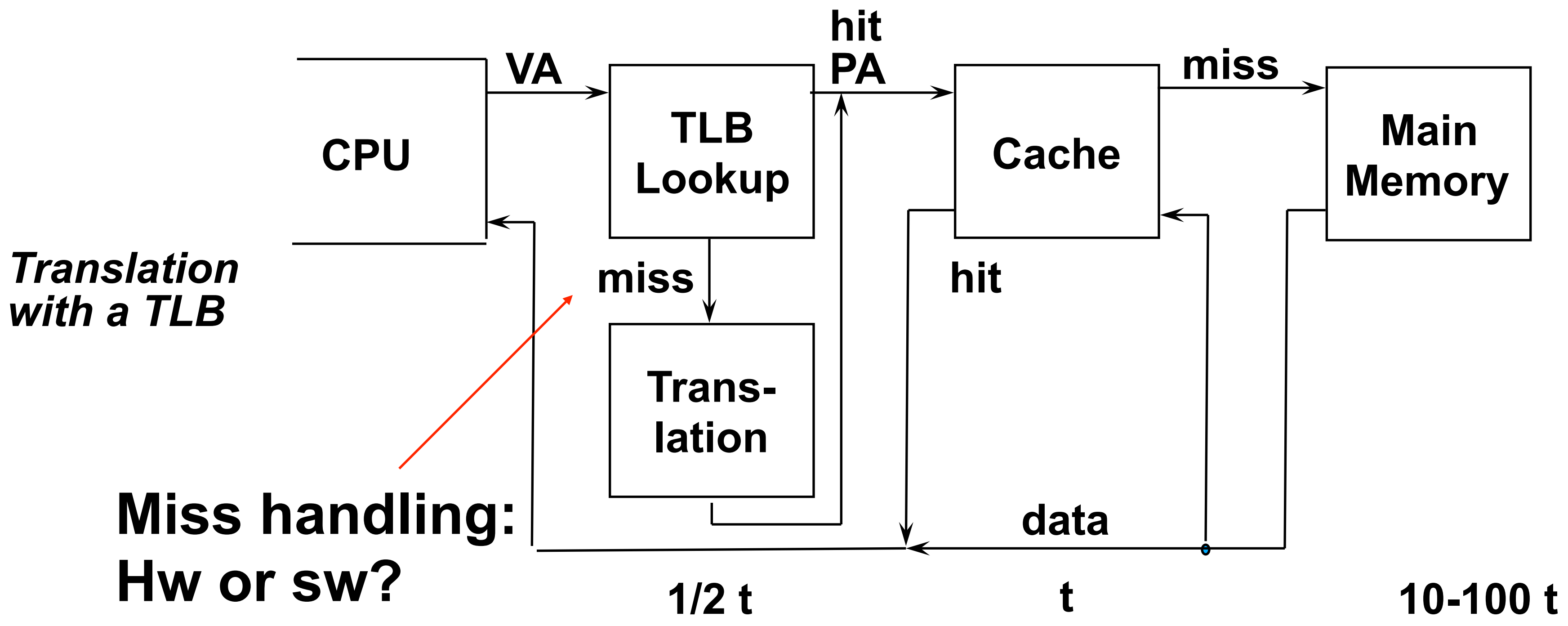
Page table entry (PTE): 20-bit block number for physical address, plus permission bits (fits in 4 Bytes)

HW/SW Tradeoffs

- Per-access translation needs to be done in hardware for high speed
  - Two memory lookups per access – too expensive
  - Translation look-aside buffer (TLB) – a specialized cache for page table entries

TLBs

- Caches often indexed by physical address
- TLBs cache translations so they can be computed before going into the cache
  - Typically, set-associative



HW/SW Tradeoffs

- How are TLB misses handled? Two approaches:
  - Software-managed TLB: raise exception and let software manager (kernel) handle it
    - MIPS, and other RISC-style processors
    - Pro: flexibility in data structures and algorithms to replace TLB entries
  - Hardware-managed TLB: hardware manager looks up the page table and handles miss without software involvement
    - x86 is a prime example
    - Pro: fast TLB miss handling

Mapping pages to storage

- Additional flexibility given by virtual addresses – *page mappings do not need to be always to main memory*
  - A page may map to a block of a storage subsystem (e.g. hard disk)
- Benefit:
  - Allows the available virtual memory to exceed the main memory capacity
- Challenge:
  - Must be careful with performance – disk is orders of magnitude slower

Mapping pages to storage

- Overall approach:
  – References to virtual addresses of pages that are mapped to memory translated by processor hardware (TLB, page table)
  – Pages mapped to storage blocks are flagged as not mapped, such that an exception is raised (*page fault*)
  – System software (kernel) responsible for bringing page from/to disk into memory, and letting the processor retry

Reading

- Sections 5.5-5.6