Concurrent Systems

3D4 ⟷ CS2016

Operating Systems

Andrew Butterfield
ORI.G39, Andrew.Butterfield@scss.tcd.ie

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

with thanks to Mike Brady

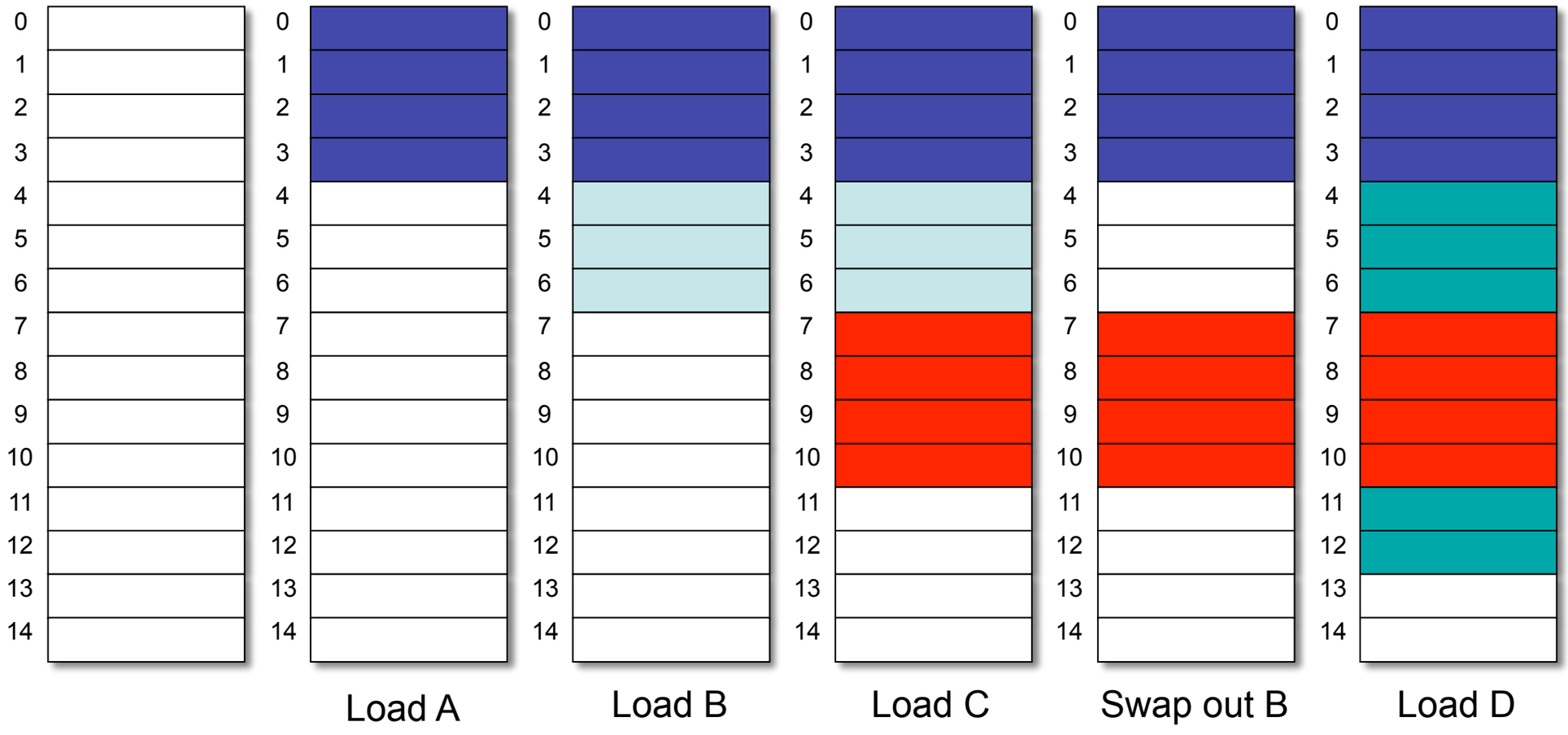
Going “virtual”

- An operating system is running a (large) number of processes
  - running their code, interrupting them, running some other process’ code…
- Nowadays we expect to be able to run more processes than will fit in physical memory
  - memory of currently running process needs to be in RAM
  - what about a waiting/ready process?

Process memory management

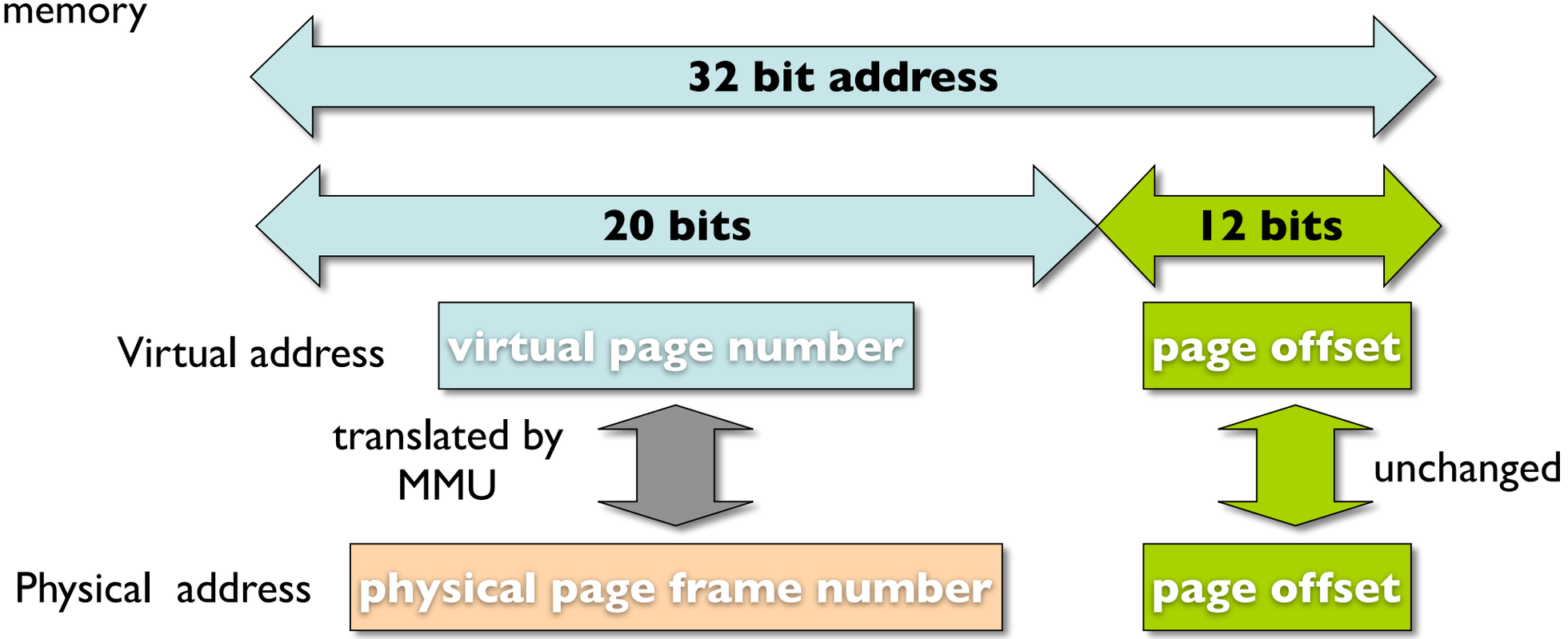
- Solution: use another memory device (e.g. a hard disk) to increase memory capacity
  - A "swap device" or "page file"
  - Treat the memory used by a process as a set of fixed size memory chunks, called pages
  - Simplifies moving memory contents to/from the swap device
  - Leads to fragmentation: process's memory is no longer contiguous in physical memory
  - Not only does the programmer not know where his/her program will be located, but now it can also move over time

Relocation and non-contiguity



Virtual memory

- Better solution: Give each process its own ***virtual address space***
  - Divide the space into fixed-size pages (e.g. 4KB or 212B)
- Programs refer to addresses in their own virtual memory space – virtual addresses
- "On-the-fly" conversion of virtual addresses into physical addresses, which are applied to physical memory



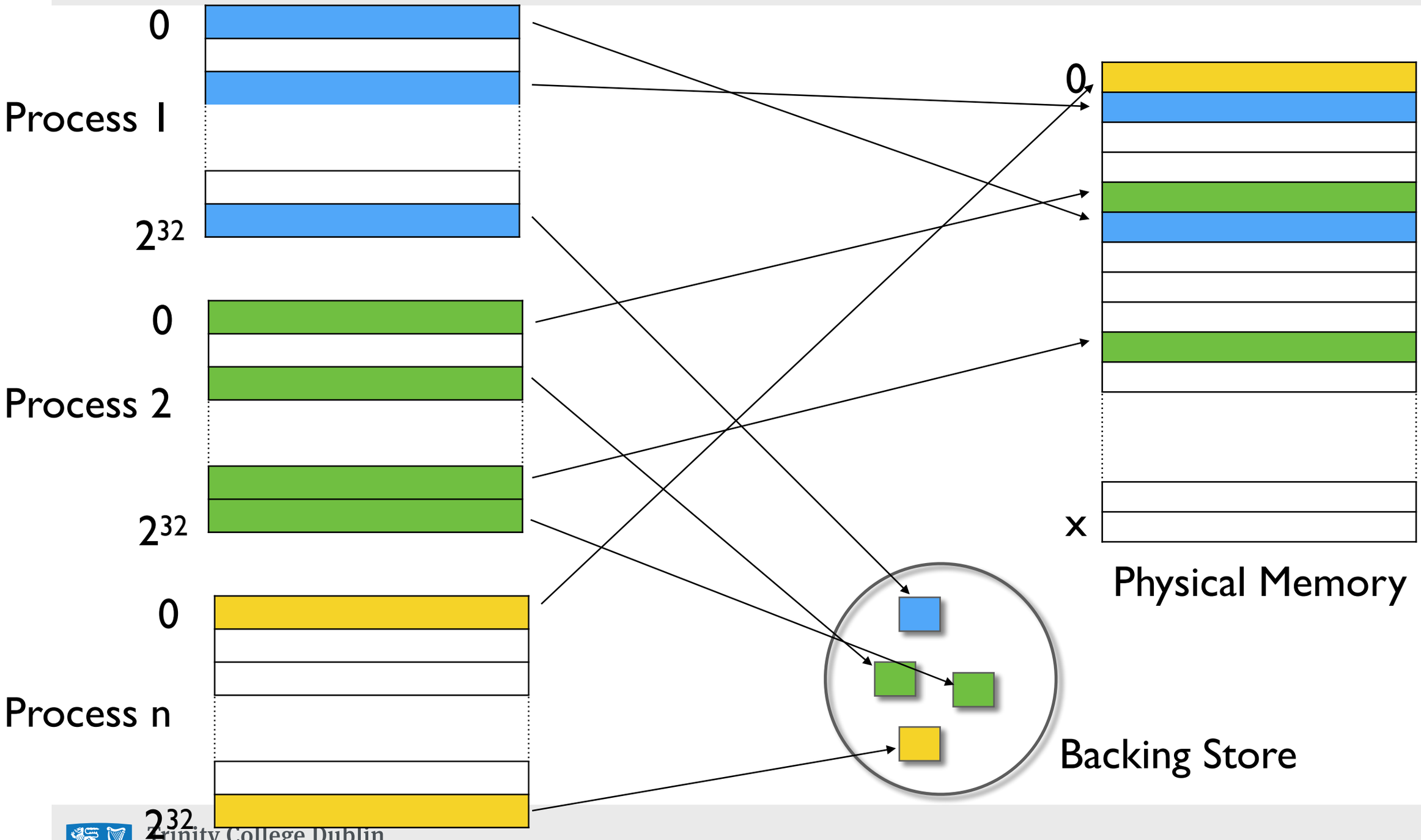
Virtual memory

- A Memory Management Unit (MMU) is usually integrated into modern CPUs
- If we use 32-bit addresses (physical and virtual) and if the address space is divided into 4KB pages
  - The total addressable memory is 2^{32} = 4GB
  - Each page contains 2^{12} bytes (4KB) requiring a 12-bit offset
  - There are 2^{20} (1,048,576) 4KB pages, each with a 20-bit page number

Virtual memory

- Per-process virtual address space
  - Each process has its own virtual address space
- The size of the virtual address space is typically much larger than the physical address space
  - Every virtual page won't be mapped to a physical page
  - Some virtual pages may not be used
  - Some pages may be mapped to a swap device or page file

Virtual memory



Virtual Memory

- Helps solve issues with
  - Location
  - Protection
  - Sharing
  - Logical Organisation
  - Physical Organisation

Location

- In a multiprogramming environment, processes (and the operating system) need to share physical memory
- Programs must be paged in and out of memory
- Programmers have no way of knowing where in physical memory their processes will be located, and processes won't be continuous
- *Virtual memory gives each process it's own contiguous virtual memory space, within which the program works.*

Protection

- In a multiprogramming environment, a user process must be prevented from interfering with memory belonging to other processes and the operating system itself
- This protection must be implemented at run-time
- *Each process only has access to its own virtual memory space, which should never map to the physical memory of another process (except under specific circumstances!)*

Sharing

- Protection must be flexible enough to allow portions of memory to be shared between processes
- *We can map the virtual pages of different processes to the same physical memory page frame (or same frame on disk)*

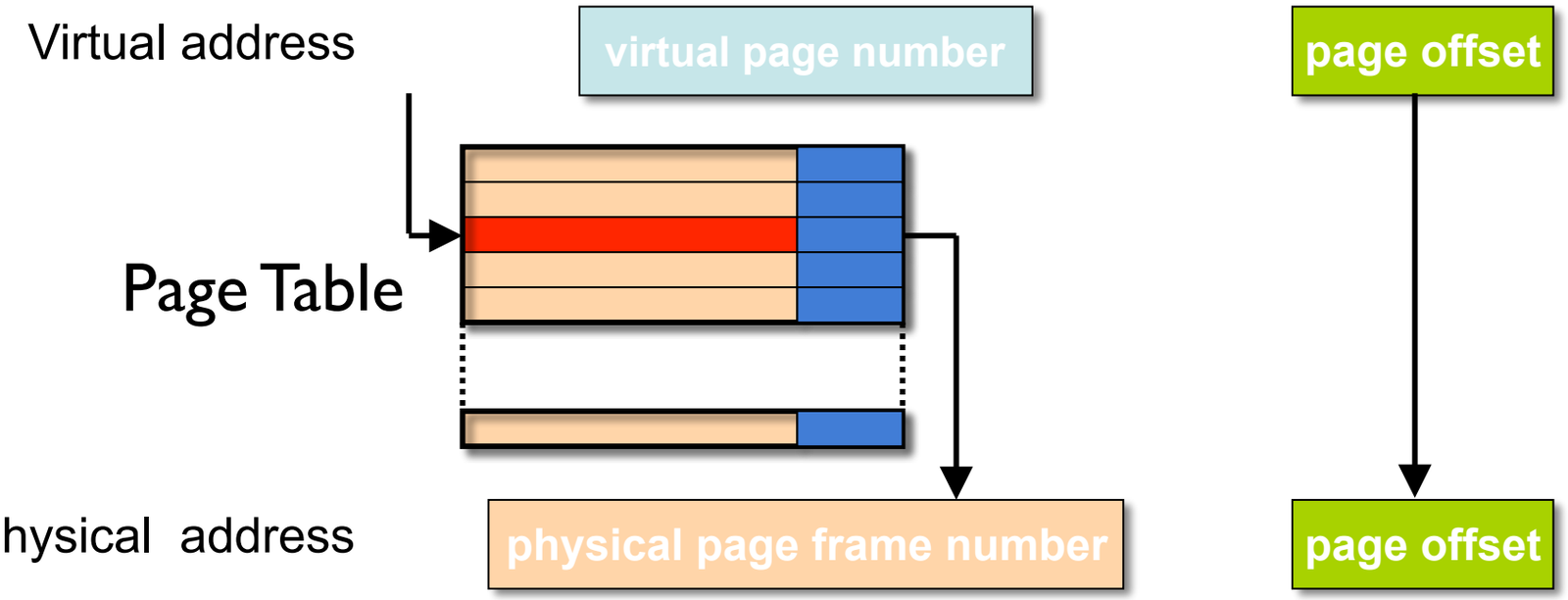
Logical Organisation

- Programs are organised as modules with different properties
  - Executable code, data, shared data
- *Assign properties to virtual memory pages, e.g. read-only, executable, non-executable, etc.*

Physical Organisation

- The program should be isolated from implementation of the memory hierarchy (e.g. swapping)
- The program shouldn't be affected with the amount of physical memory available
- *The virtual memory hides such details from the program*
- *Program can enquire about properties of interest, can ask for pages to be "wired" or locked into physical memory for performance or response reasons*

Virtual memory implementation



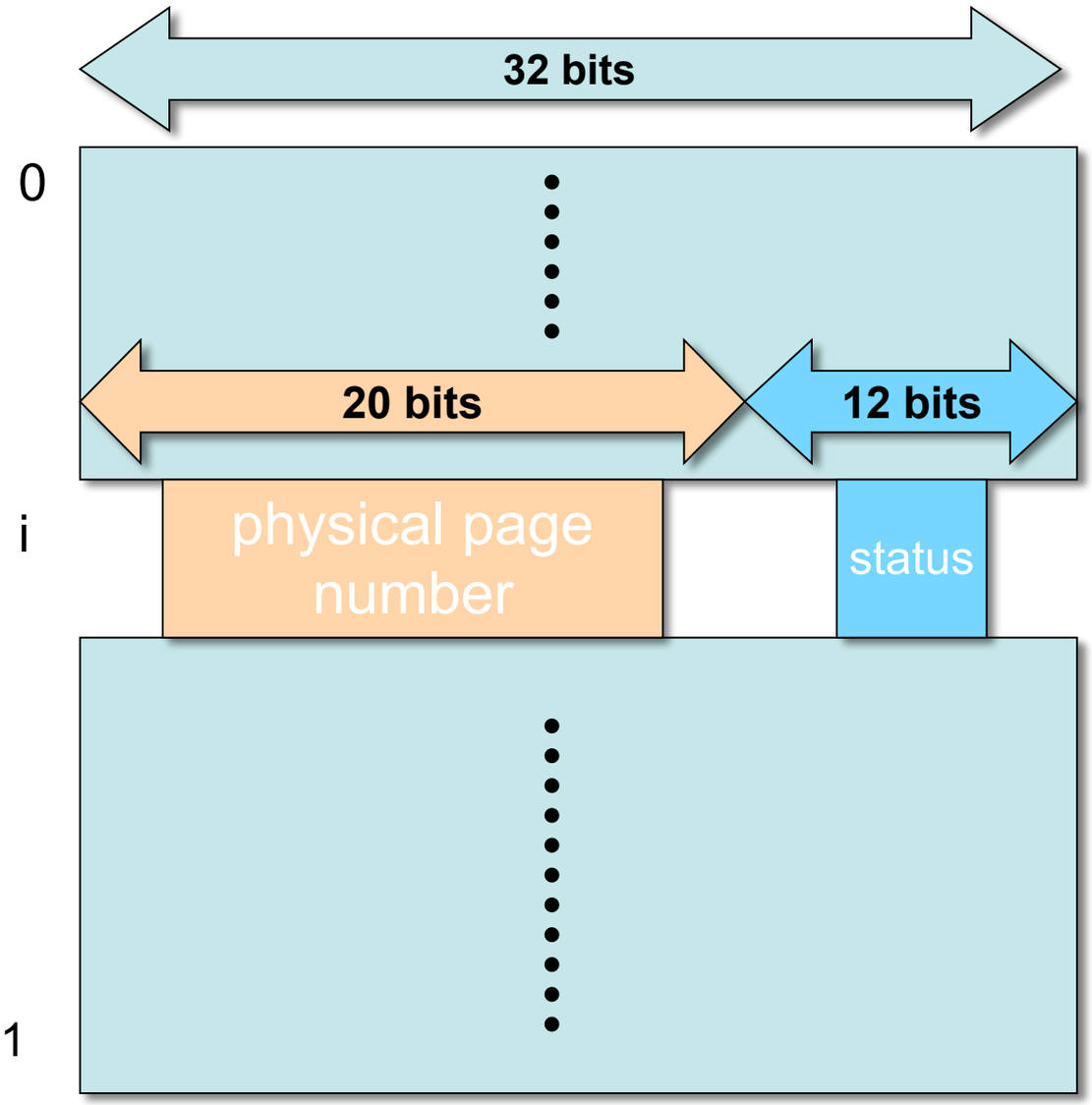
- Each process has its own page table
  - Used to map virtual pages to physical page frames
  - Page tables are also stored in memory
- To translate a virtual page number to a physical page frame number
  - Use the virtual page number as an index to the page table
  - Read the physical page frame number from the table

Virtual memory implementation

- Each process has its own page table
  - Used to map virtual pages to physical page frames
  - Page tables are also stored in memory
- To translate a virtual page number to a physical page frame number
  - Use the virtual page number as an index to the page table
  - Read the physical page frame number from the table

E.g. Virtual memory implementation

- Page table format (32-bit addresses and 4KB pages)



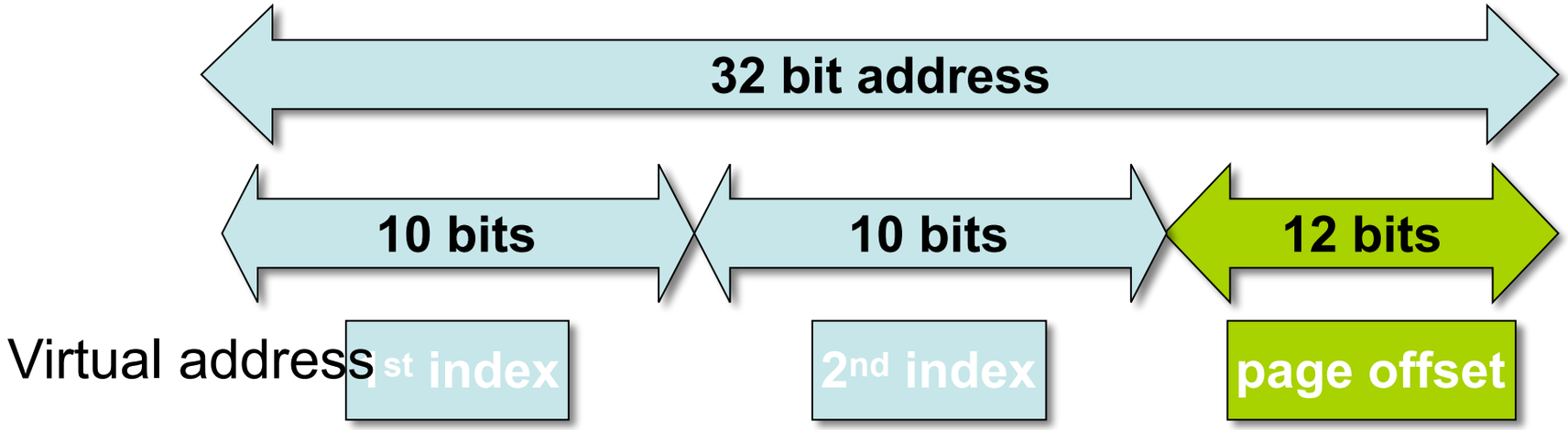
page table entry (***PTE***)

Status: e.g. valid, protection level, modified, referenced

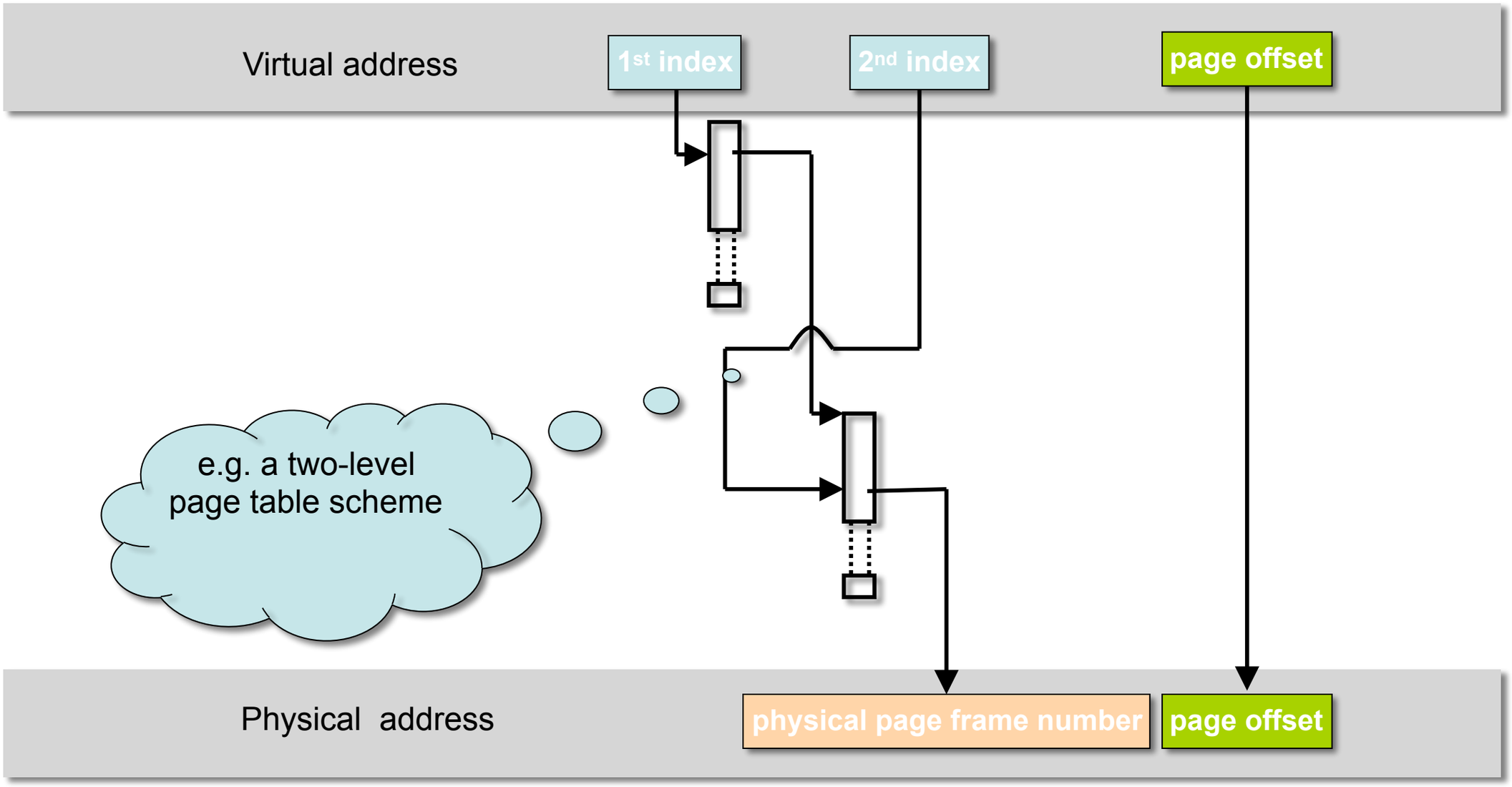
Multi-level page tables

- Assuming 32-bit addresses and 4KB page size
  - 1,048,576 page table entries x 20+ bits bytes each ≈ 3 MB
  - i.e. every process requires a 3 MB page table!!
  - wasteful, when you consider that most processes will use only a small portion of the virtual address space
  - most of the page table will be unused
- Solution
  - A multi-level page table scheme:

Multi-level page tables



Multi-level page tables



Two-level page tables

- One primary page table
- As many secondary page tables as needed
- Secondary page tables are created (and removed) as needed
  - Depending on how much of the virtual memory space is required by the process

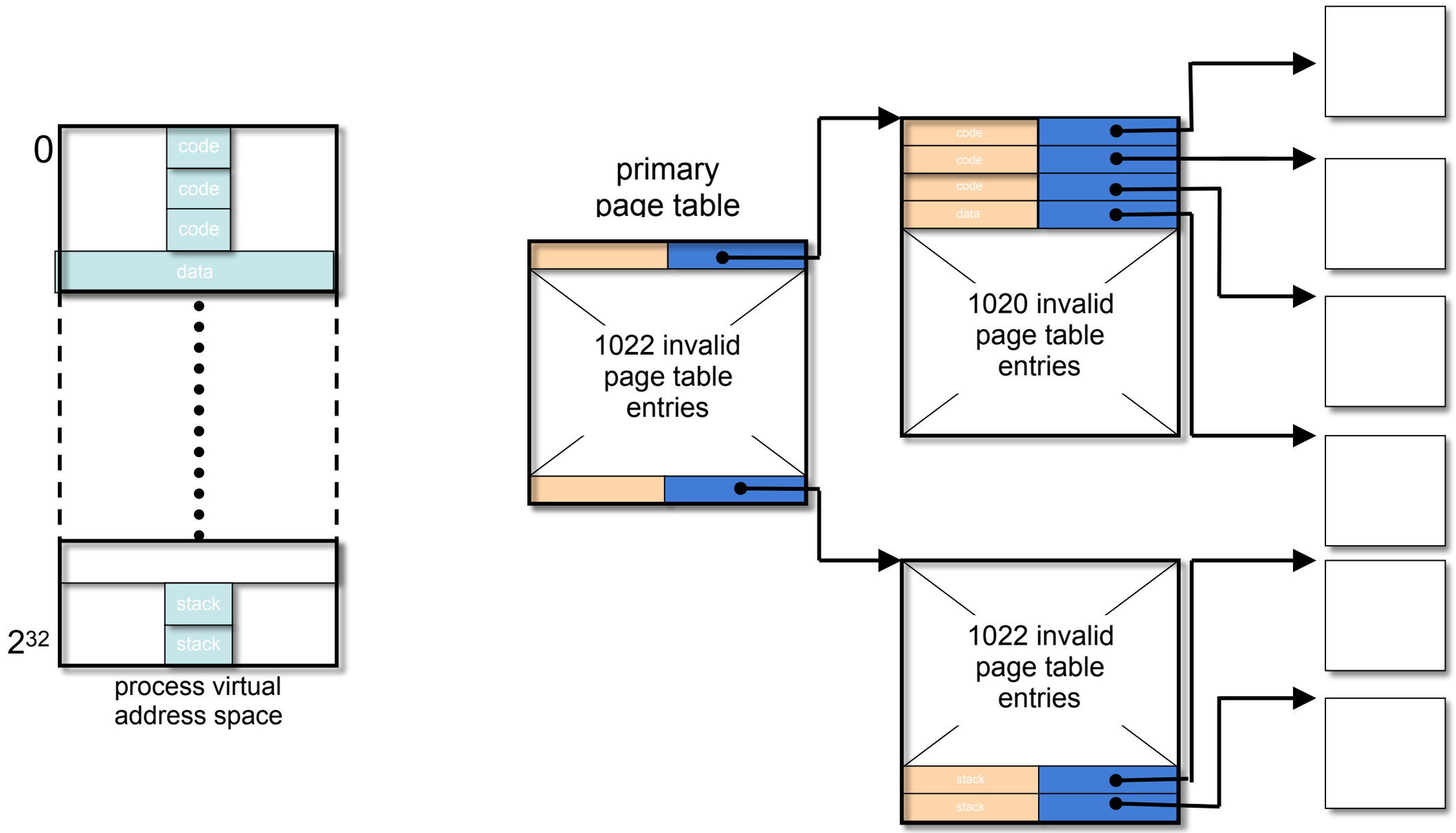
Two-level page tables

- What is the total size of all page tables in the preceding example, if two primary PTEs are used?

Two-level page tables

- In the preceding example, each page table was 4 kB in size so each page table fitted in exactly one page frame
- Different schemes will use different size bit-fields for 1st and 2nd indices
- Different schemes will use three (or more?) levels of page tables
- More efficient than a single-level scheme
- The page offset still remains un-translated.

Typical process memory layout



Page Faults

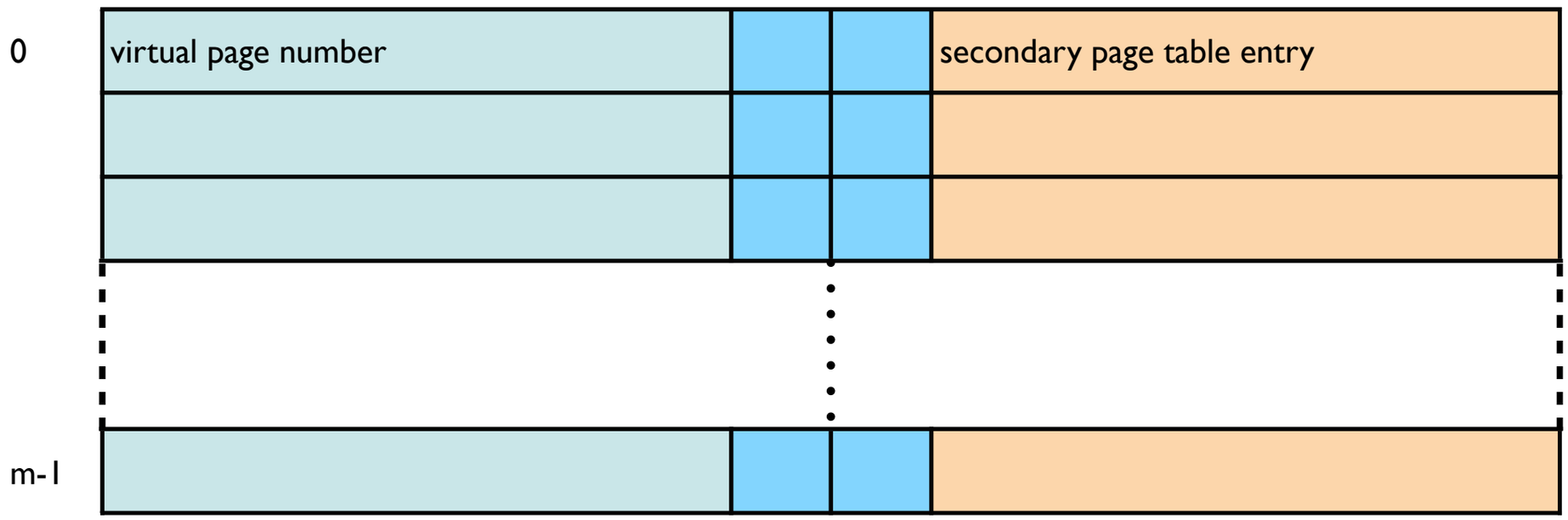
- When the MMU accesses a PTE, it checks the Valid bit of the status field to determine if the entry is valid:
  - if V==0 and a primary PTE is being accessed
    then a secondary page table has not been allocated physical memory
  - if V==0 and a secondary PTE is being accessed
    then no physical memory has been allocated to the referenced page
  - in either case, a page fault occurs

Page Faults

- The OS is responsible for handling page faults by …
  - … allocating a page of physical memory for a secondary page frame
  - … allocating a page of physical memory for the referenced page
  - … updating page table entries
  - … writing a replaced page to disk if it is dirty
  - … reading code or data from disk (another thread will be scheduled pending the I/O operation)
  - … signalling an access violation
  - ... retrying to execute the faulting instruction

Translation look aside buffer

- Each virtual-to-physical memory address translation requires one memory access for each level of page tables
  - 2 memory accesses in a 2-level scheme
- To reduce the number of memory access required for virtual-to-physical address translation, the MMU uses a Translation Look-Aside Buffer (TLB)
  - An on-chip cache
  - Stores the results of the *m* most recent address translations



Translation Look-Aside Buffer

- Before walking the page tables to translate a virtual address to a physical address…
  - … the MMU checks each entry in the TLB
- If the virtual page number is found, the cached secondary PTE is used by the MMU to translate the address …
  - … and there is no need to walk the page tables in memory
- If a TLB match is not found …
  - … The MMU walks the page tables and locates the required PTE
- The least recently used (LRU) TLB entry is now replaced with the current virtual page number and the PTE from the secondary page table
- What are the TLB implications of a context switch?

Page fault handling