Virtual Memory and MMUs

CSU34021 - Computer Architecture II

History / Background

- Main historical motivation: What to do when the program was too big to be loaded in main memory?

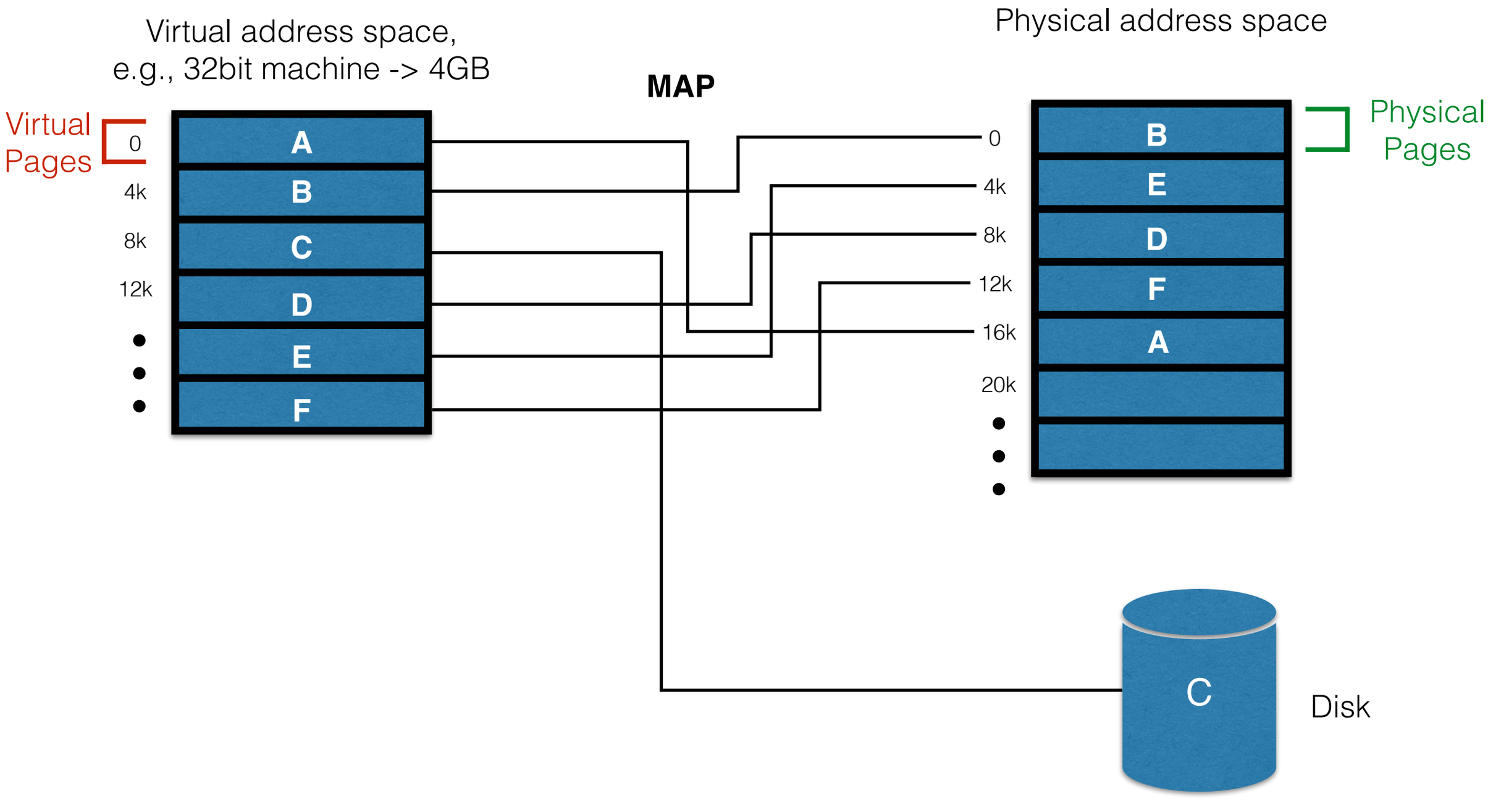
- It used to be the programmer task to make it fit!

- **Overlays:** The programmer would divide the code into different overlays that could fit into main memory, store into disk the rest, and make sure that the proper overlay was loaded into main memory every time it was needed.

- **Imagine what a burden!**

- **Virtual memory** was initially invented in order to automatise this process.

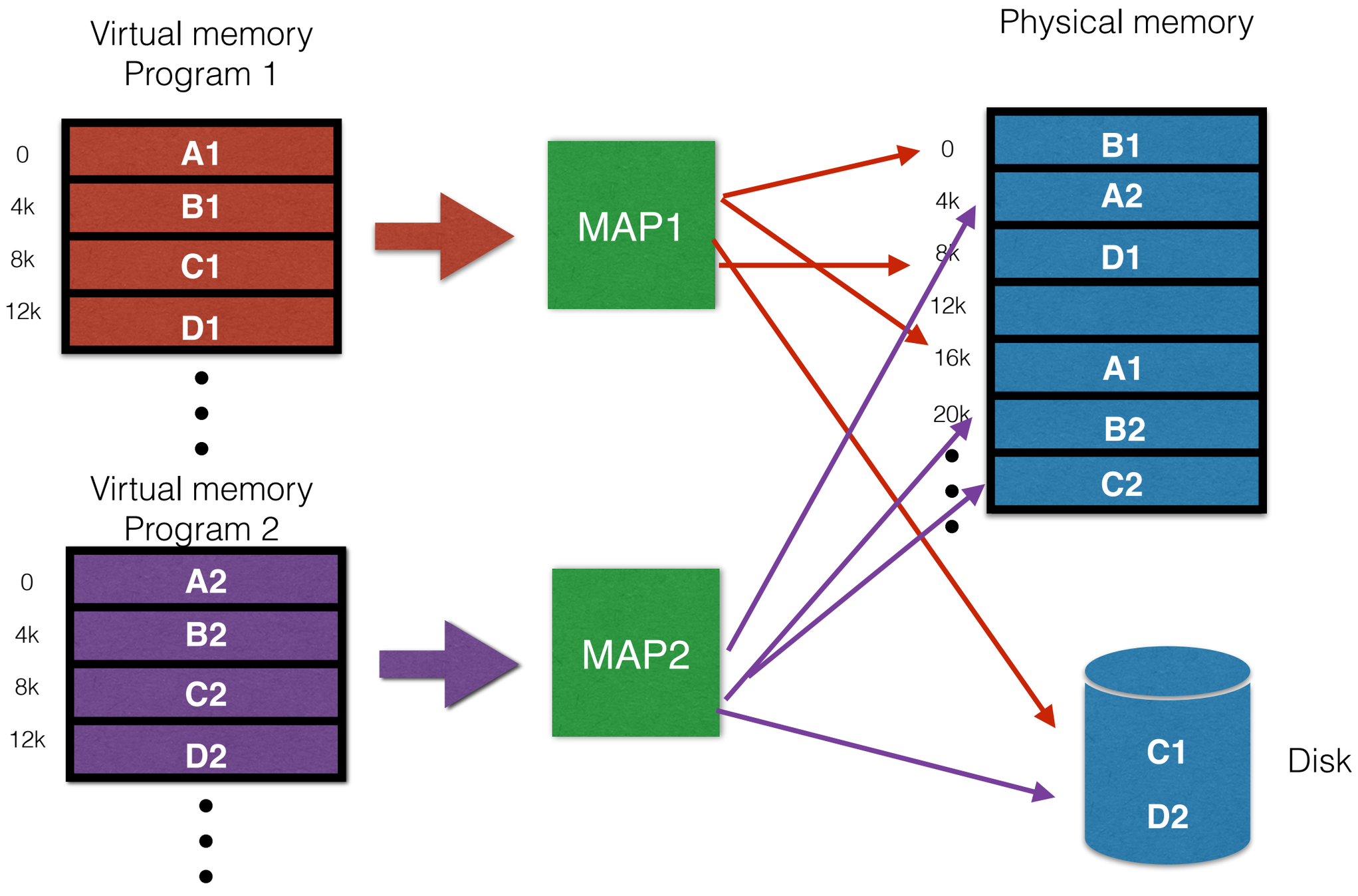
Virtual Memory: The main idea



Advantages of Virtual Memory

- You can run programs larger than physical memory [not too relevant anymore for normal programs…]
- You can run many, many programs at the same time without having to worry about diving the memory in different segments for different programs. Virtual memory takes care of that.
- Protection: A correct mapping will make sure that programs can’t interfere with each other memory.
- Code and data can be shared across different programs.

Virtual Memory for multiple programs



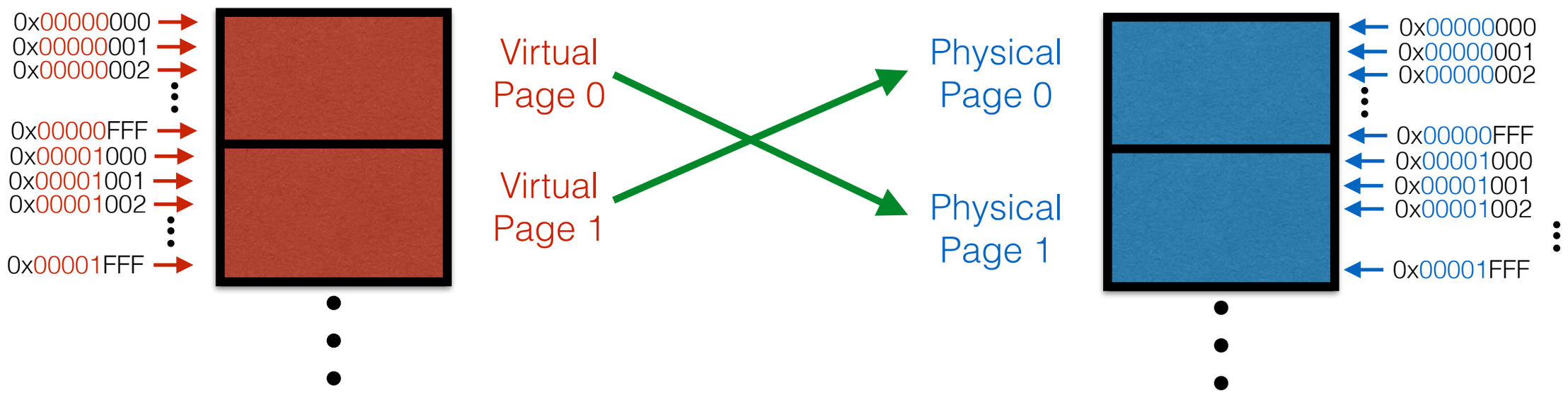
Virtual Memory

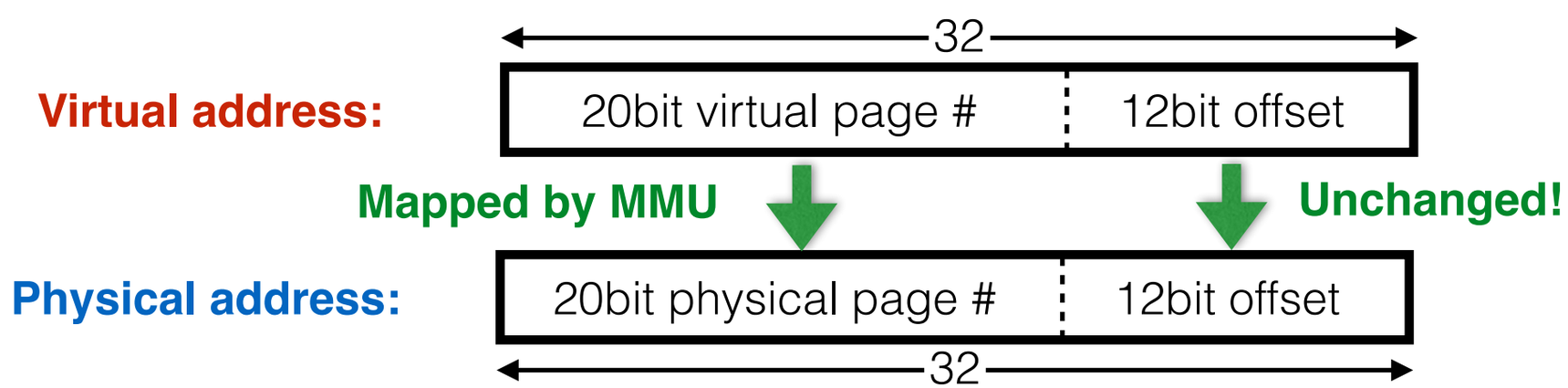
- How does paging works?
- How is the MAP defined?
- How can we speed up the search and reduce the size of the MAP?
- How does sharing/protection work?

Paging

- The address space is divided into fixed size chunks: **pages**.
- Previous implementations used variable sized chunks, called segments.

Example on 32bit machine: 4K pages:





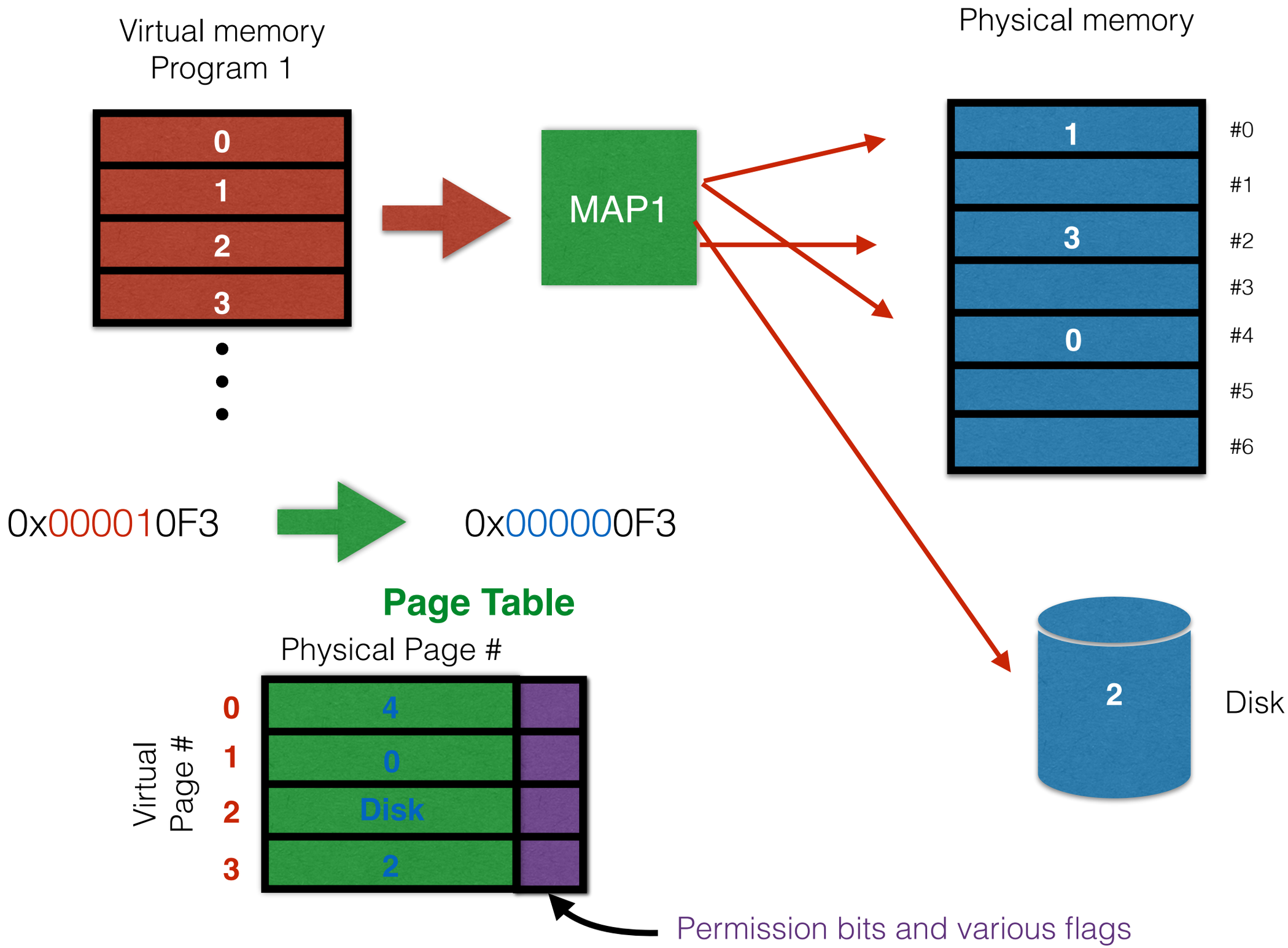
Memory Management Units (MMUs)

- MMUs are today integrated on chip with the CPUs.
- Each core will typically have one MMU for instruction and one for data access.
- Typically implementation for IA32: 4GB address space divided into 2^{20} pages of 4K each.
- Virtual and physical memory do not need to be the same size.

MMU and pages

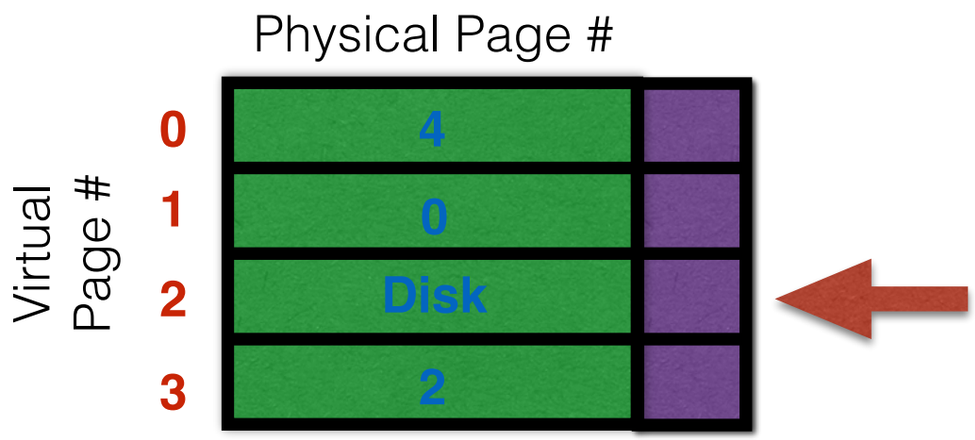
- Typically each process runs its own virtual address space.
- PTB register in MMU holds the physical address of page table associated to process currently running.
- Virtual pages are mapped to physical pages by the MMU – so that virtual addresses get translated into physical ones.
- Virtual pages can be:
    - Not allocated/mapped [the process hasn't accessed them yet]
    - Allocated in physical memory.
    - Allocated on disk.

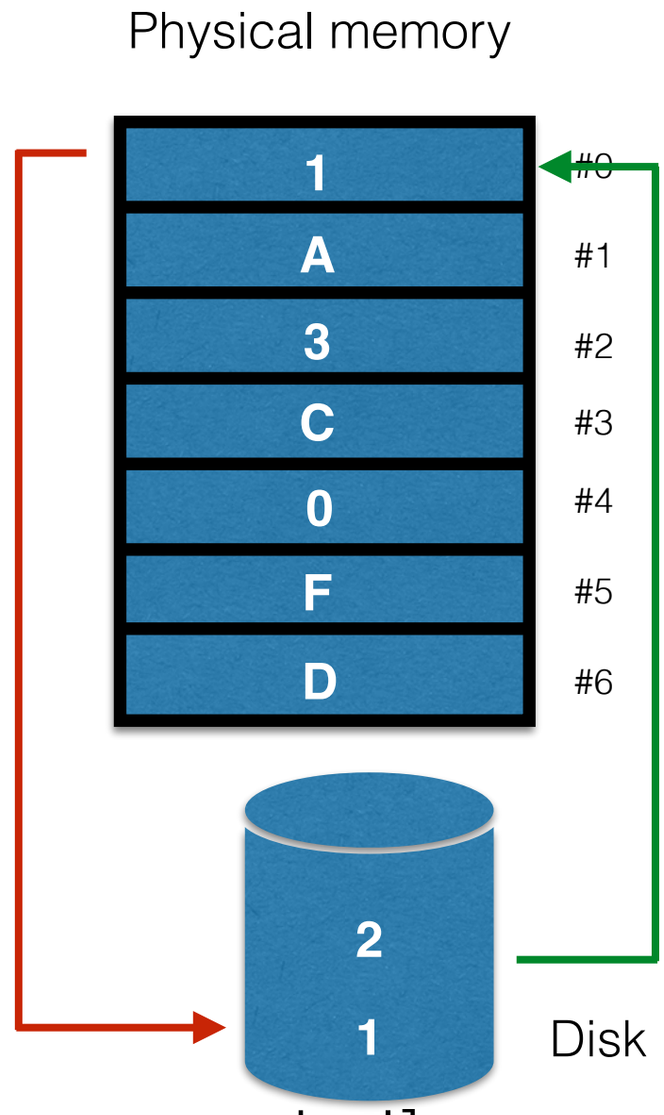
Page Tables


Virtual memory
Program 1
0
1
2
3
MAP1
Physical memory
1
3
0
#0
#1
#2
#3
#4
#5
#6
0x000010F3
0x000000F3
Page Table
Physical Page #
Virtual
Page #
0
1
2
3
4
0
Disk
2
Permission bits and various flags
2
Disk

Page Faults

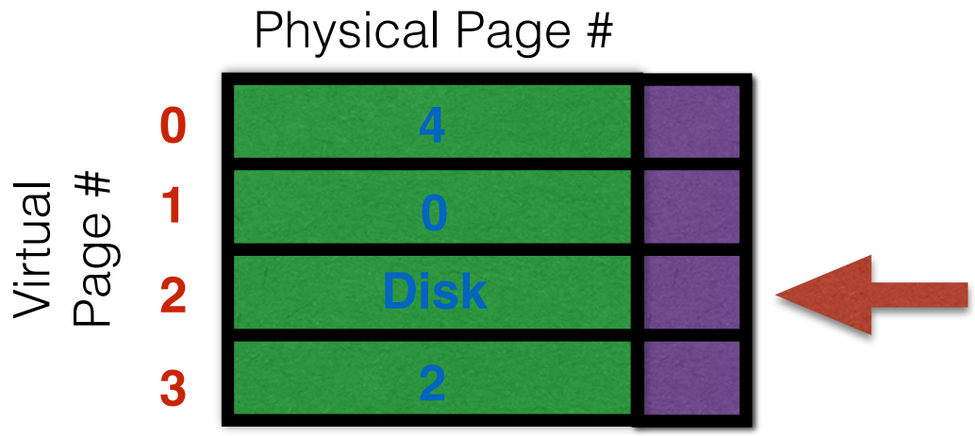
- What happens if our PTE says the page is on disk?

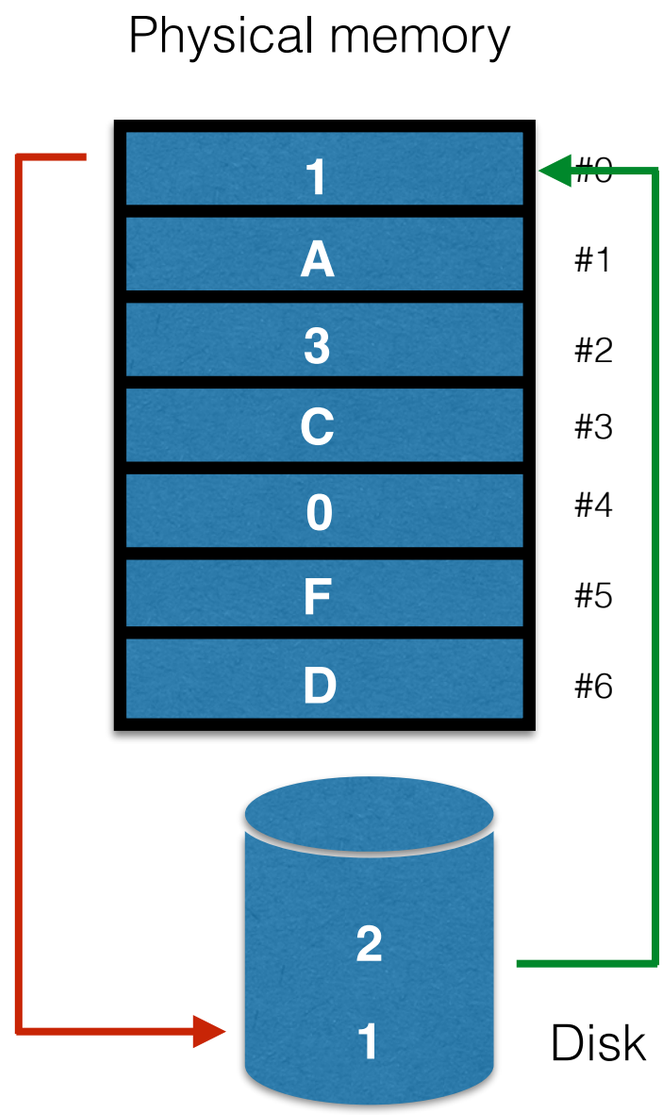




- The MMU rises a page fault exception.
- The OS's kernel takes control to resolve the page fault:
    1. OS choses a page to evict from RAM.
    2. If page is dirty, it will be written back to disk. [Very expensive!]
    3. The desired page is then written into RAM. [Very expensive!]
    4. The PTE is updated.
    5. The OS gives back control to the instruction that cause the page fault.

Page Faults – Why the OS?

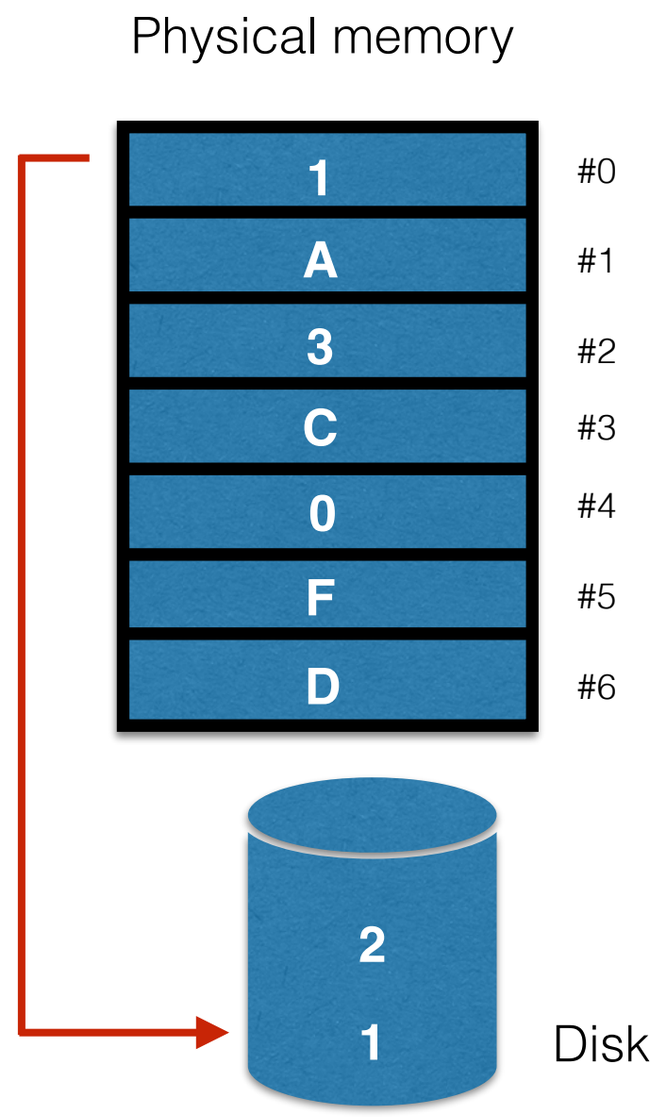


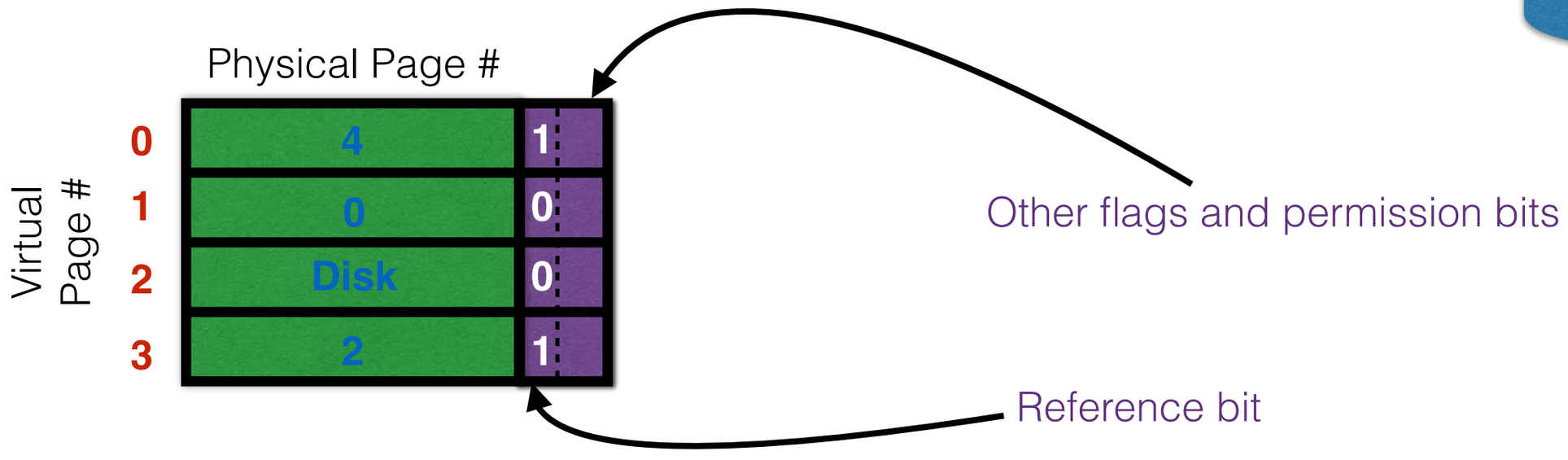


- Going to disk will take forever, it could be millions of cycle.
- The OS will typically chose another process to run for the meantime.
- This is why page fault are handled in software and not in hardware… Imagine what stalling the CPU would mean otherwise…

What page to evict from memory?

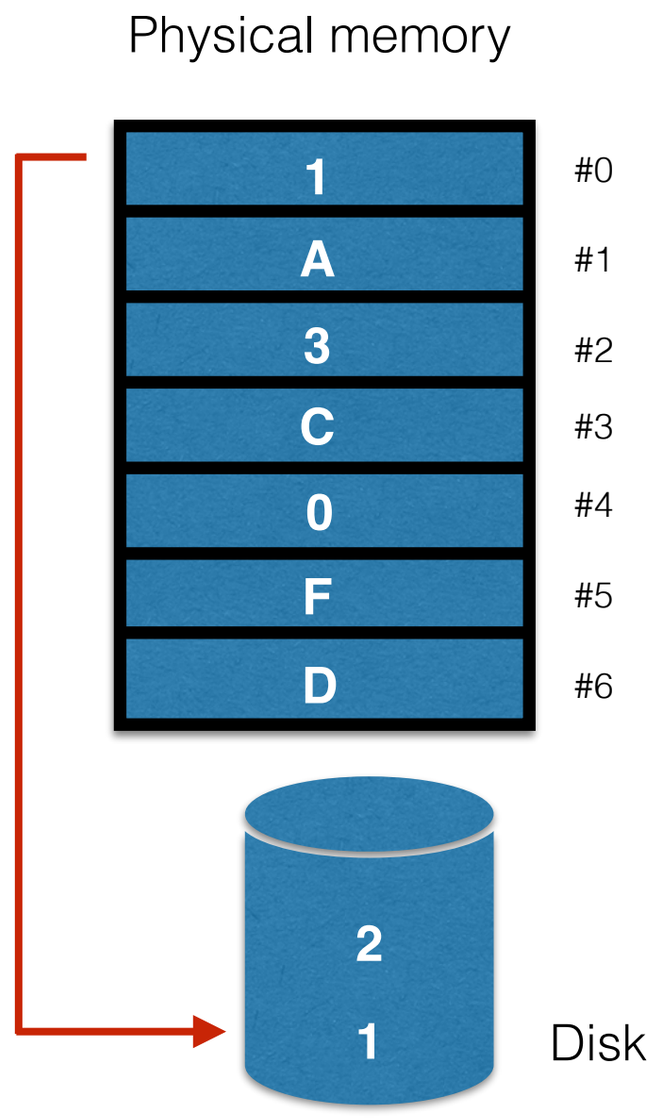
- The OS wants to minimise the number of access to disk. For this reason it tries to select for eviction the LRU (least-recently used) page.
- For this reason PTE have a reference (or use) bit. This is set by the OS each time the page is accessed [in reality only on translation miss].
- Periodically the OS clears all the reference bits in the PTE.
- Upon page fault it selects one Entry with R=0.





Do you really need to write it to Disk?

- Another quick optimisation generally built in modern CPUs is a check on whether you really need to write the page to disk.
- PTE have a dirty bit which signals to the OS whether the page has been modified since it has last be loaded from disk.
- The writing is done only if the dirty bit is zero.



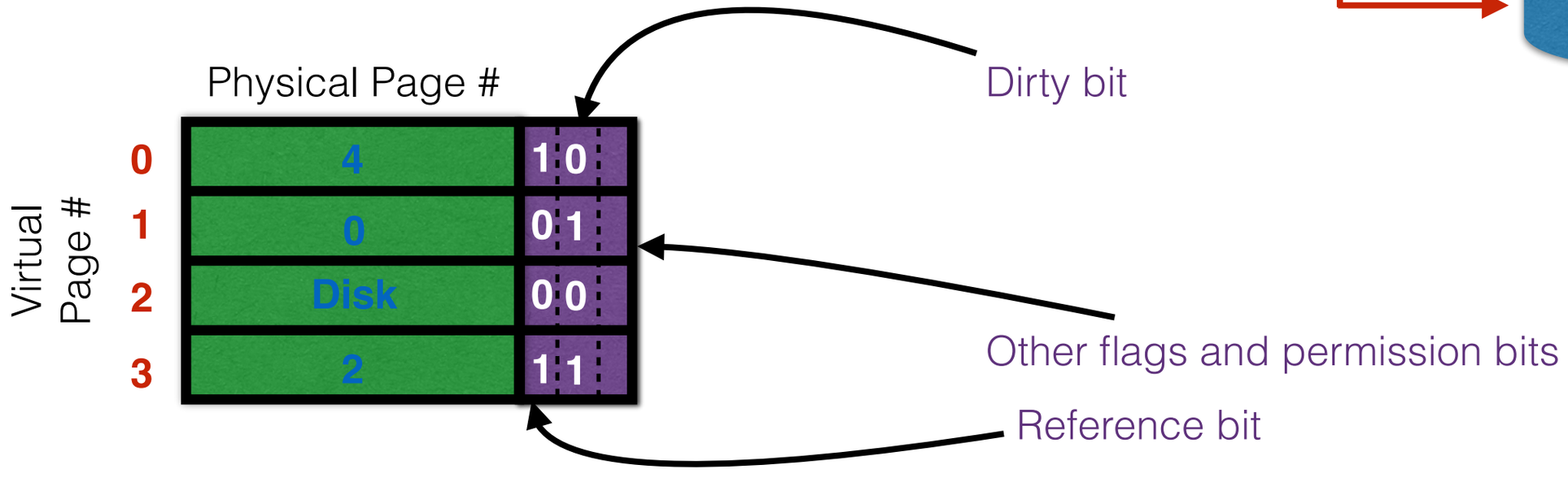
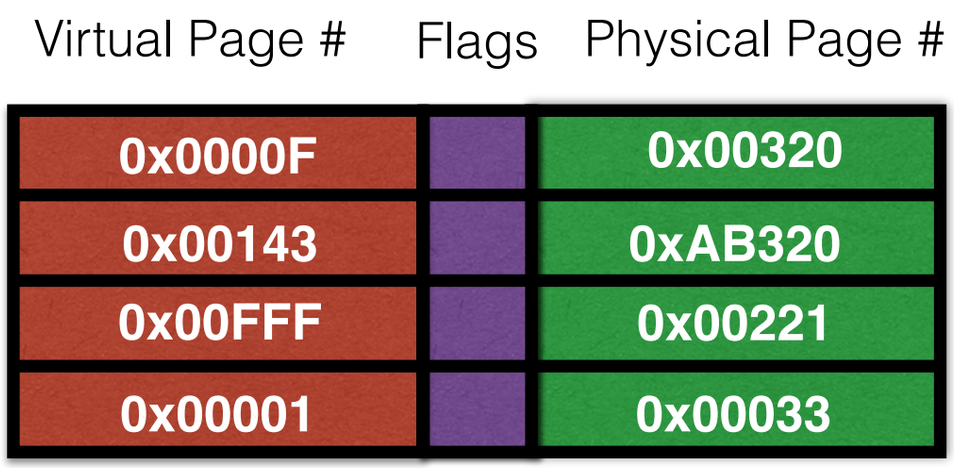


Table Lookaside Buffer (TLB)

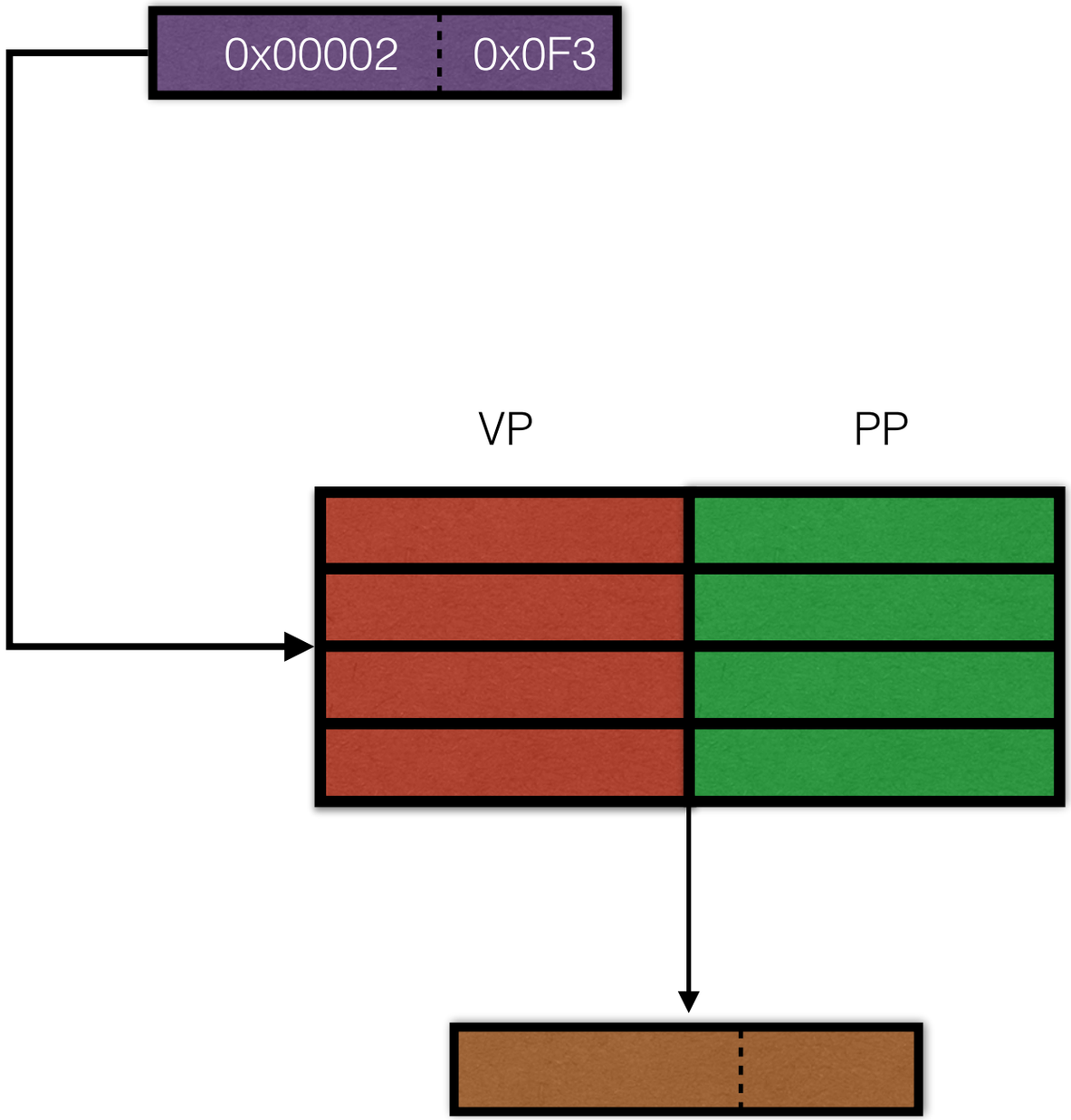
- Paging constraints us to investigate the full Page Table every time we want to access memory. Even w/o faults, this adds up a big overhead! [more than one million entries for a 32 bit machine!]

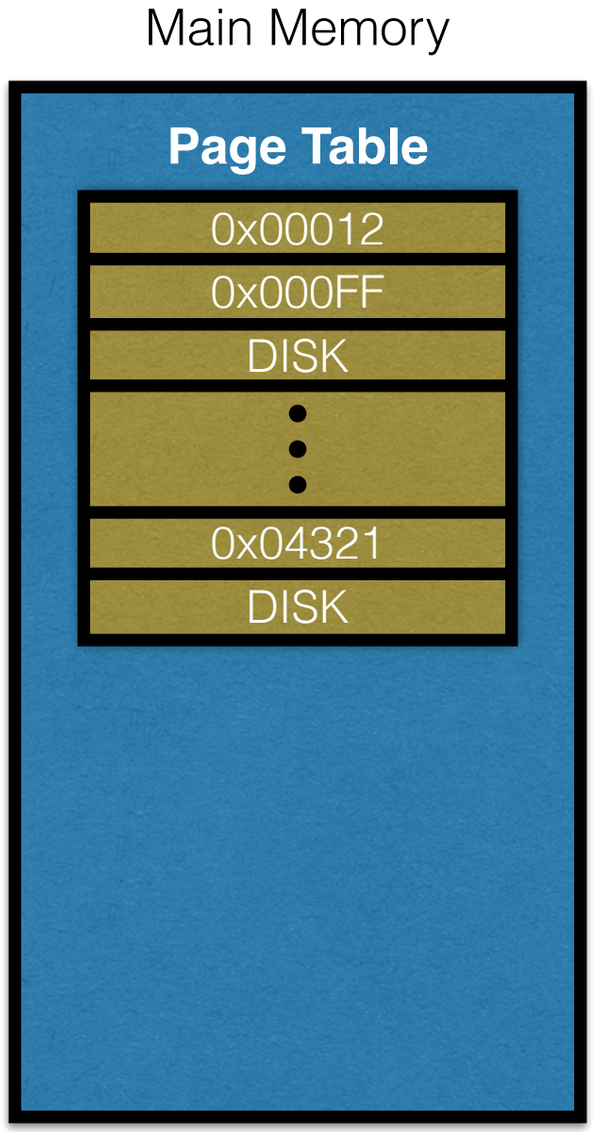
- This is mitigated with the introduction of a small cache for the Page Table – the **Table Lookaside Buffer**.



- The TLB is a small, m-way fully associative cache. Now if the VP is on the TLB the translation is quick.

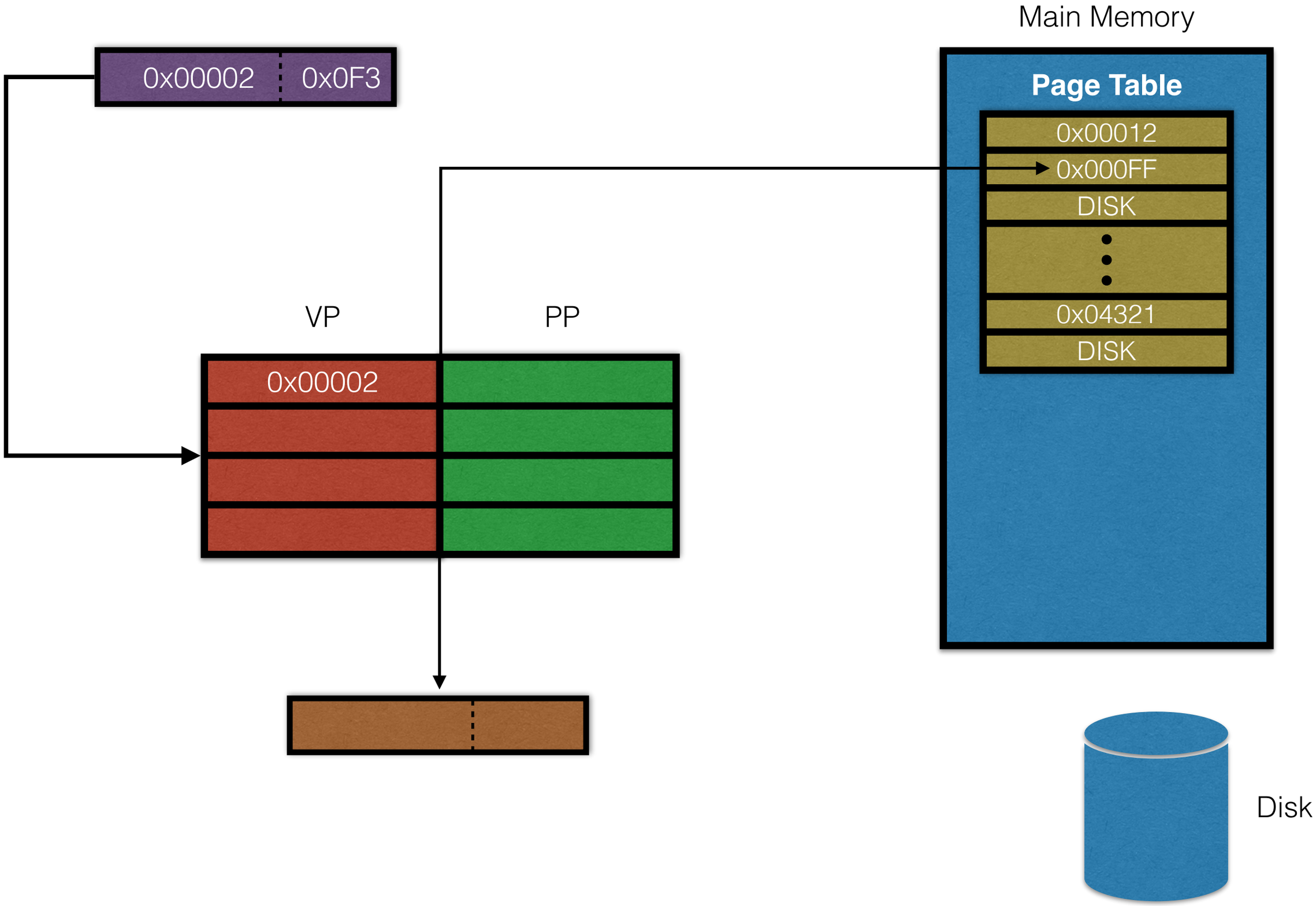
TLB example


0x00002
0x0F3
VP
PP

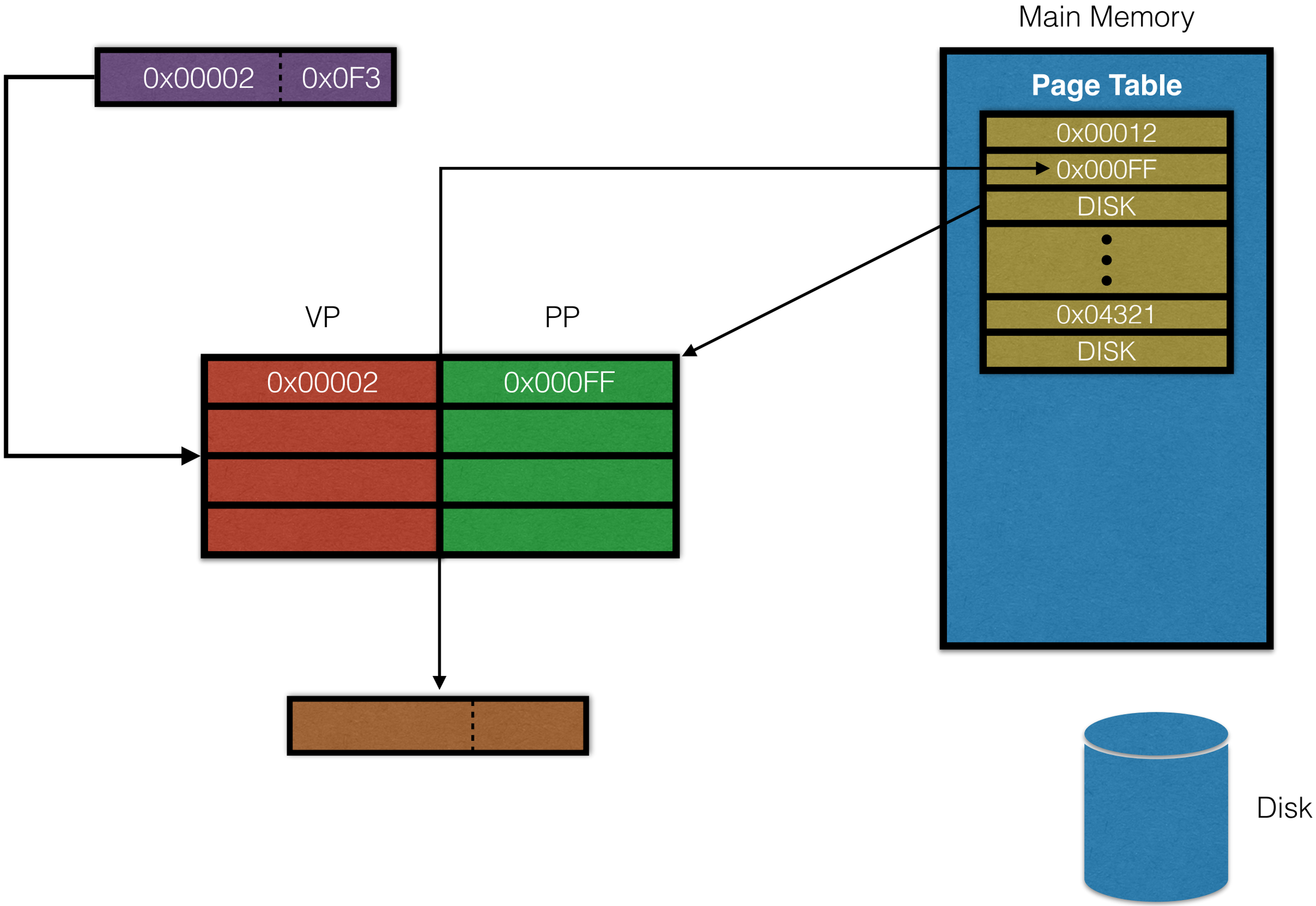

Main Memory
Page Table
0x00012
0x000FF
DISK
0x04321
DISK


Disk

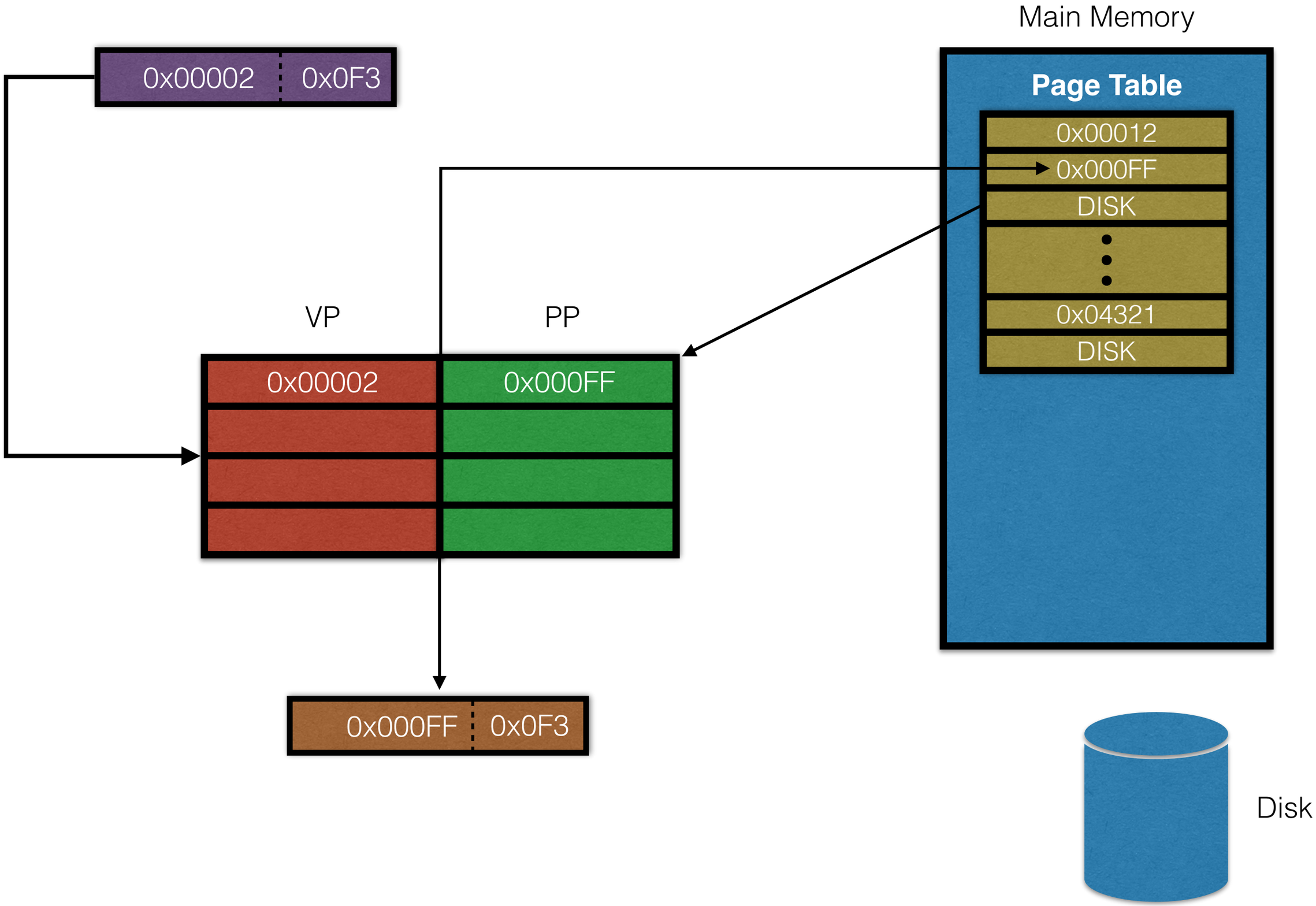
TLB example


Main Memory
0x00002
0x0F3
VP
PP
0x00002
Page Table
0x00012
0x000FF
DISK
0x04321
DISK
Disk

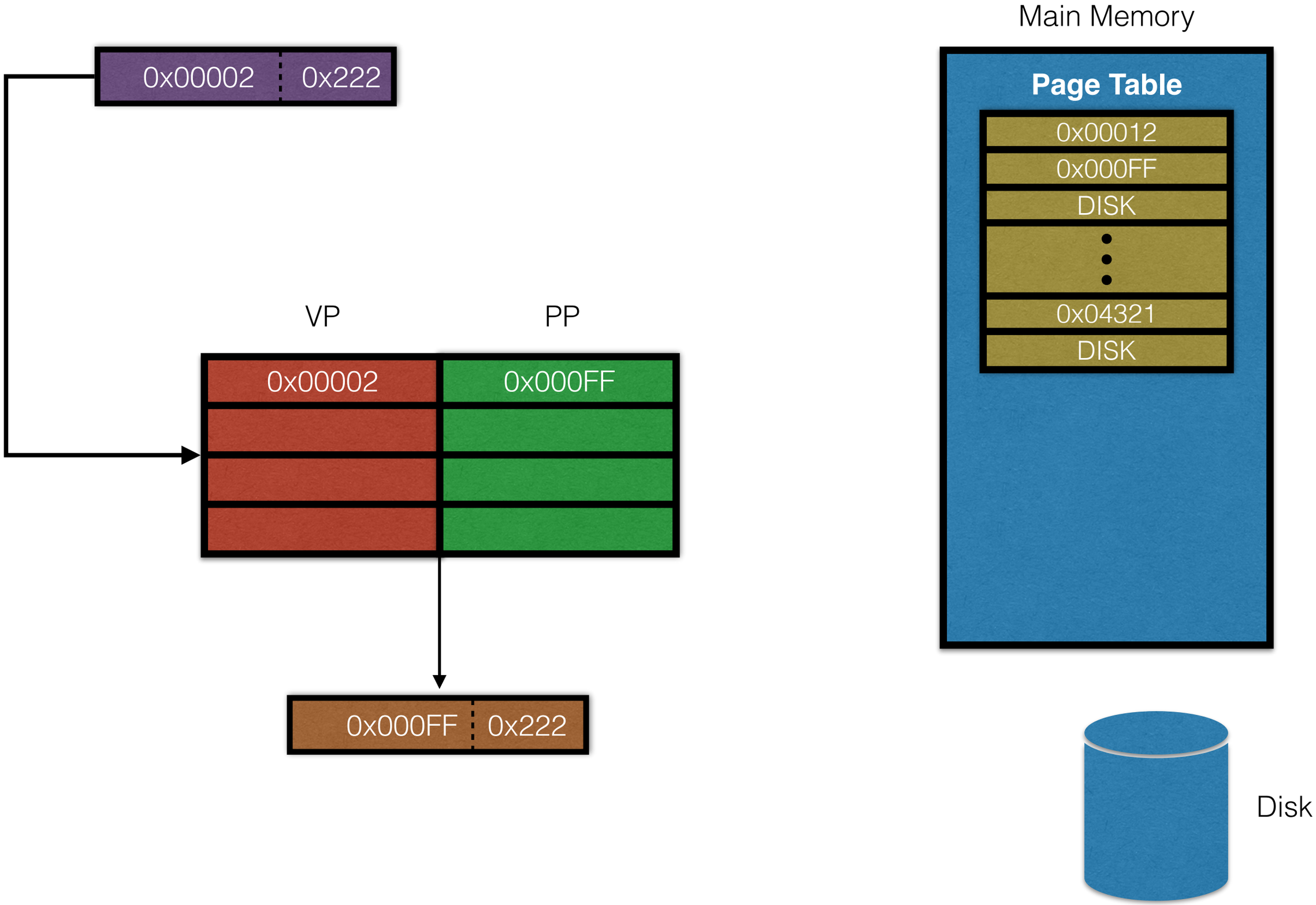
TLB example


Main Memory
0x00002
0x0F3
Page Table
0x00012
0x000FF
DISK
0x04321
DISK
VP
PP
0x00002
0x000FF
Disk

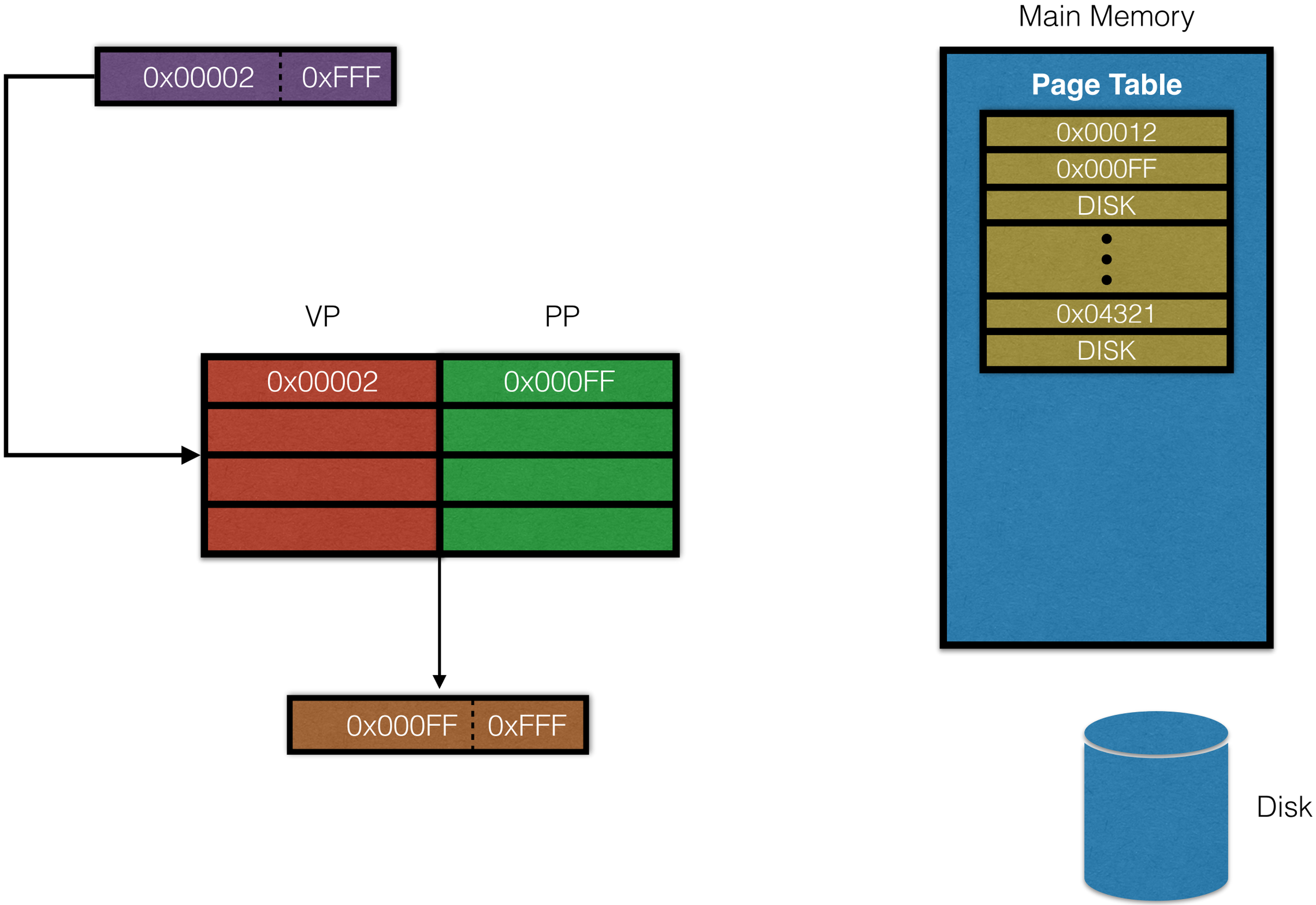
TLB example


Main Memory
0x00002
0x0F3
Page Table
0x00012
0x000FF
DISK
0x04321
DISK
VP
PP
0x00002
0x000FF
0x000FF
0x0F3
Disk

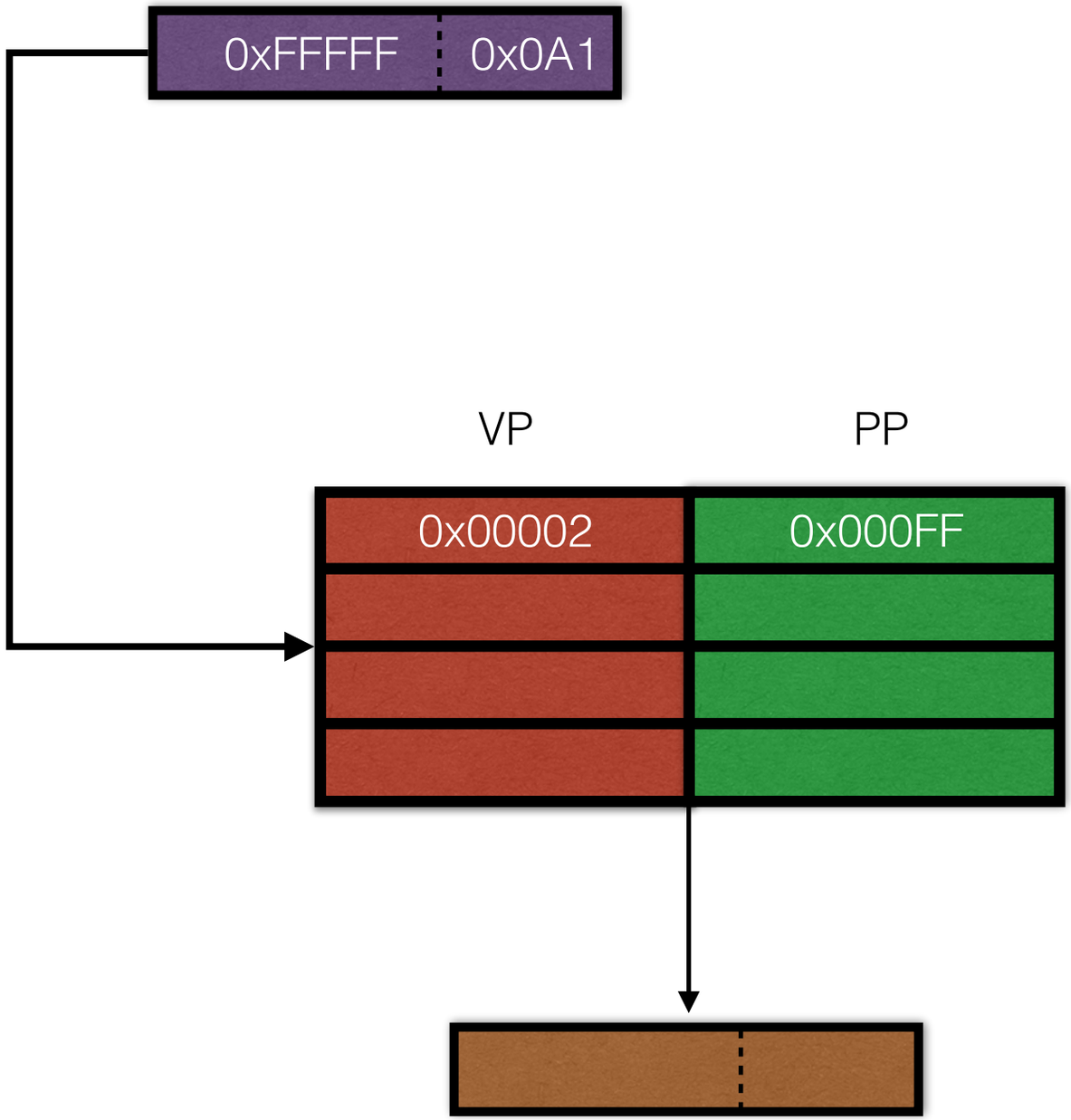
TLB example

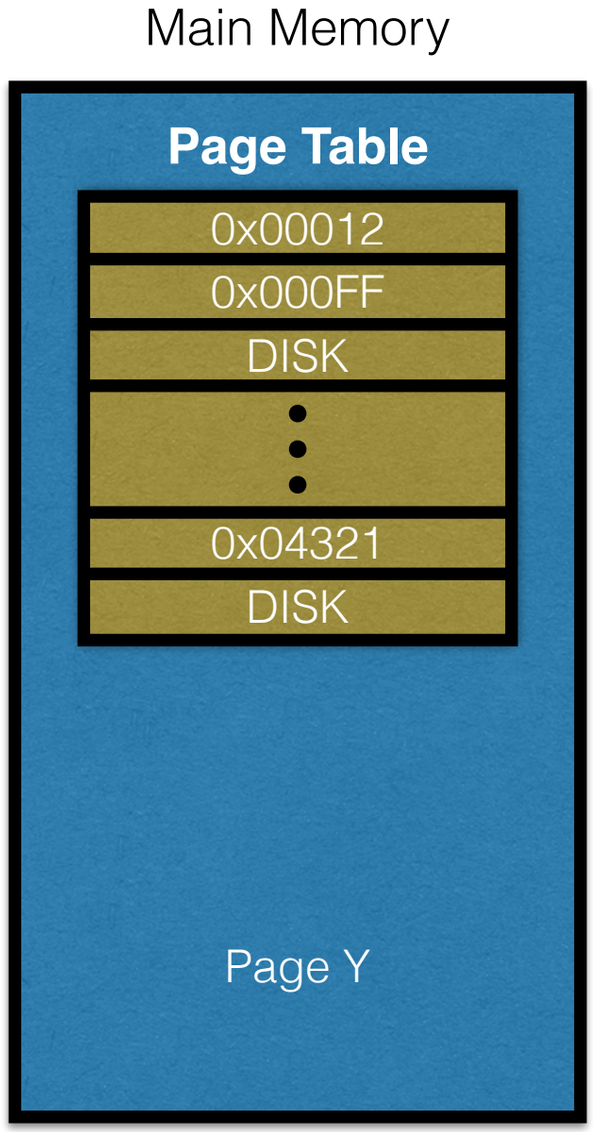

0x00002
0x222
VP
PP
0x00002
0x000FF
0x000FF
0x222
Main Memory
Page Table
0x00012
0x000FF
DISK
0x04321
DISK
Disk

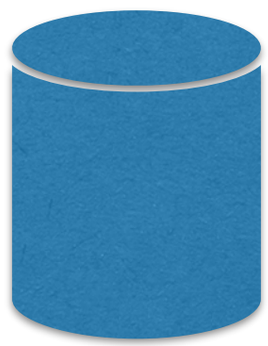
TLB example


0x00002
0xFFF
VP
PP
0x00002
0x000FF
0x000FF
0xFFF
Main Memory
Page Table
0x00012
0x000FF
DISK
0x04321
DISK
Disk

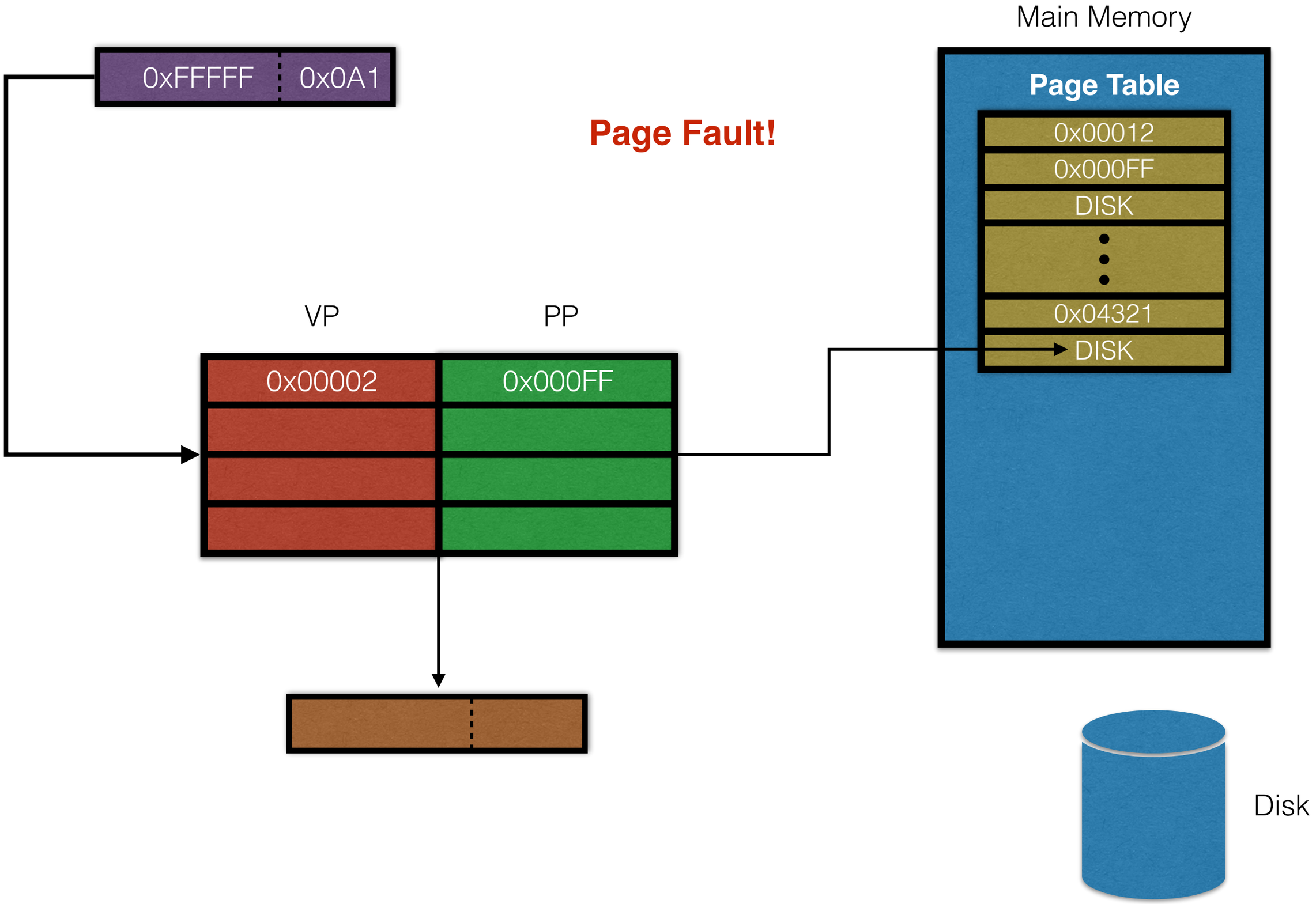
TLB example


0xFFFFF
0x0A1
VP
PP
0x00002
0x000FF

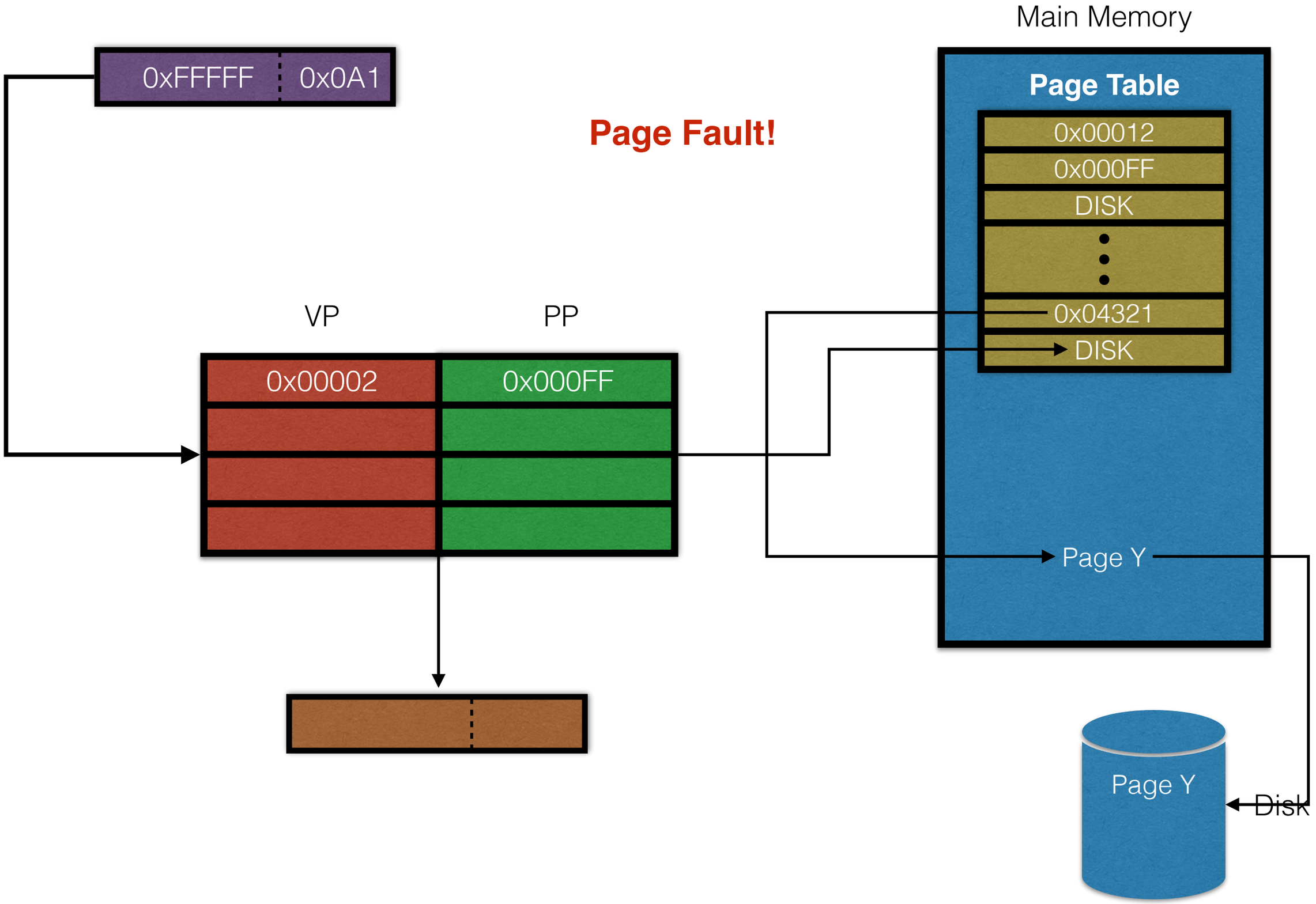

Main Memory
Page Table
0x00012
0x000FF
DISK
0x04321
DISK
Page Y


Disk

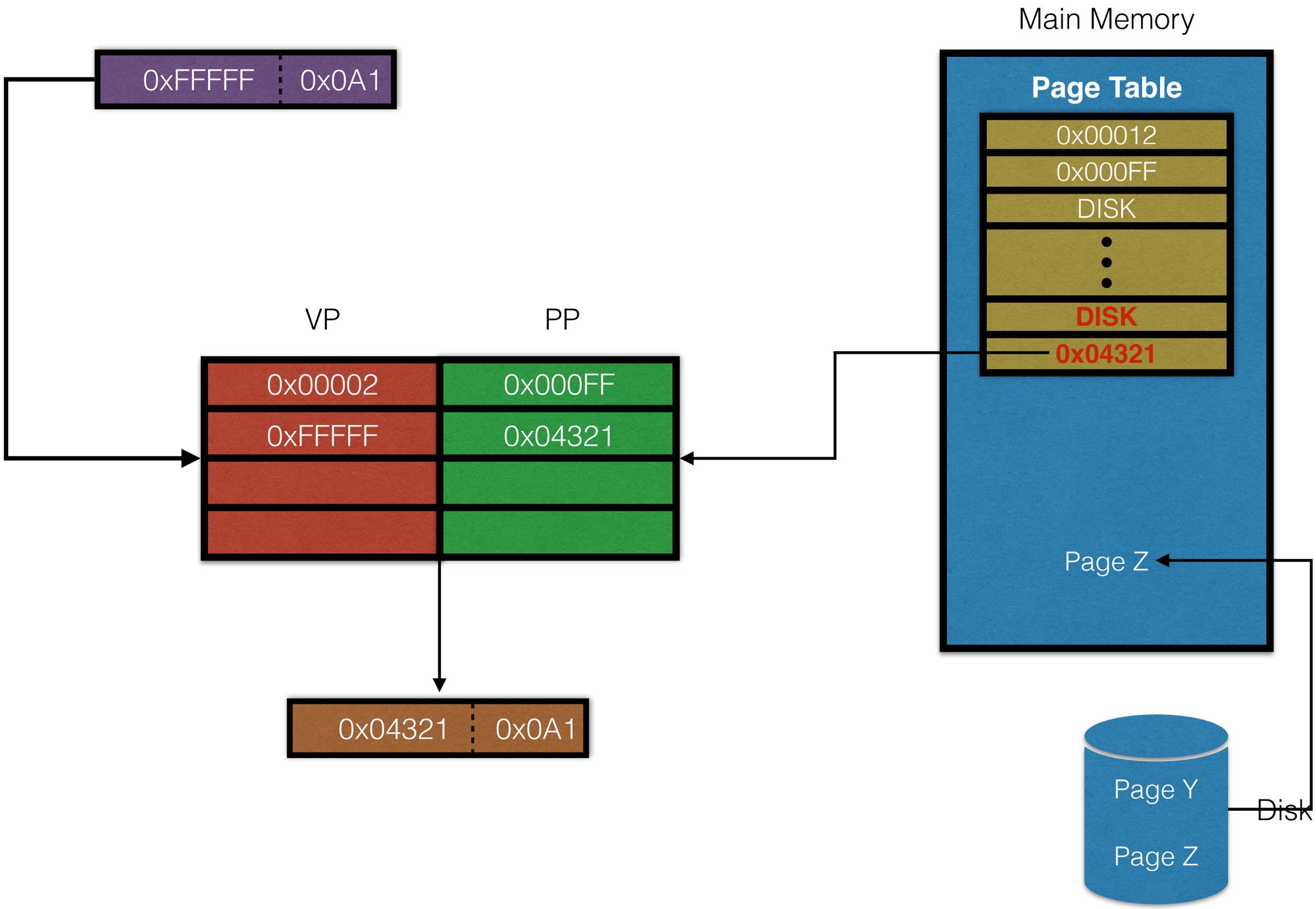
TLB example


0xFFFFF
0x0A1
Page Fault!
Main Memory
Page Table
0x00012
0x000FF
DISK
0x04321
DISK
VP
PP
0x00002
0x000FF
Disk

TLB example


Main Memory
0xFFFFF
0x0A1
Page Fault!
Page Table
0x00012
0x000FF
DISK
0x04321
DISK
VP
PP
0x00002
0x000FF
Page Y
Page Y
Disk

TLB example

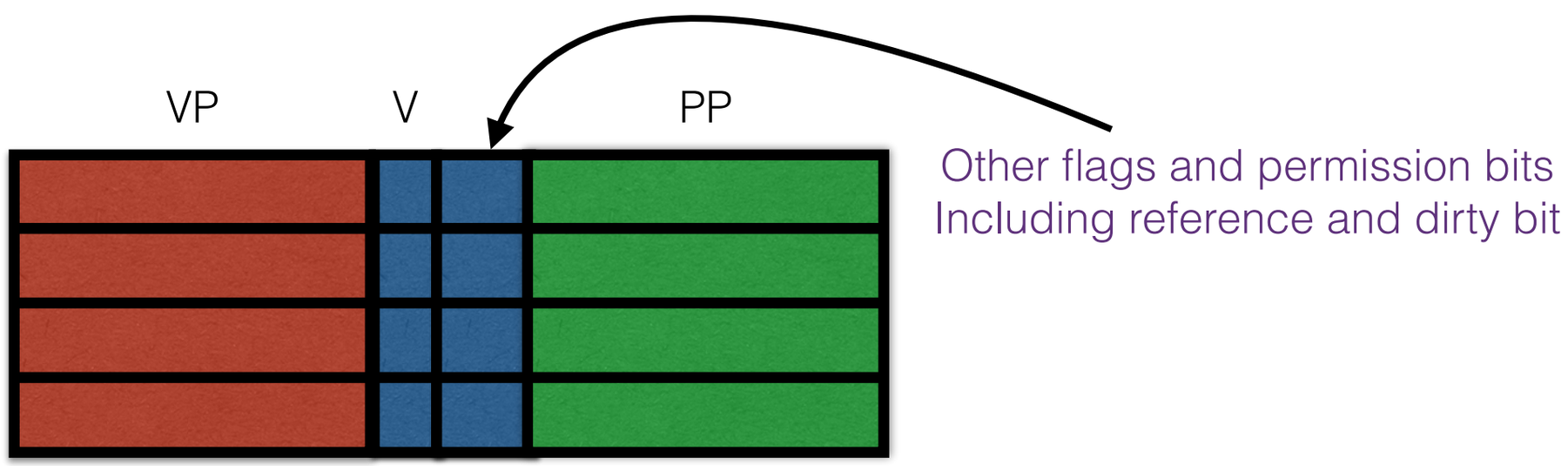

0xFFFFF
0x0A1
VP
PP
0x00002
0x000FF
0xFFFFF
0x04321
0x04321
0x0A1
Main Memory
Page Table
0x00012
0x000FF
DISK
DISK
0x04321
Page Z
Page Y
Page Z
Disk

TLBs

- Separate instructions and data TLBs are generally used.
- TLBs are typically very small (around 64 entries for 4K pages – versus 1Million PTE!).
- Typically, implementation today have >90% hit rate.
- A modern CPU will typically have one MMU for data and one for instruction (needed for parallel access to data and instruction cache)
- Some modern processors will have two levels of TLBs to reduce the missing rate [similar concept to two level caches].

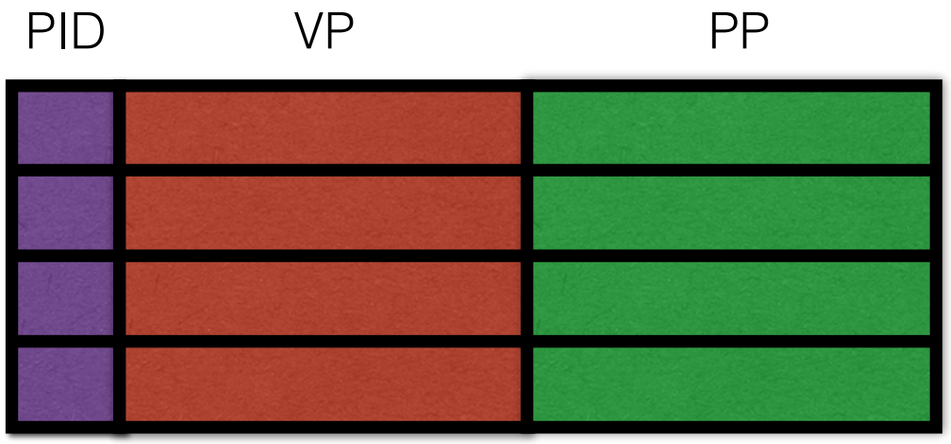
TLB Coherency

- It is the responsibility of the OS to maintain the coherency of the TLB.
- If a PTE is changed in the RAM (as result of page fault) the OS must, if needed, do the same update to the TLB (if entry is in TLB).
- TLBs have a valid flag indicating whether the entry is valid or not.
- CPUs have instruction for the purpose [e.g. IA32 “INVLPG va”]



Multiple Processes and TLBs

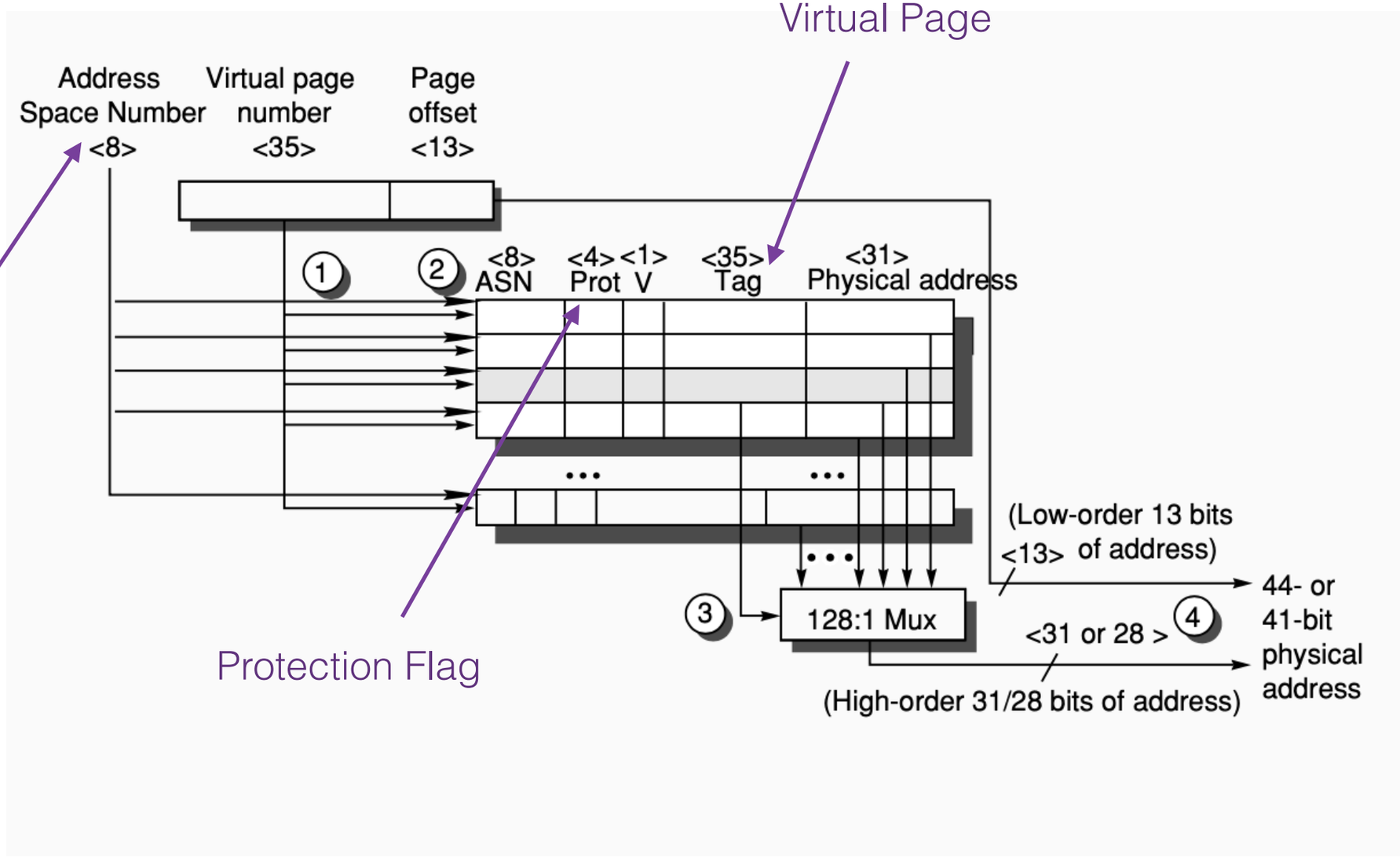
- What happens to the TLB upon process switching?
- Well, each process has its own virtual address, and they are all equal! TLB entries are not valid anymore, they would refer to different page tables!



- Process ID is attached to the VP – and handled by the OS.
- If new process use old PID, all previous entries need to be invalidated too.

TLB implementation example

- TLB in Alpha 21264 (taken from PH - 2nd edition).



Size of Page Tables

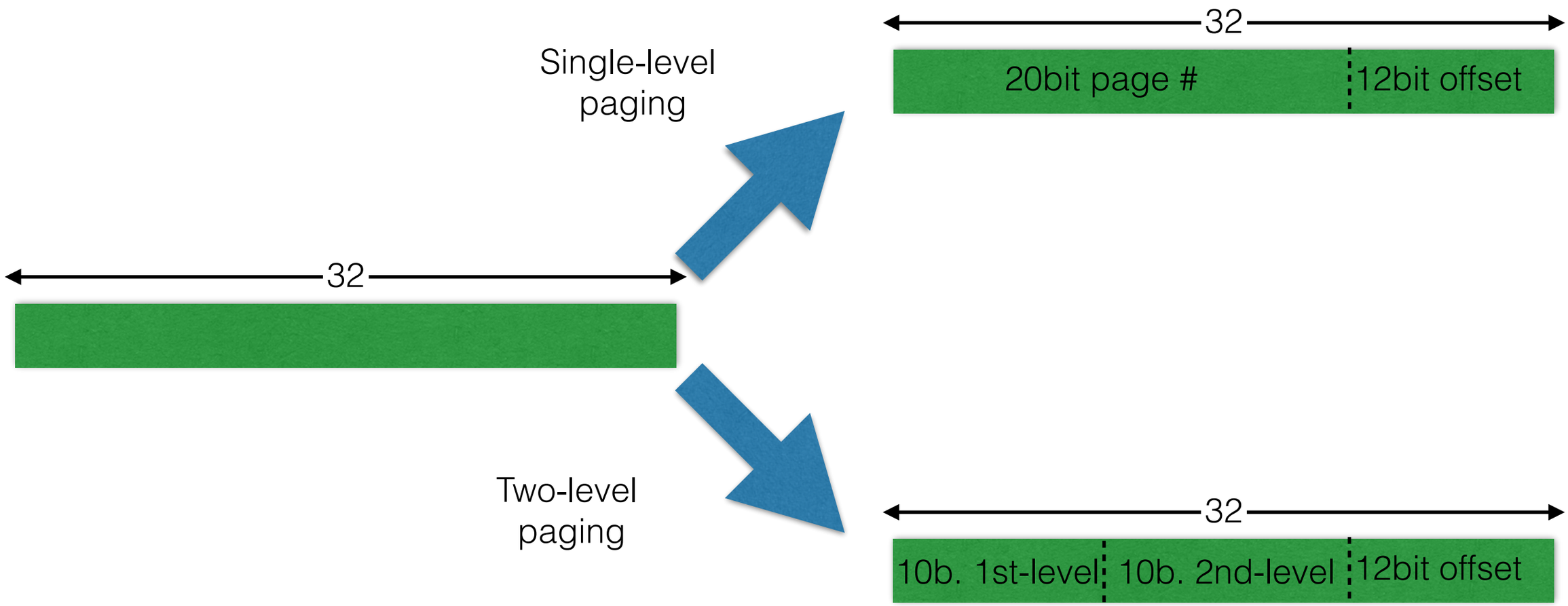
- 32-bit machine, with 4K pages —> Page Table is of size (at least) 4MB.
- If you have 100 process (definitely a lower bound!). This would mean 100 Page tables —> 400 MB that needs to be in RAM at all times! Not Ideal!

Size of Page Tables

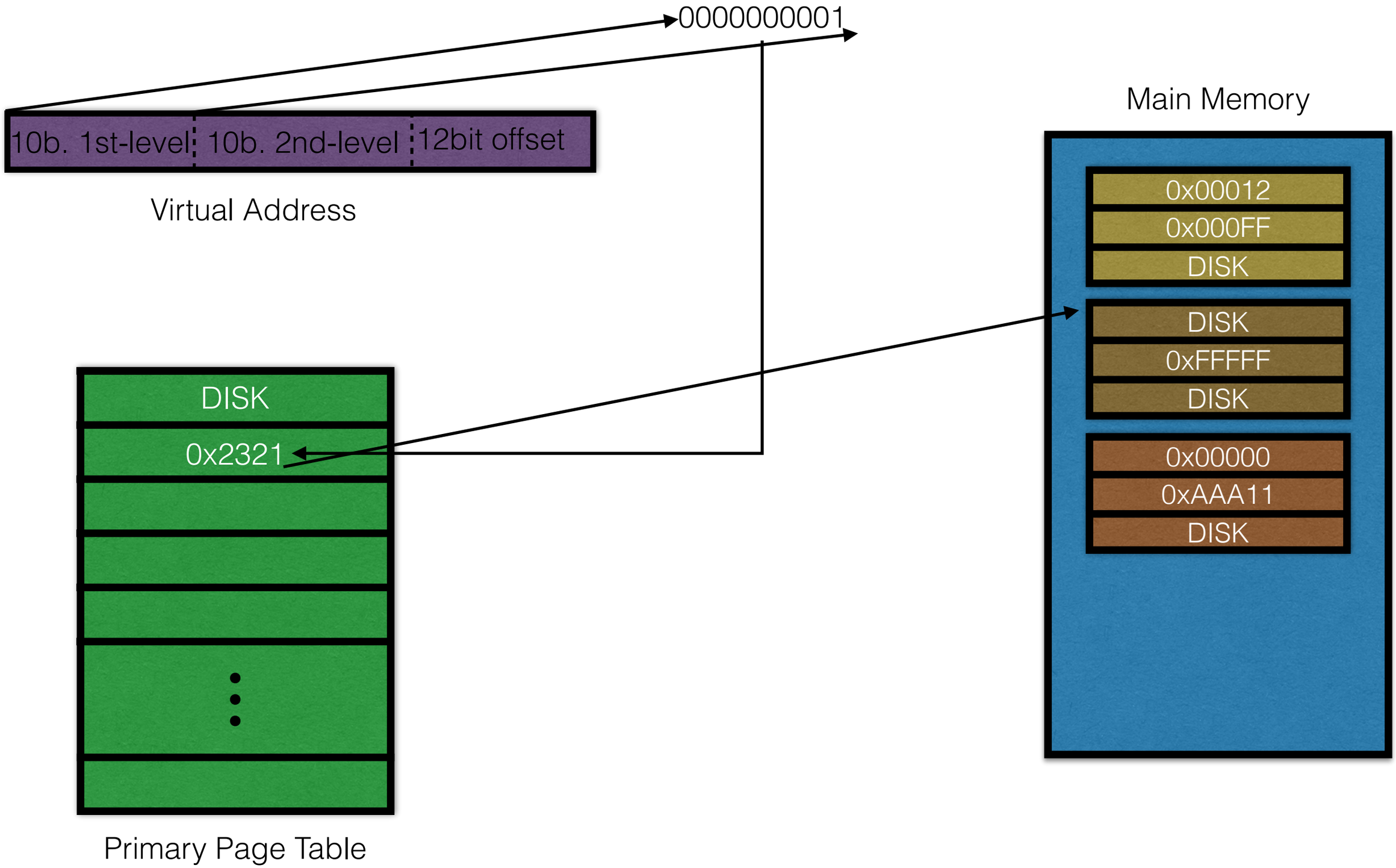
- An alternative could be playing with page sizes:
- Making each page bigger will decrease the number of PTE.
- Will also increase TLB hit rates.
- However it will also increase the internal fragmentation of memory [1.5 times the page size…].
- Process startup will also be slower.

Multi-Level Page Tables

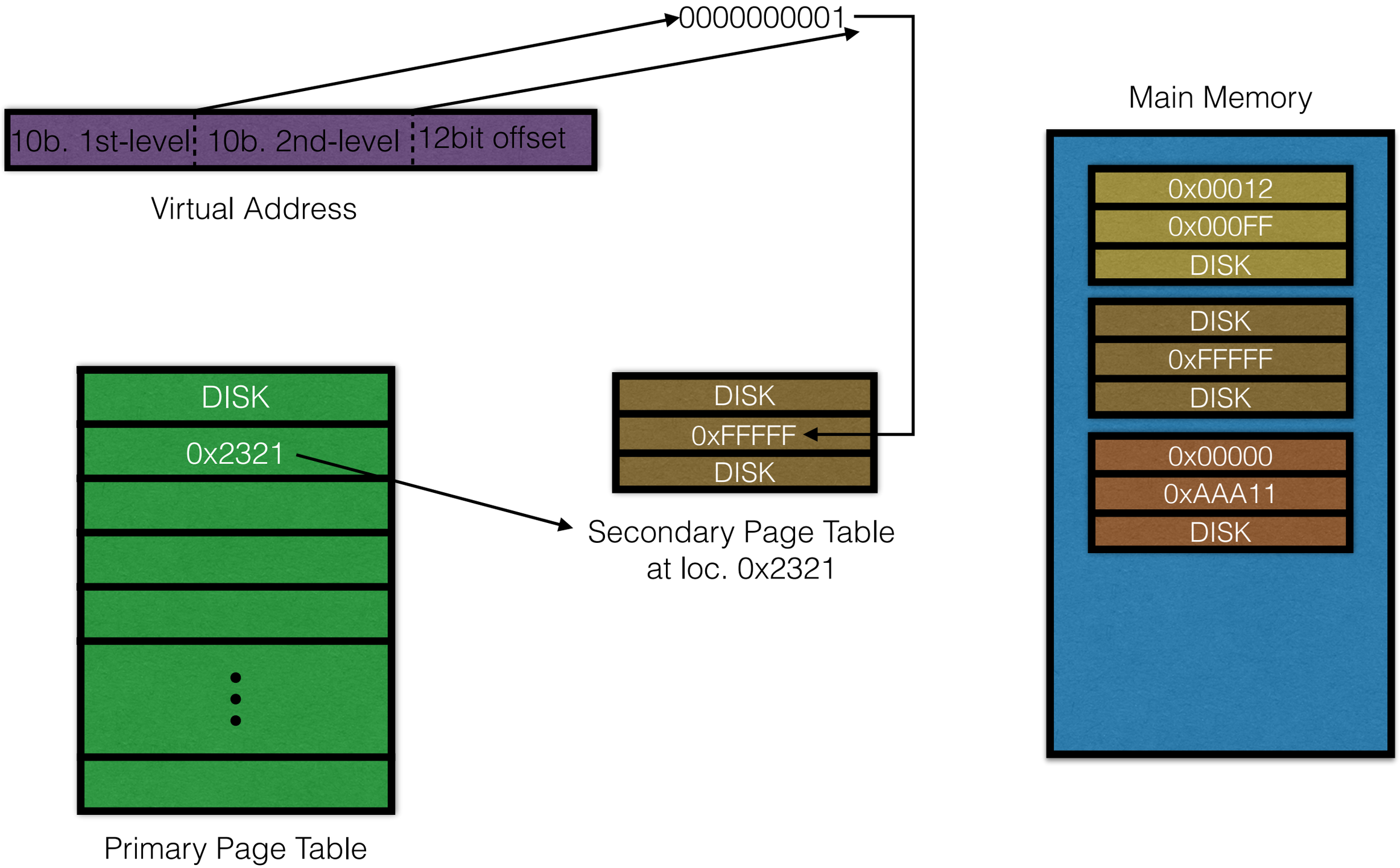
- Very similar to what we have already done with paging in the first place…



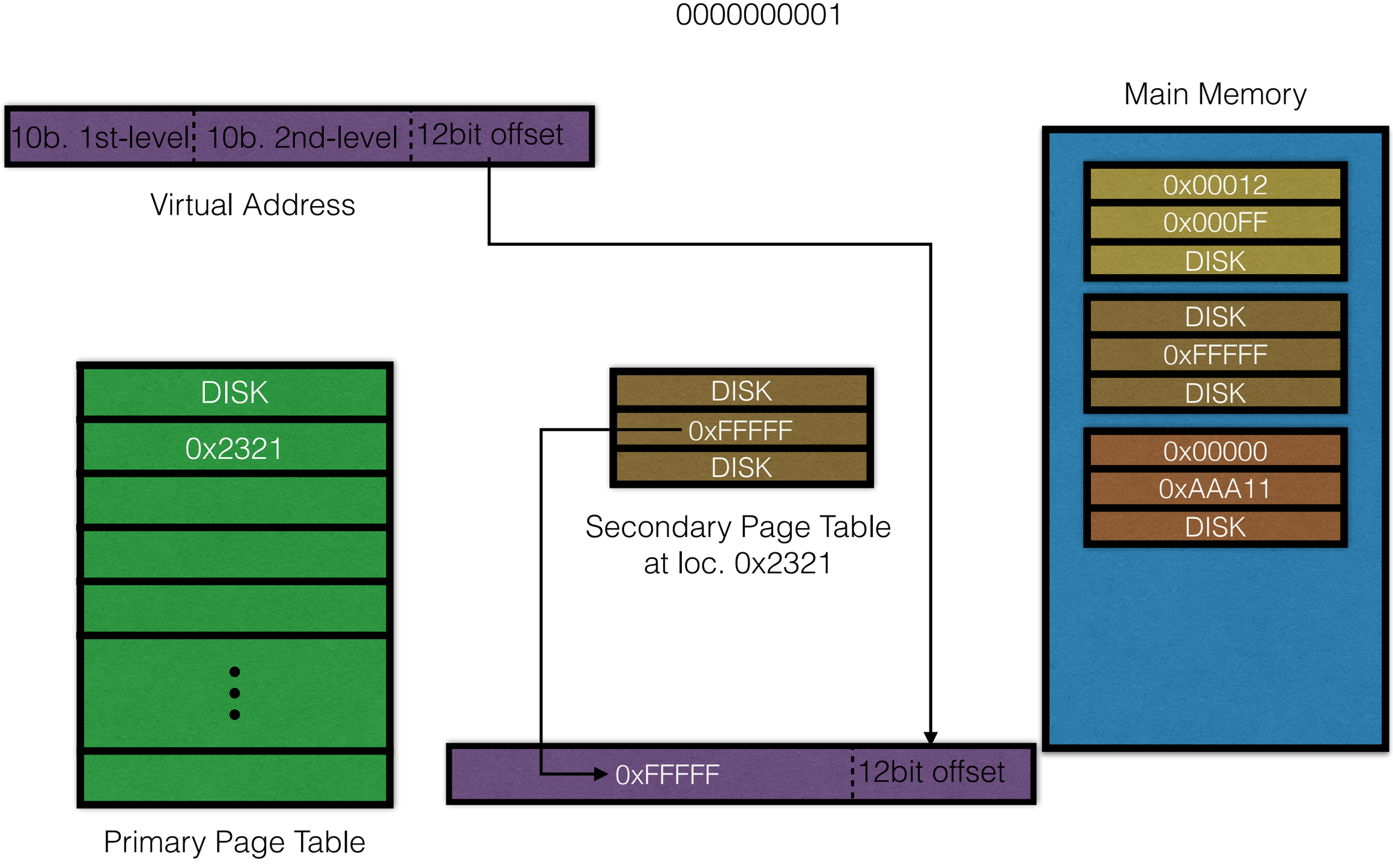
2-level page Tables



2-level page Tables



2-level page Tables



2-Level page table

- With 2 level page tables, every page takes only 4K.
- Now for a single running process the minimum I need is only 8K in RAM (all the rest can be stored in disk).
- Much more manageable! What's the cost though?

Support for Different Page Sizes

- Some MMU try to get the best of both world by allowing for adaptable Page Sizes.

- The advantage is that in this way, for some entries, the TLB could translate large virtual memory into correspondingly large contiguous physical memory location.

- For example, the OS could be loaded into a large contiguous area of memory, which could then be accessed by using a single TLB entry.

- IA32 for example includes an extra bit in the primary PTE which flags whether the entry points to a secondary level page table or points to a 2^22=4 MB chunk of physical memory location.

Breakpoint registers

- Modern MMUs have a number of breakpoint address registers and breakpoint control registers.
- These are used to trigger interrupts if the breakpoint address is read/written or executed.
- Debuggers typically operate by setting breakpoints at virtual addresses.
- MMU breakpoint registers are part of the process state.
- This is how Linux ptrace syscall works.

MMU/OS interface

- The Interface between the MMU and the OS is not always clear. Many MMUs for a single OS, and many OSs for a single MMU…

- Generally PTE have bits which can be accessed and modified only by the OS.

- IA32 PTEs generally have 3 such bits. Those are used by the OS to interface with the MMU operation.

- For example, a simplified hypothetical UNIX kernel could use 2 of such spare bits to define 4 different operations mode when PTE is valid, and another for when PTE is not valid

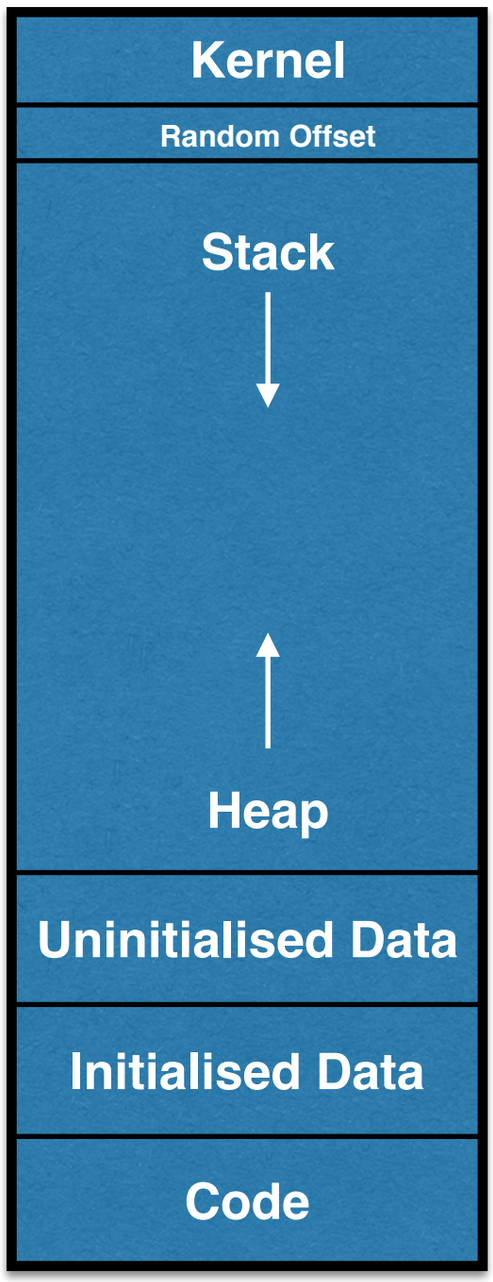
PTE is Valid

- `MEM`: the PTE maps a VA to a PA.
- `LOCK`: the PTE maps a VA to a PA, + is locked in physical memory [hint not to evict…].
- `SPY`: the PTE maps a VA to a particular PA.

PTE is Invalid

- `NULL`: Page not yet mapped into physical memory.
- `DISK`: Page not yet mapped into physical memory & upon mapping, page must be initialised using disk data. The PTE contains a disk block number.
- `IOP`: Disk Input/output is currently in progress.
- `SPT`: PTE is shared across multiple processes.

Memory Mapping for Unix/ Microsoft Process



Often memory segmented so that the Kernel always has an allocated space in virtual memory [Useful for protection and for sharing]

[Handled directly by the program]

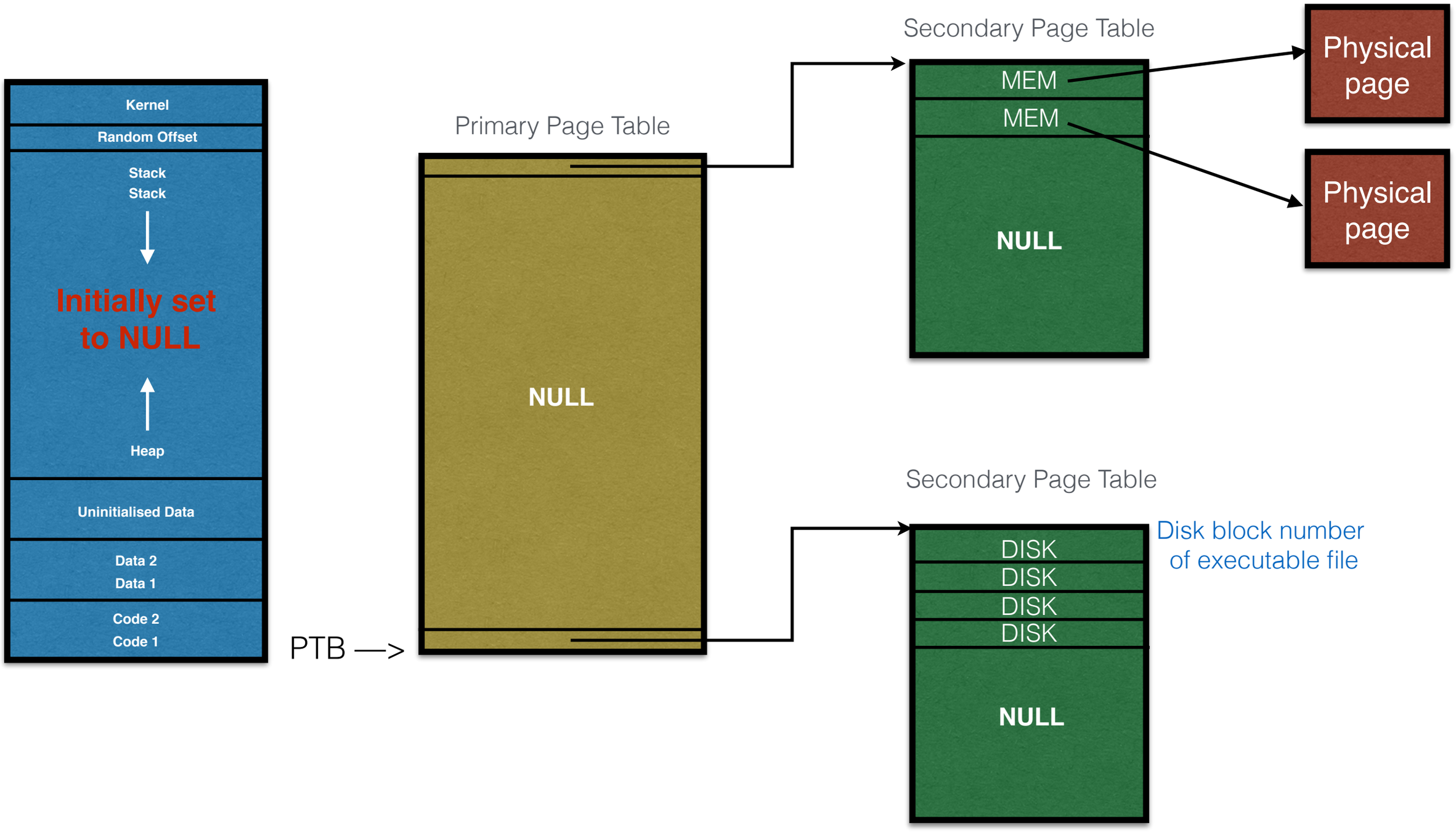
[Allocated through call to the OS]

Read from the executables

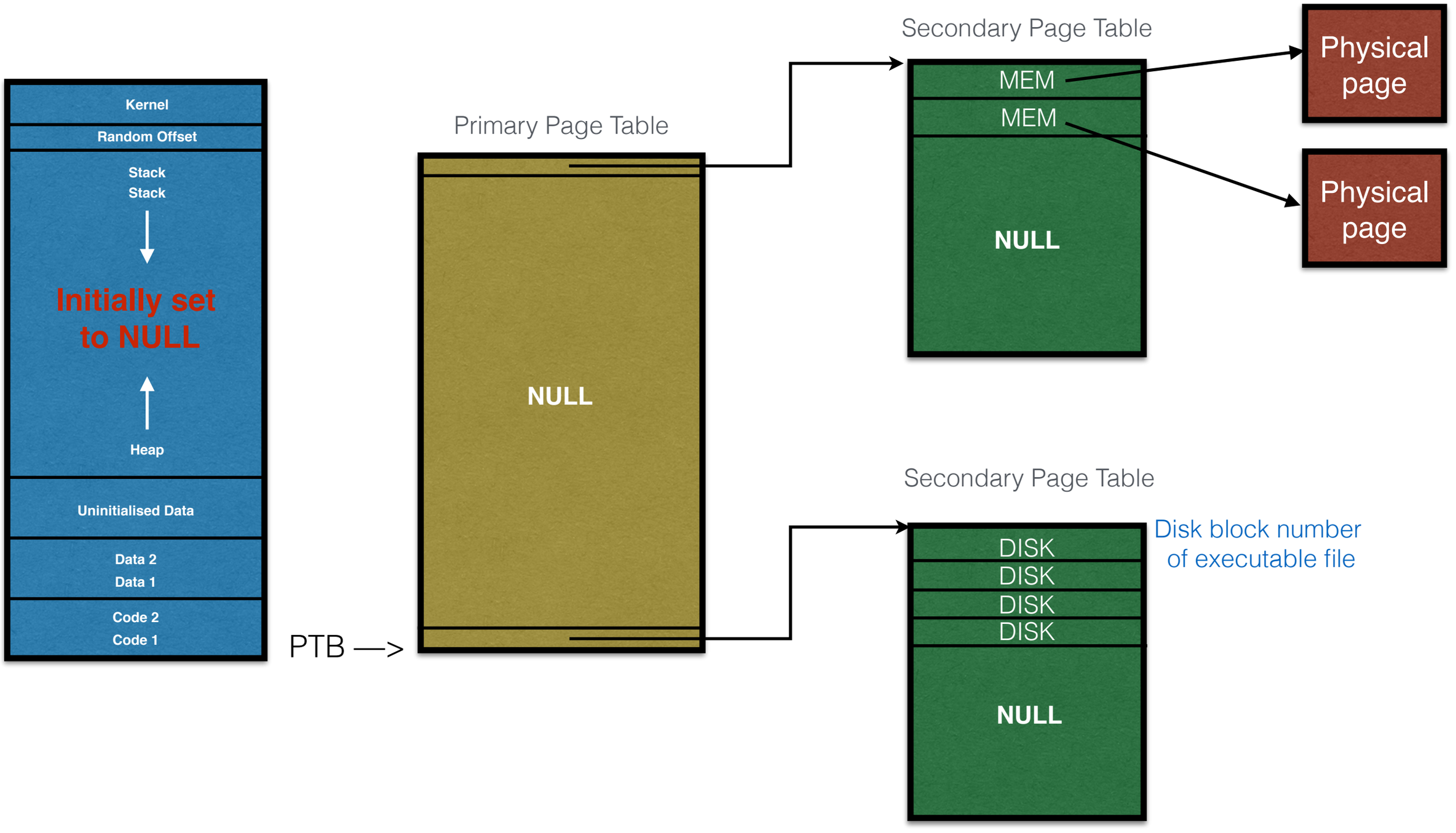
Initial Mapping of Processes

- Text and initialised data PTEs are initially set to type DISK [with specified disk block number].
- A Number of stack pages is allocated so that parameters passed to the process and environmental data can be stored.
- All the remaining PTEs are set to type NULL.
- All further pages are allocated only on demand.

Initial Mapping of Processes



Initial Mapping of Processes



Only 5 physical pages initially allocated by the process!

Initial Mapping of Processes

- After the initial page table is created, the process starts execution.
- It will immediately generate a page fault because the page containing the first instruction is still on disk. This will be loaded and execution will proceed.
- Page faults will continue to happen as the process keeps on executing.
- Each page fault will be handled differently by the MMU/OS depending on the type of page fault.

Page Faults on Process Execution

DISK → Allocate a physical page [either by using an empty one or by evicting unused pages from memory] and fill with data from the desired disk block [context switch performed while data is being read]. Operation:

DISK —> IOP —> MEM

NULL → Physical memory not yet allocated for the page. If page fault virtual address is not in uninitialised data, or heap, or stack then it is considered memory violation. Otherwise a page is allocated by OS.

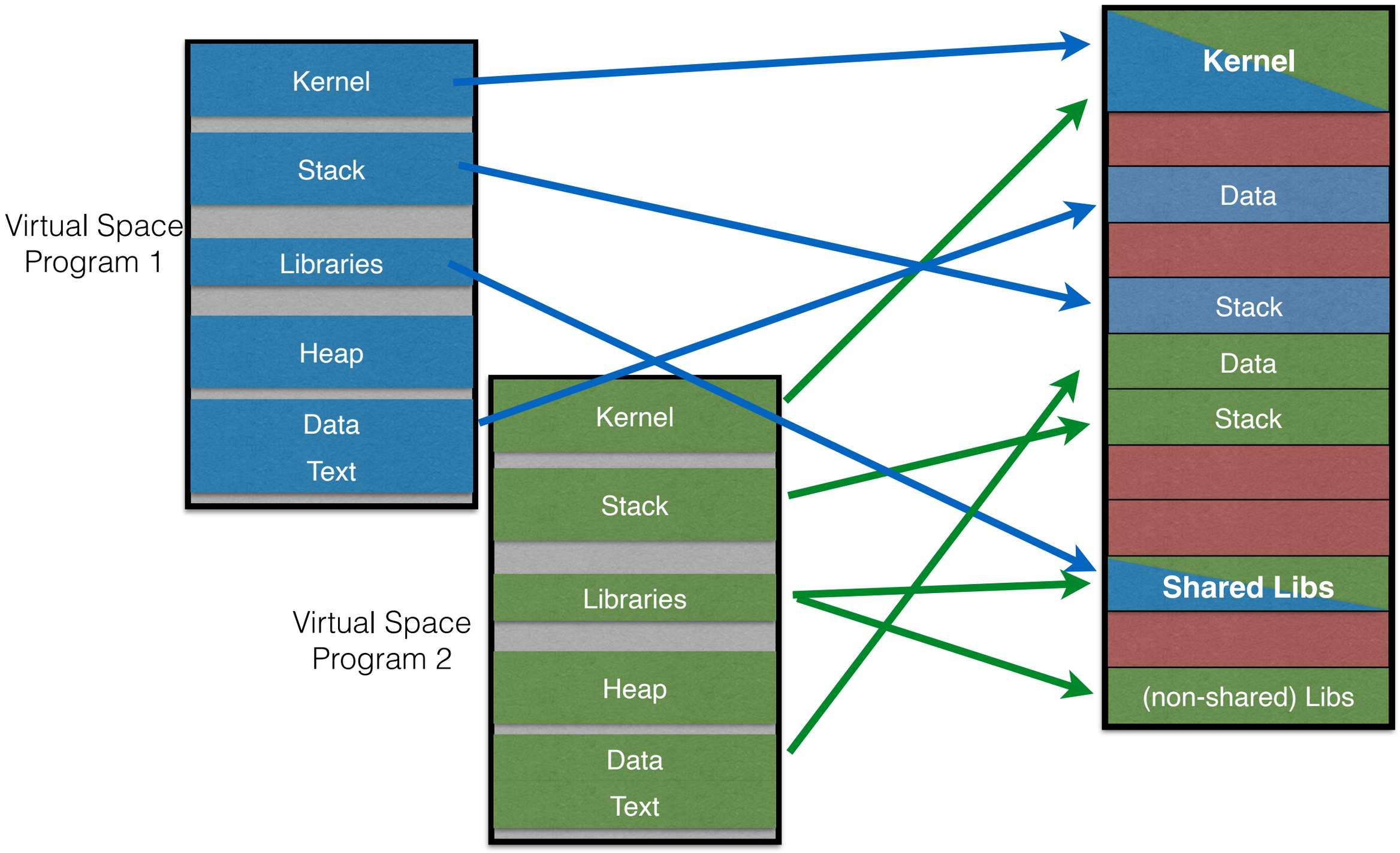
NULL —> MEM

IOP → Waiting for DISK read/write to complete

MEM, LOCK, SPY → Protection level fault [e.g. writing to text via NULL pointer]

If OS cannot resolve page fault it will call panic() and rebooting happens

Sharing/Protection Across Processes



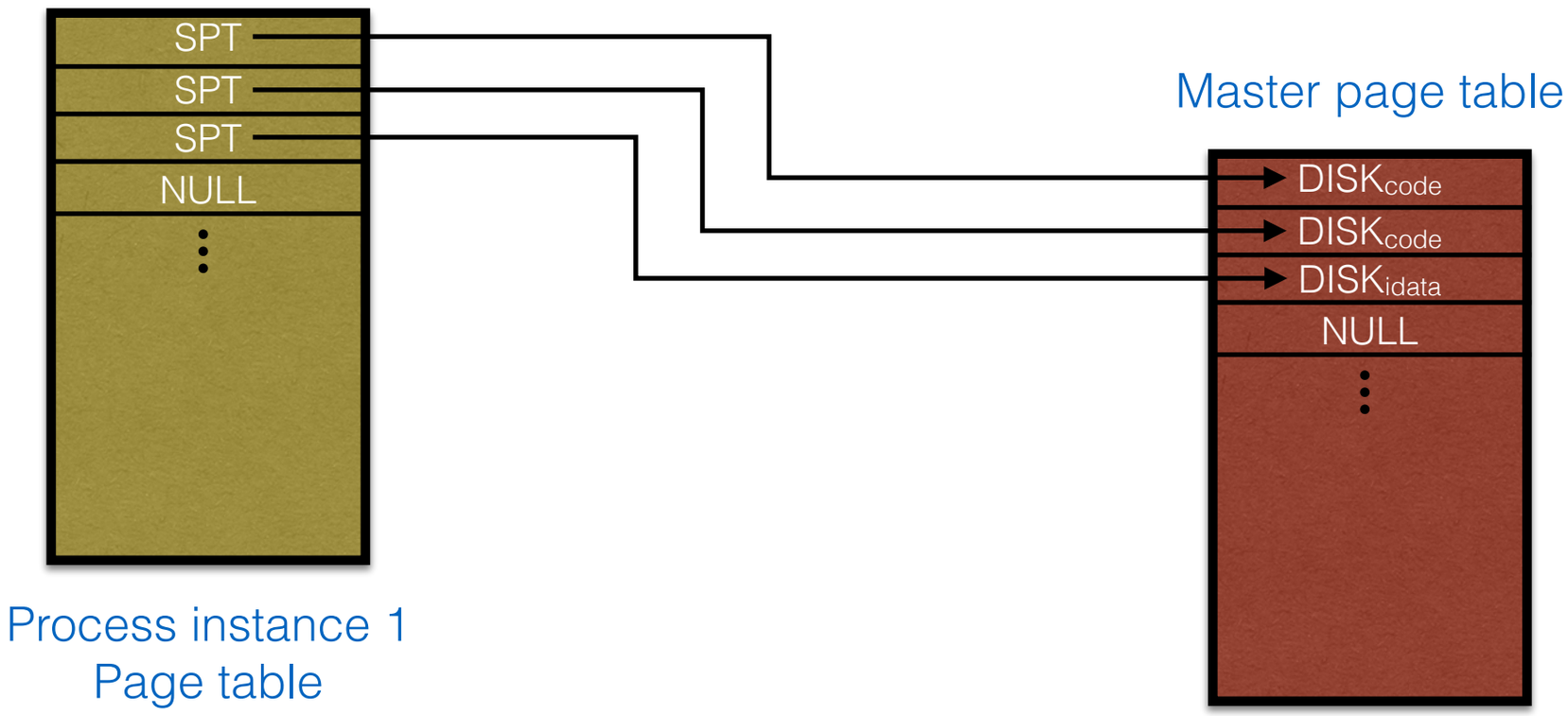
Text Sharing on multiple execution

- If the same process is executed more than once, then you need only one copy of the code to be stored in memory.
- Also initialised data can be shared if read-only. However each process call still needs its own virtual space for data, stack and heap.
- One way of doing so is to create a master page table when a process is first executed.
- The master table PTEs corresponding to code and initialised data are set to DISK. While all the remaining one are set to NULL.

Text Sharing

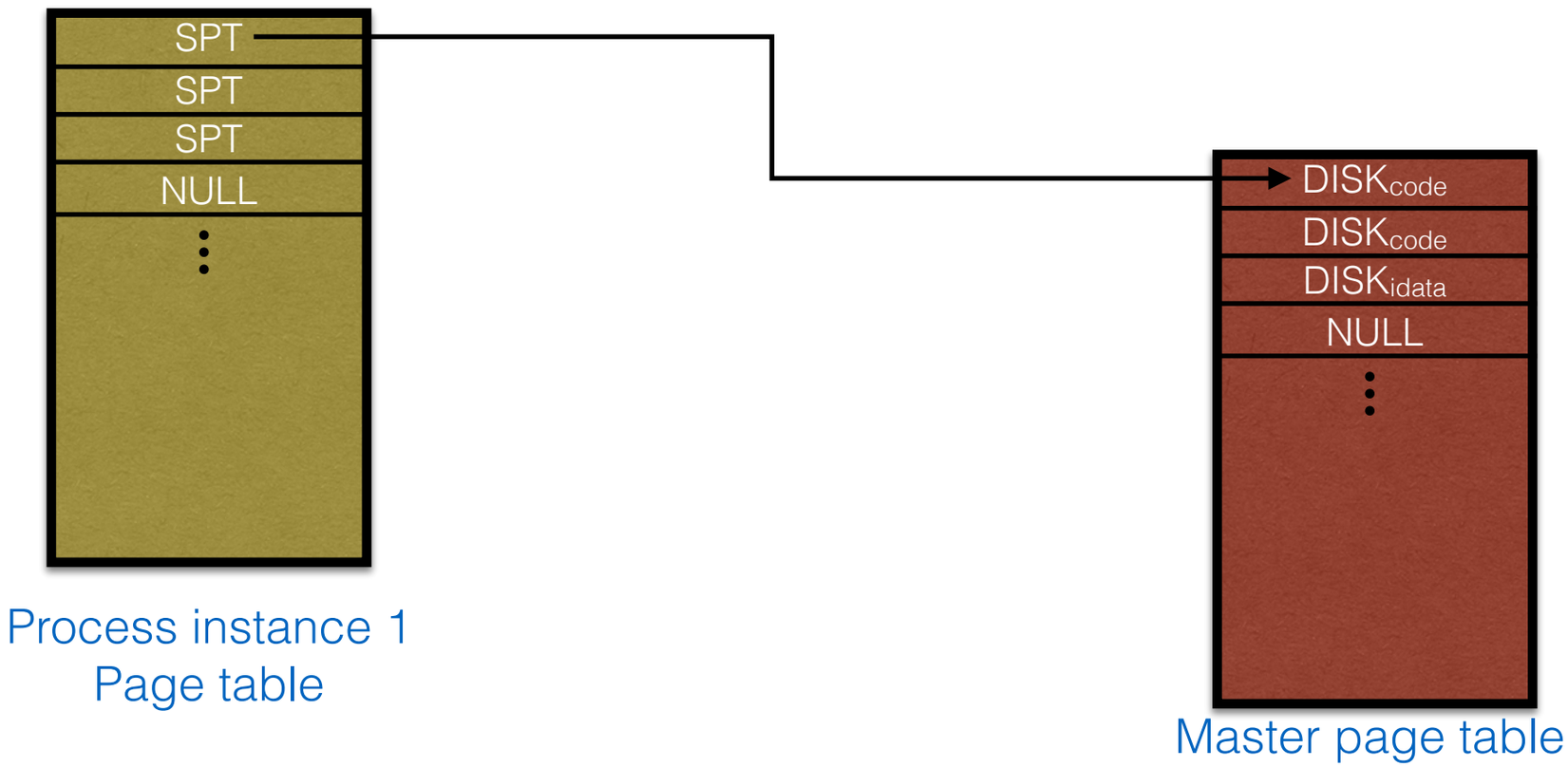
- The process page table is then initialised by setting code and initialised data to type SPT. SPT entries point to the corresponding entries in the master page table.

- The physical pages for the initial stack are then attached to the process page table, and all remaining PTEs are set to type NULL.



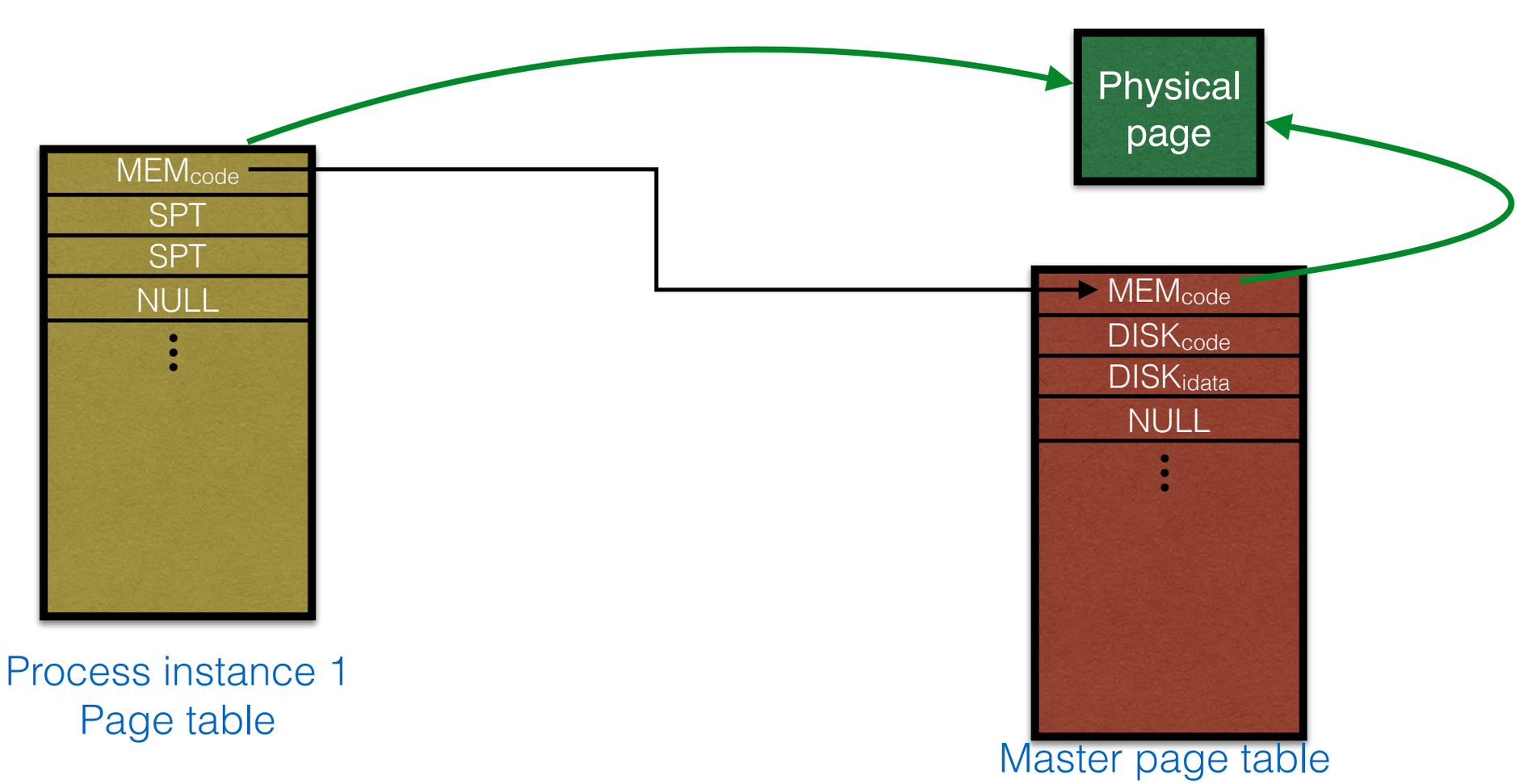
Text Sharing

- Upon encountering an SPT page fault, the kernel follows the SPT entry to the corresponding master page table entry.
- If $Disk_{code}$ then physical page memory is allocated, and then both the process page table and the master page table get updated.



Text Sharing

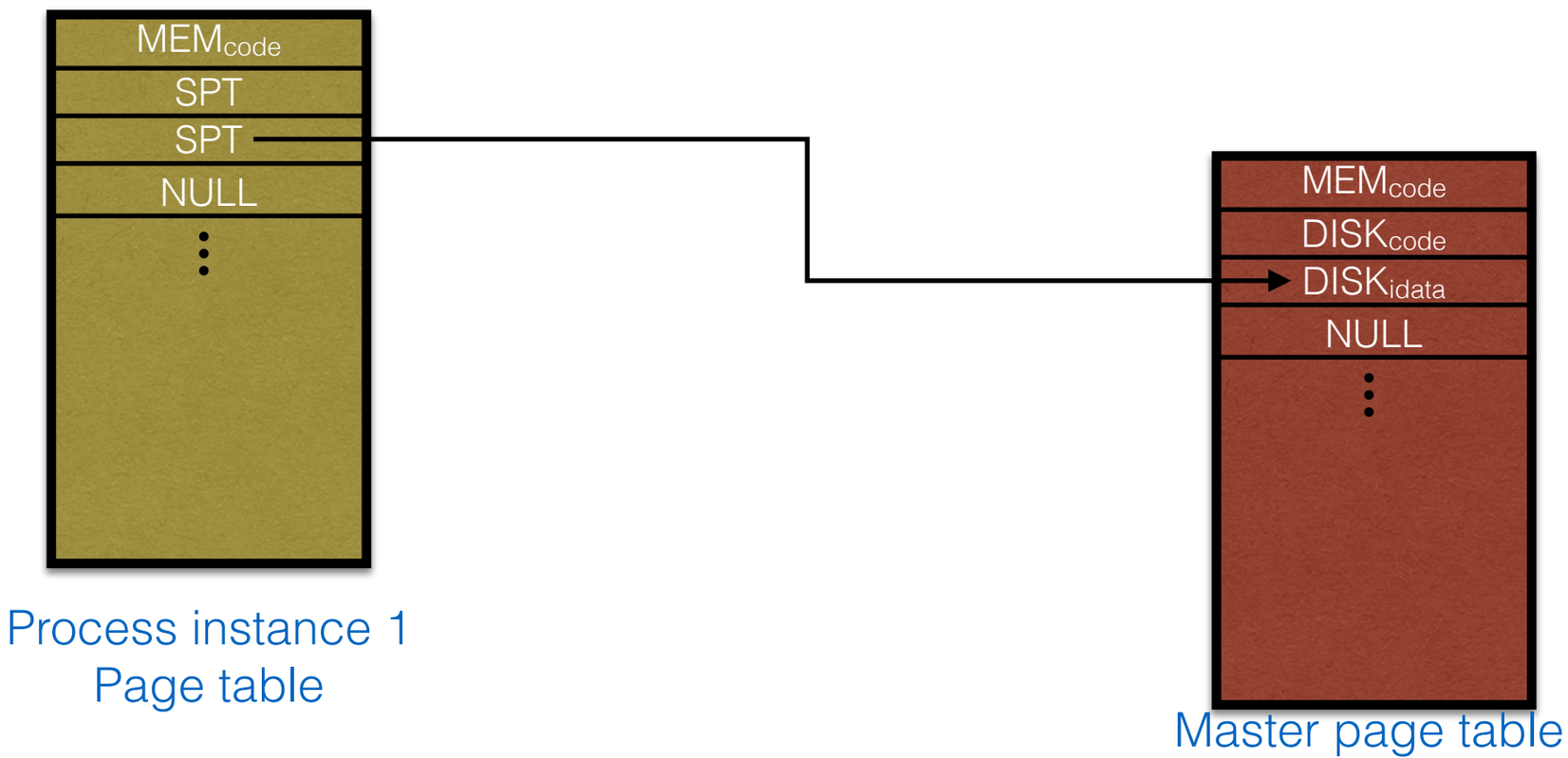
- Upon encountering an SPT page fault, the kernel follows the SPT entry to the corresponding master page table entry.
- If $Disk_{code}$ then physical page memory is allocated, and then both the process page table and the master page table get updated.



Text Sharing

If $Disk_{idata}$ then:

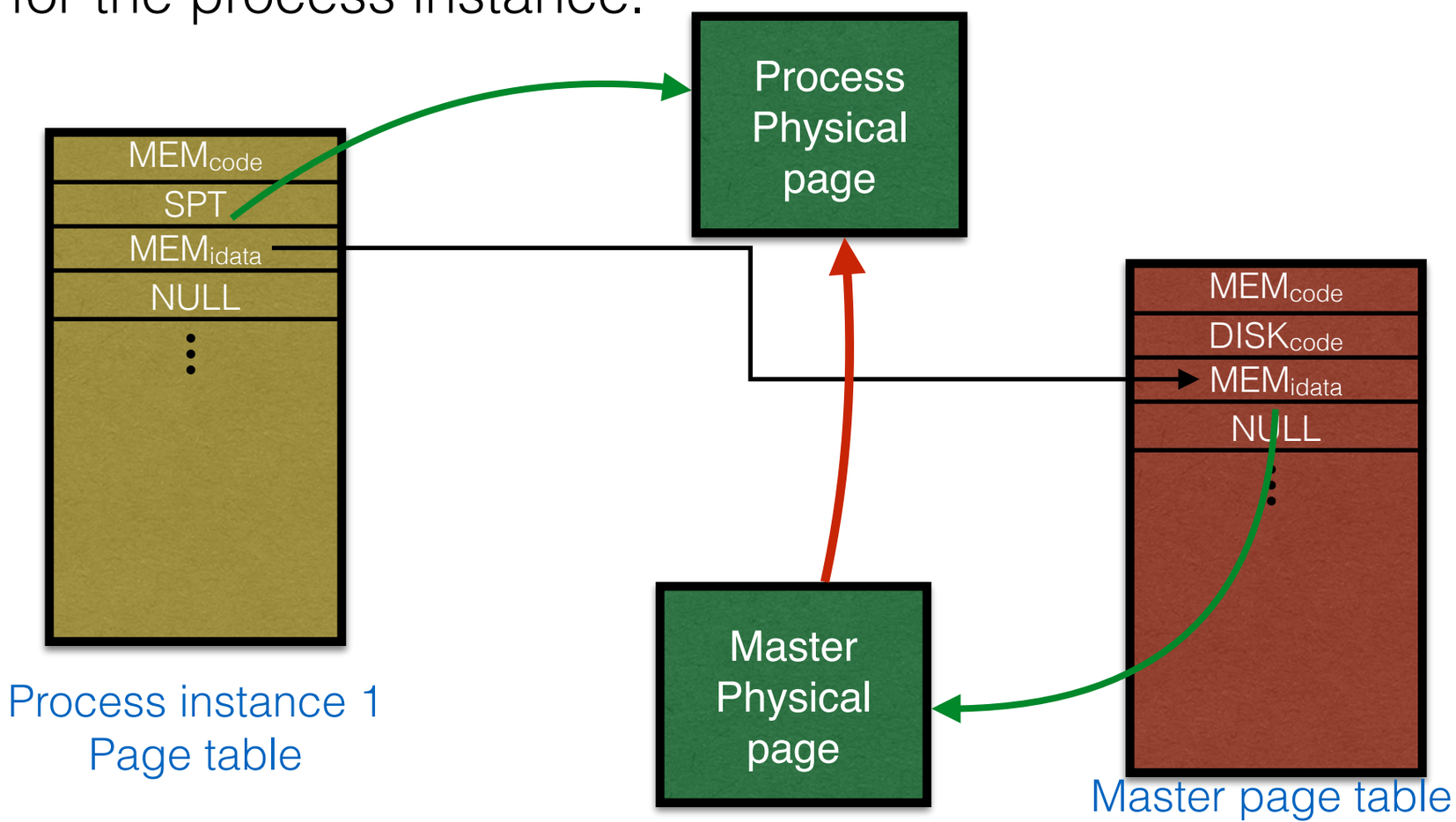
- Allocate page of physical memory for master page table, fill with disk data, and set to read only.
- Allocate second page of physical memory for the process page table, and copy data from the master page table physical page. This physical page can be read/write for the process instance.



Text Sharing

If $Disk_{idata}$ then:

- Allocate page of physical memory for master page table, fill with disk data, and set to read only.
- Allocate second page of physical memory for the process page table, and copy data from the master page table physical page. This physical page can be read/write for the process instance.



Text Sharing Second process

- Now if another instance of the same process is executed, the process can rely on the master page table already created for its code and initialised data.
- MEM_{code} entry are copied immediately, while remaining entries are set to SPT.
- Stack, heap and data are managed as normal.

MMU Summary

- We have discussed what virtual memory is, and what its advantages are.
- Modern features of MMUs implementation like TLB and N-level page tables.
- Initial mapping of virtual memory and working of abstracted MMU/OS interface upon process execution
- Text/data sharing and protection.