University of Pittsburgh

Introduction to Operating Systems
CS 1550

Spring 2023

Sherif Khattab

ksm73@pitt.edu

(Some slides are from **Silberschatz, Galvin and Gagne ©2013)**

Announcements

- Upcoming deadlines
  - Homework 9 due this Friday
  - Project 3 is due Friday 4/7 at 11:59 pm
  - Lab 4 is due on Tuesday 4/11 at 11:59 pm

Previous Lecture

- How to simulate page replacement algorithms
  - Clock, LRU, OPT

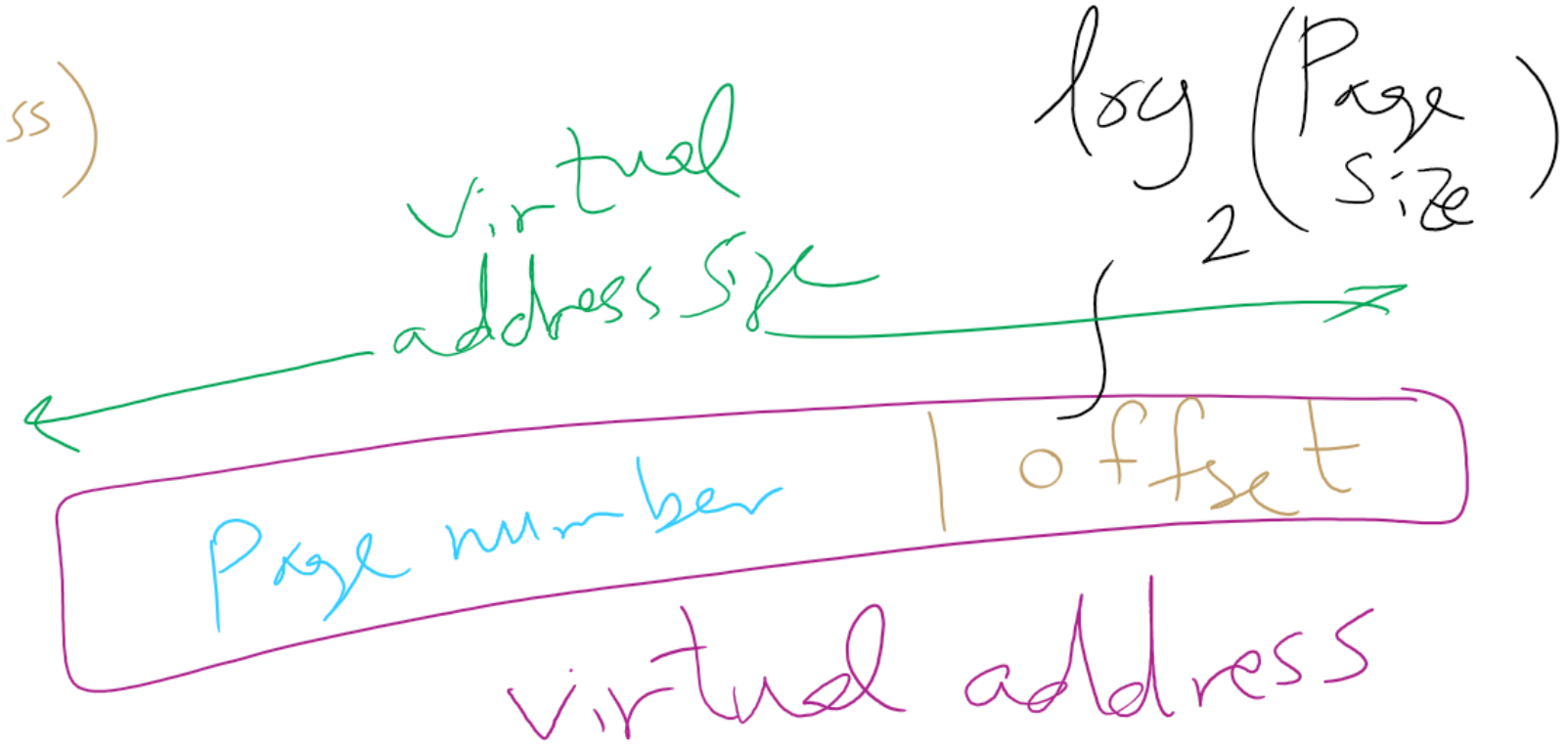
This Lecture …

- Problem of Too Large Page Tables
  - multi-level page tables + TLB
  - inverted page tables
- Miscellaneous Memory Management Issues

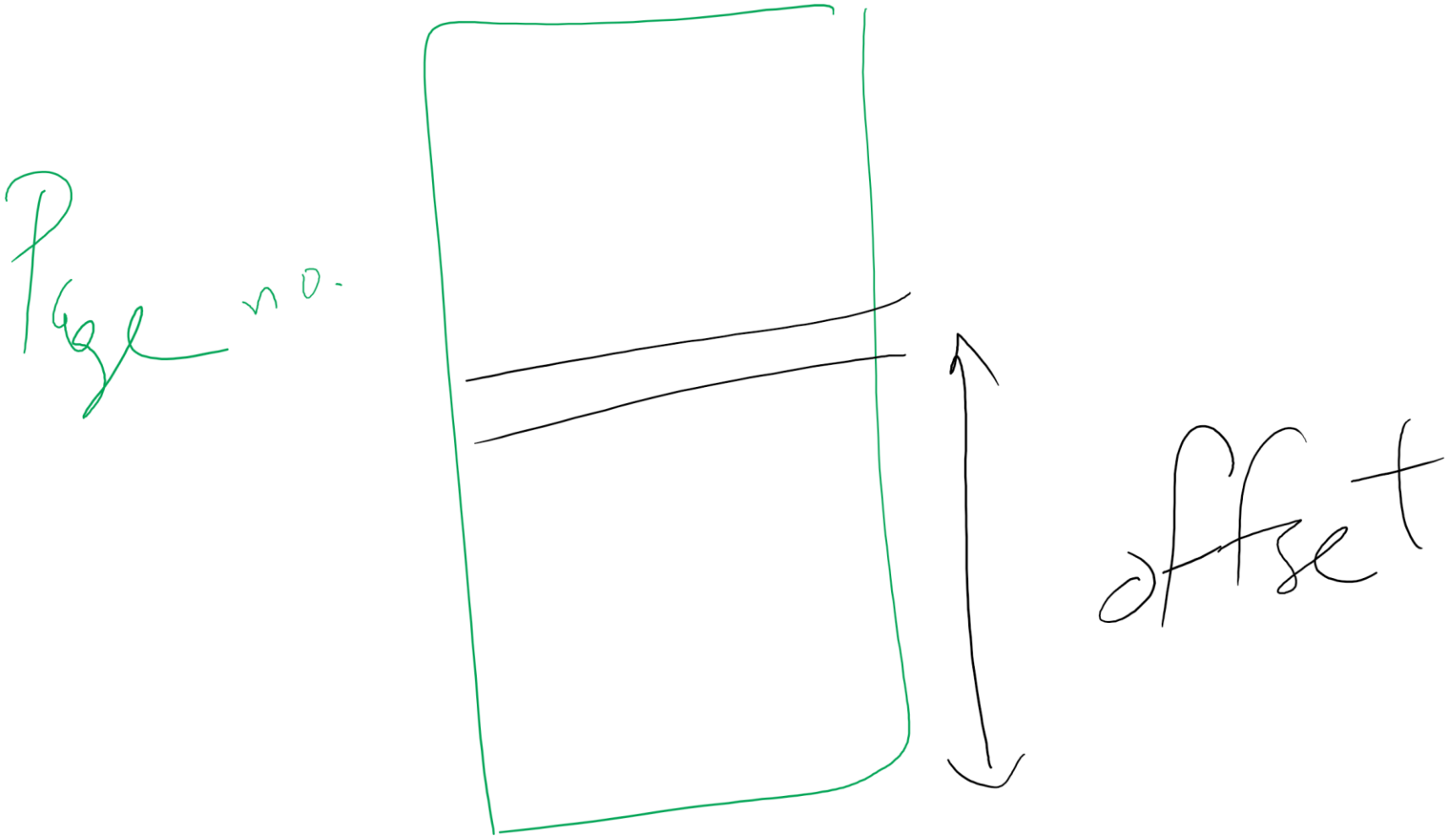
Page Table Size

- How big can a page table be?
  - 64-bit machine
  - 4 KB page size
  - How many pages?
  - How many PTEs?
  - How big is a PTE?
  - How big is the page table of one process?

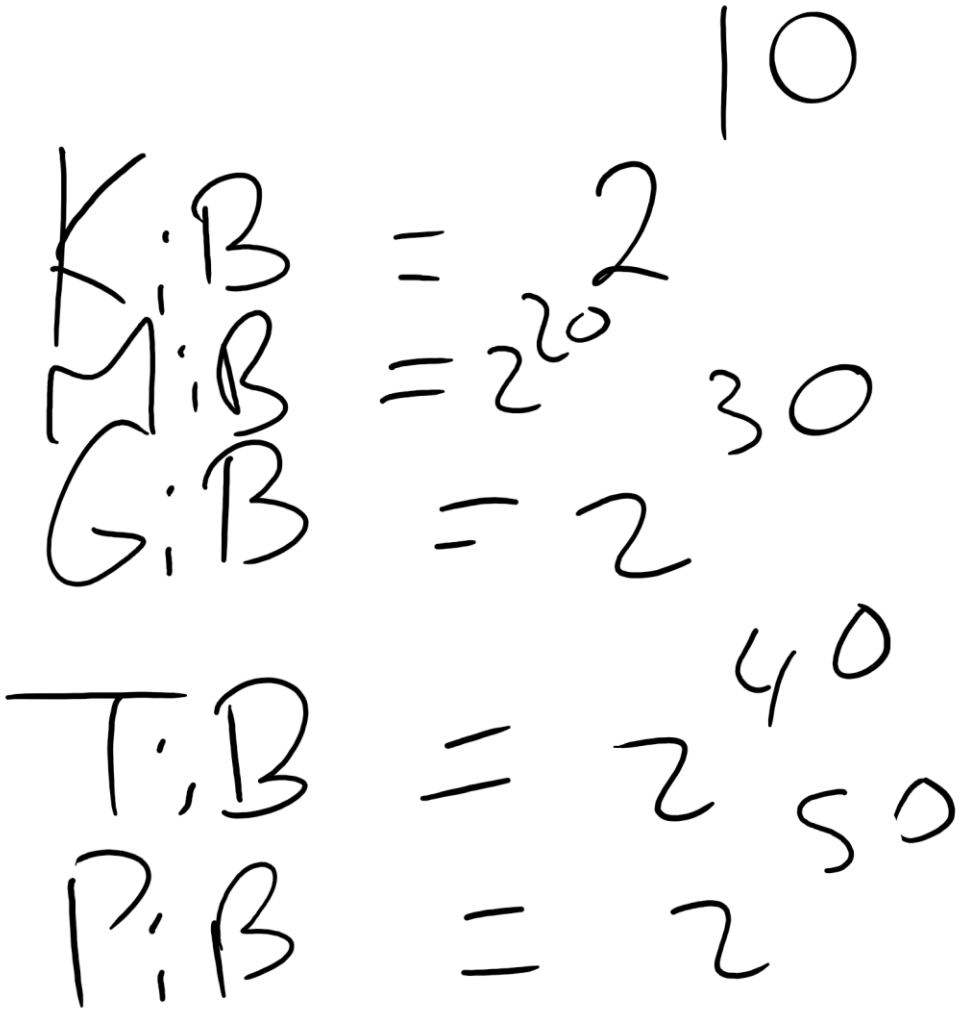
Virtual Address



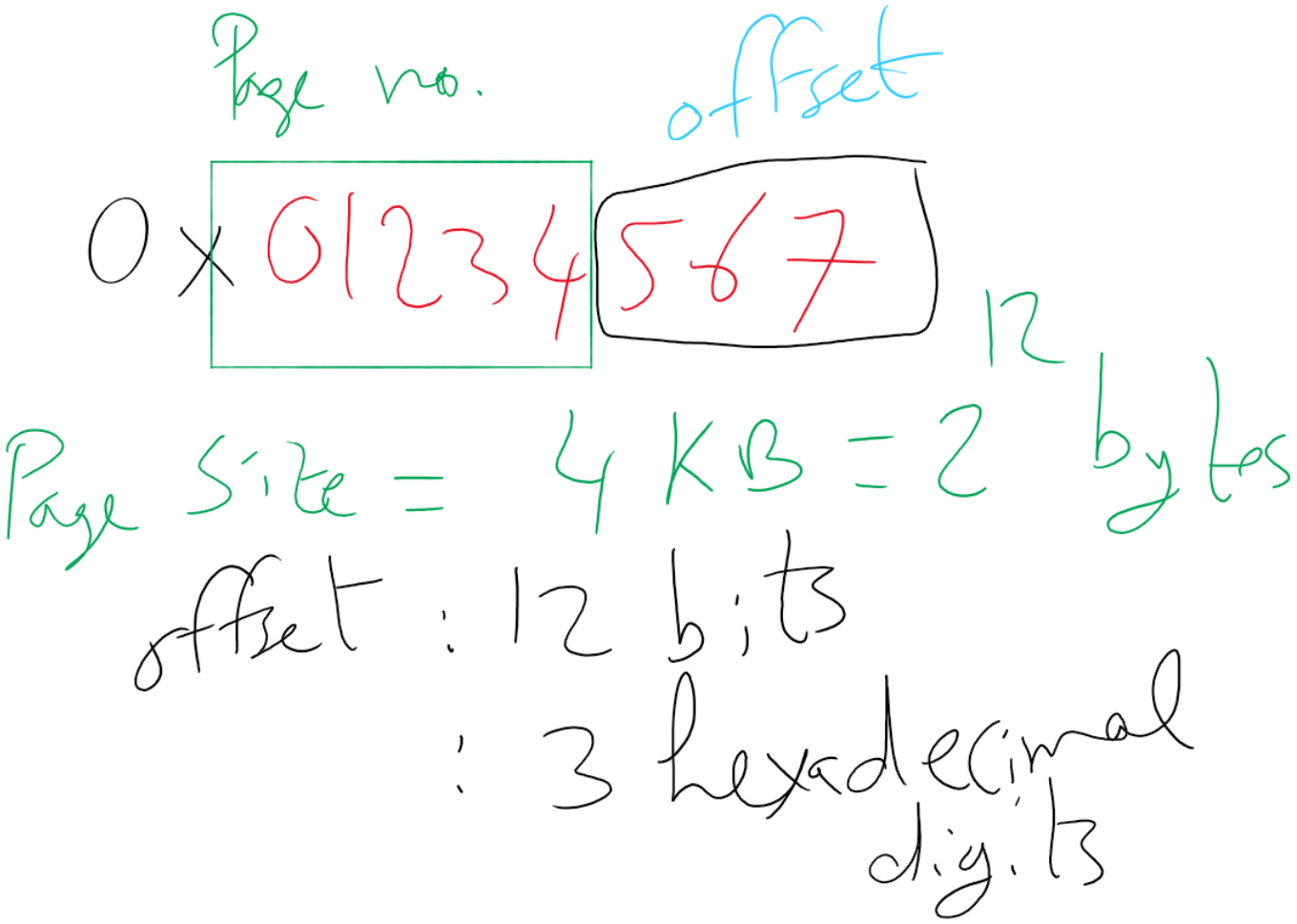
Page number vs. Offset



Memory Sizes



Example on splitting a virtual address



PTEs

$$\#PTEs = \frac{\text{Virtual address space size}}{\text{Page size}} = \frac{2^{(\text{logical address size})}}{\text{Page size}}$$

$$= 2^{(\text{Page no. size})}$$

$\lceil$

PTE Size

$$\text{frame number size} = \left\lceil \log_2 \#\text{frames} \right\rceil$$

$$\#\text{frames} = \frac{\text{RAM size}}{\text{frame size}}$$

Page Table Size

Page Table Size = # PTEs * PTE size

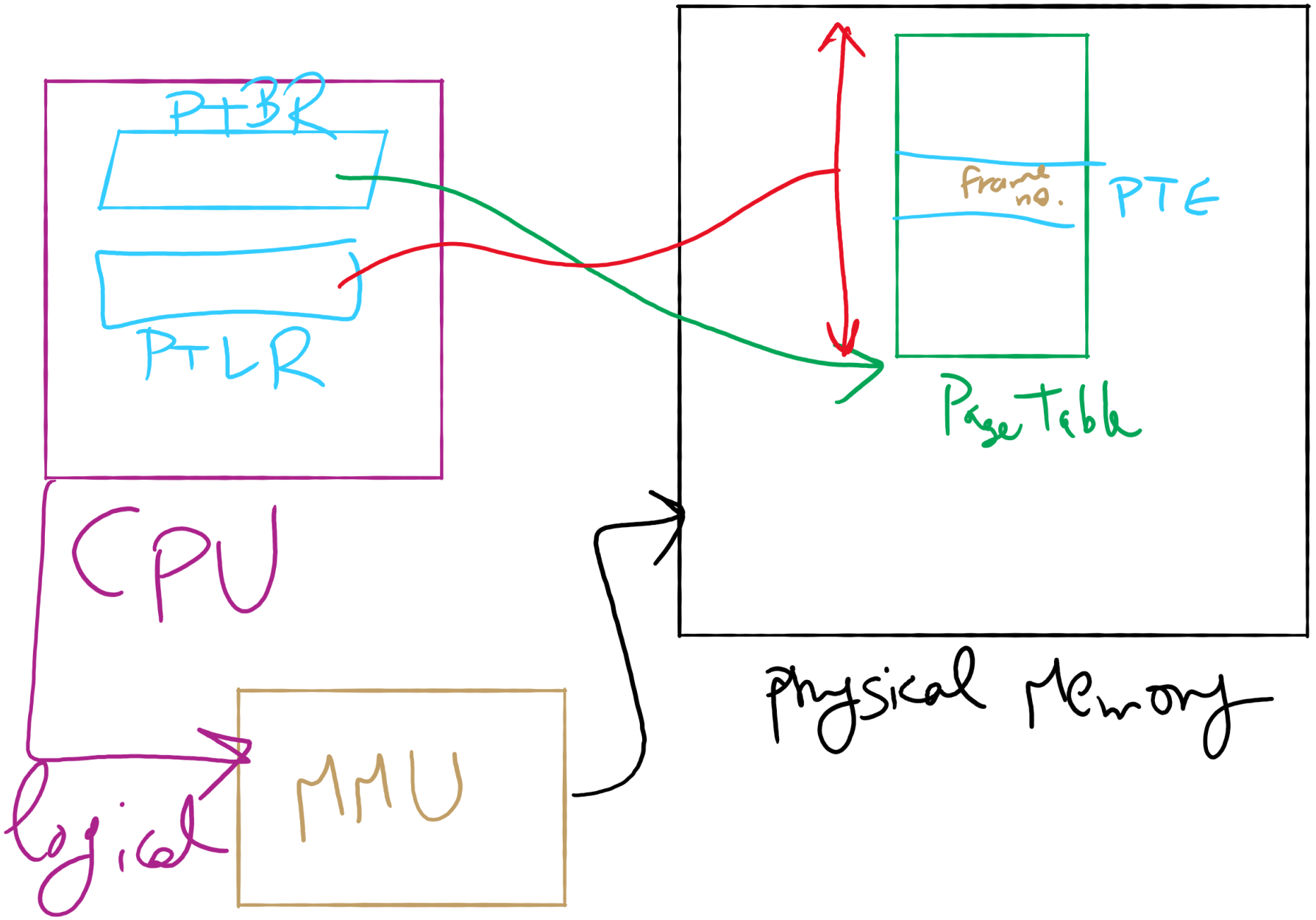
Page Table Size

- How big can a page table be?
  - 64-bit machine → virtual address size = 64 bits
  - 4 KB page size →page offset is $\log_2 2^{12} = 12$ bits
  - How many pages?
  - How many PTEs?
  - #PTEs = 2^{64} / page size = 2^{64} / 2^{12} = 2^{52}
  - How big is a PTE?
  - Assume 4 GB RAM (Physical Memory) = 2^{32} bytes
  - # Frames = 2^{32} / page size = 2^{32} / 2^{12} = 2^{20}
  - Frame no. size = $\log_2 2^{20} = 20$ bits
  - PTE size = 20 + (let's say) 12 bits for flags = 32 bits = 4 bytes
  - How big is the page table of **one process**?
  - #PTEs * PTE size = 2^{52} * 4 = 2^{54} bytes = **16 PB**

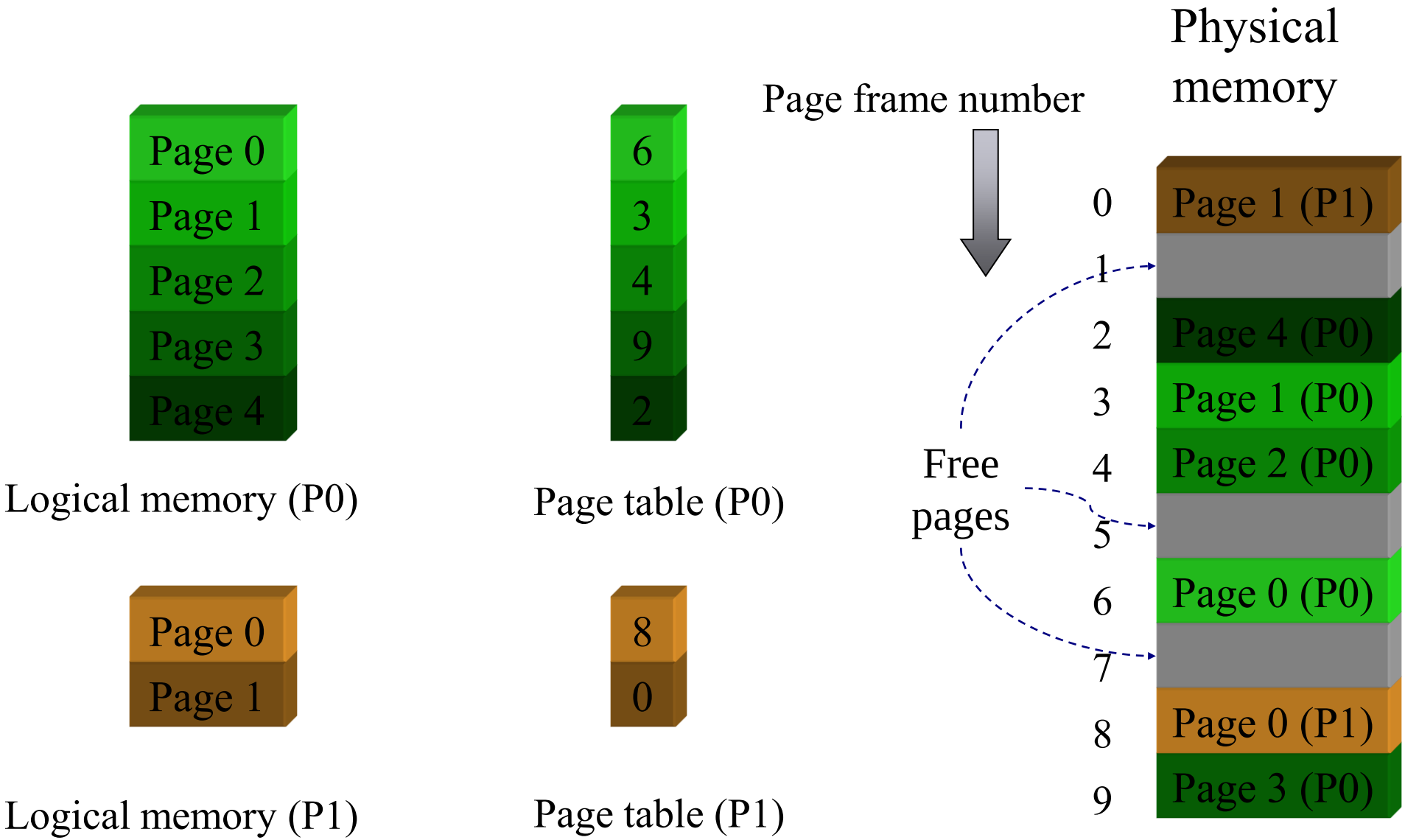
Problem of the Day

- How big is the page table of **one process**?
- #PTEs * PTE size = $2^{52} * 4 = 2^{54}$ bytes = **16 PB**
- Such page table is too big to put in memory!
- But why does it have to be in memory?
- Let's see how address translation happens …

Address Translation Structures



Memory & paging structures



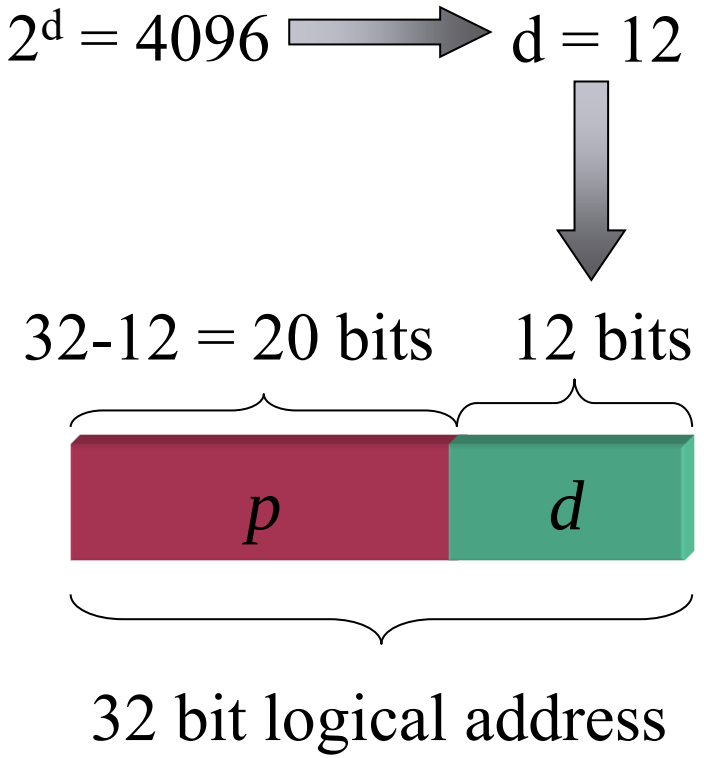
Mapping logical => physical address

- Split address from CPU into two pieces
  - Page number (*p*)
  - Page offset (*d*)
- Page number
  - Index into page table
  - Page table contains base address of page in physical memory
- Page offset
  - Added to base address to get actual physical memory address
- Page size = 2^d bytes

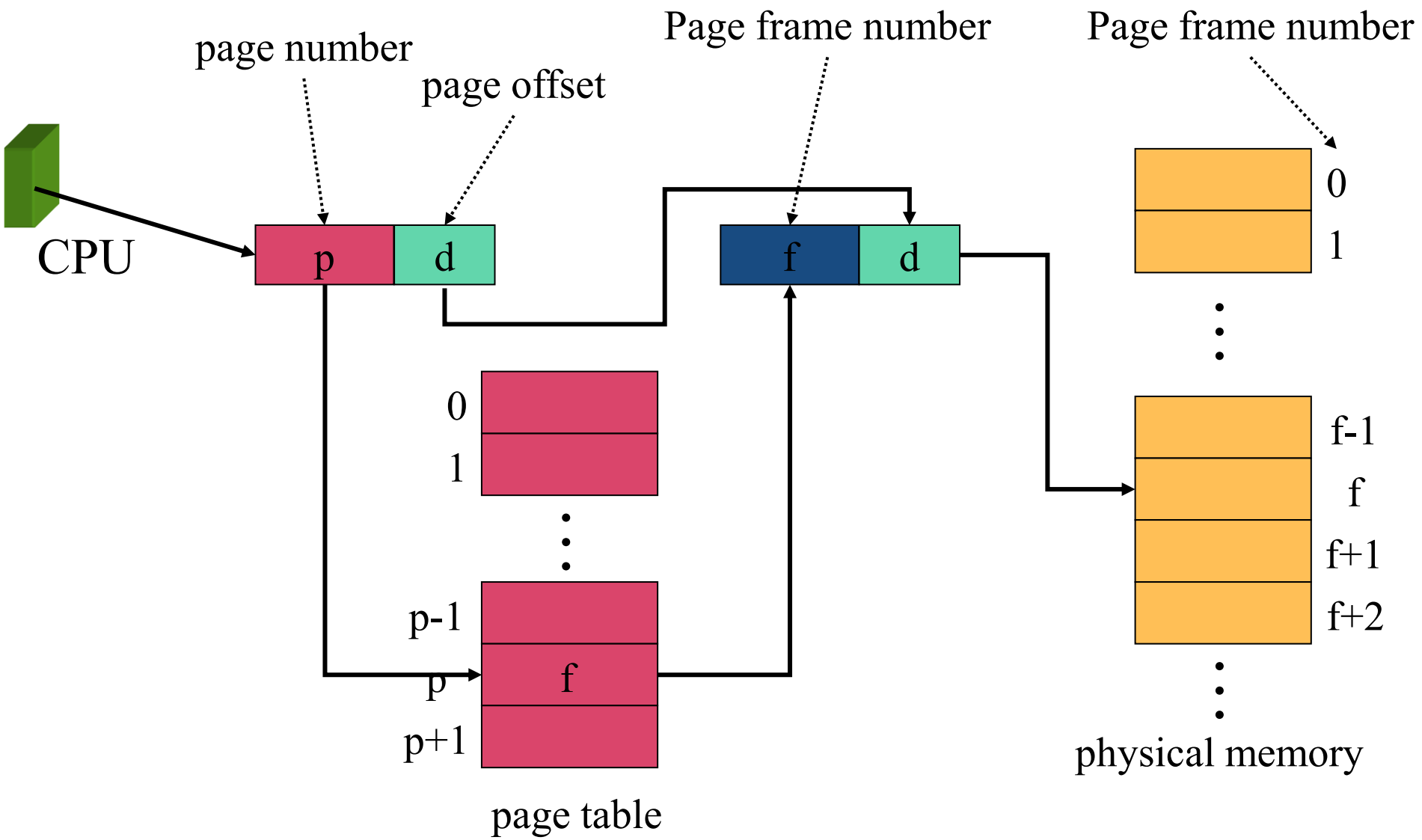
Example:

• 4 KB (=4096 byte) pages

• 32 bit logical addresses

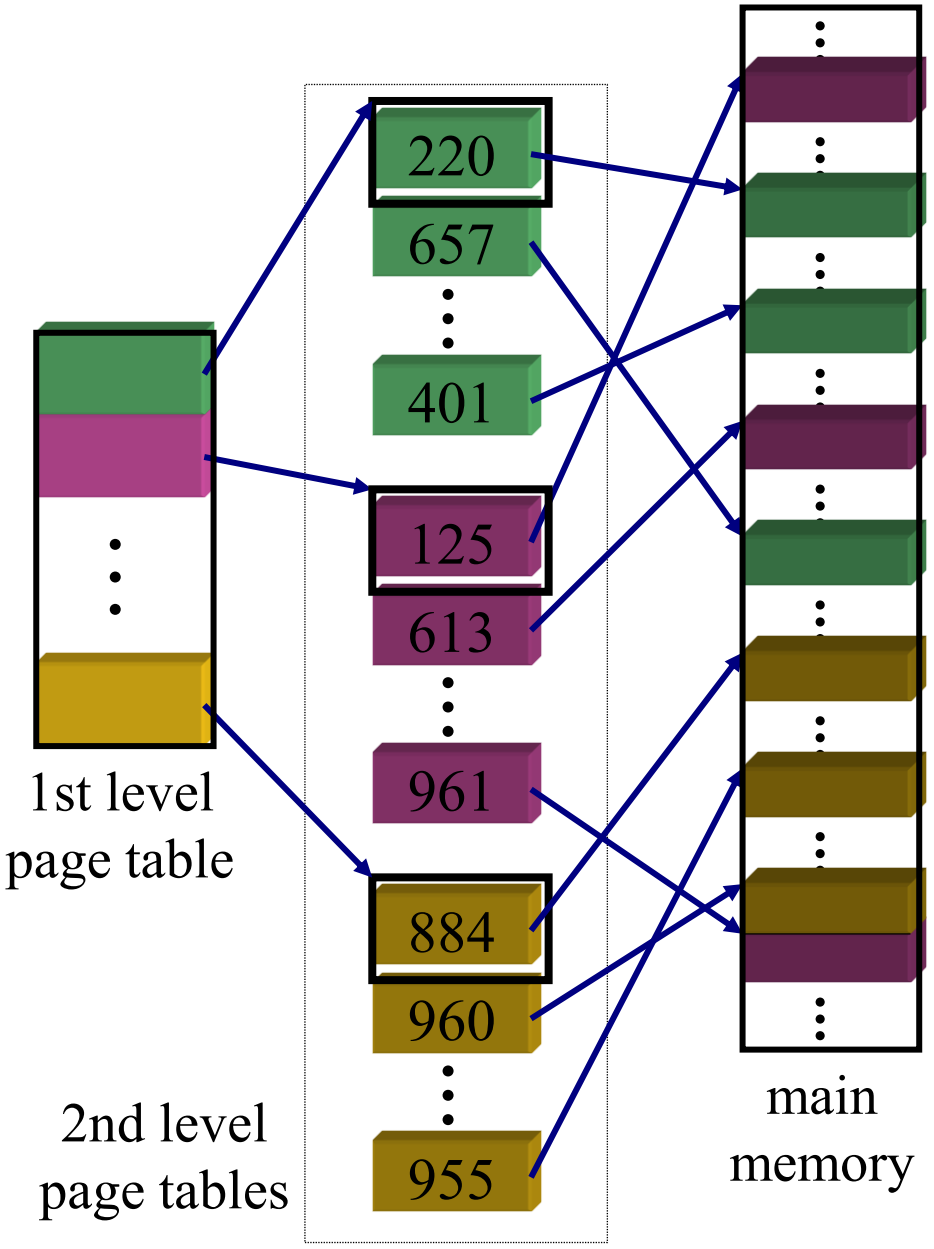


Address translation architecture



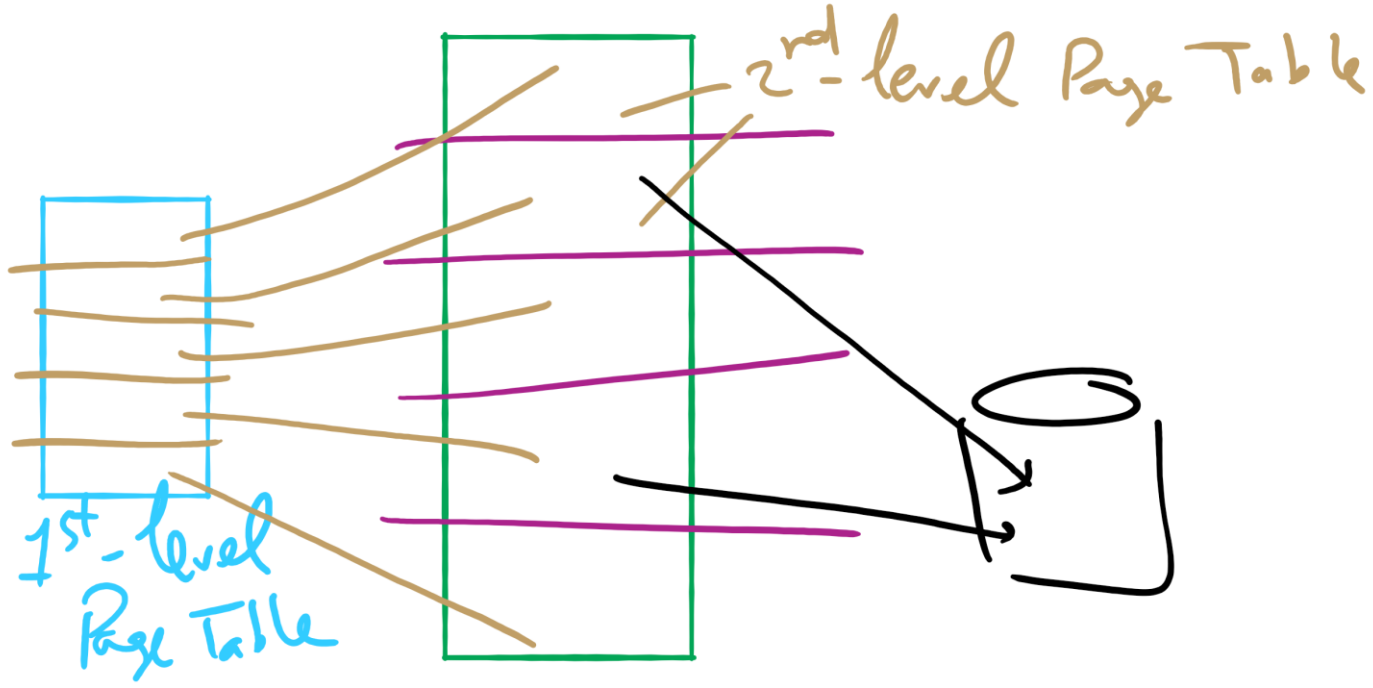
Solution 1: Two-level page tables

- 1st level page table called **page directory**
  - PDEs have pointers to 2nd level page tables
- PTEs in 2nd level page table have actual physical page numbers
- All addresses are physical
- Protection bits kept in 2nd level



How does that solve the problem?

- 2nd level page tables don't have to be in memory all at once
  - can be loaded **on demand**
  - PDEs marked invalid in first page table

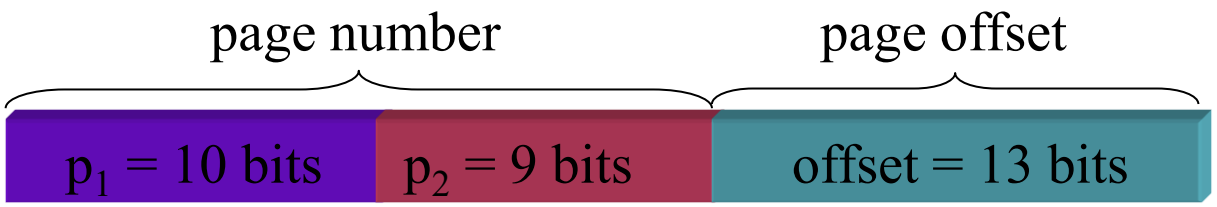


More on two-level page tables

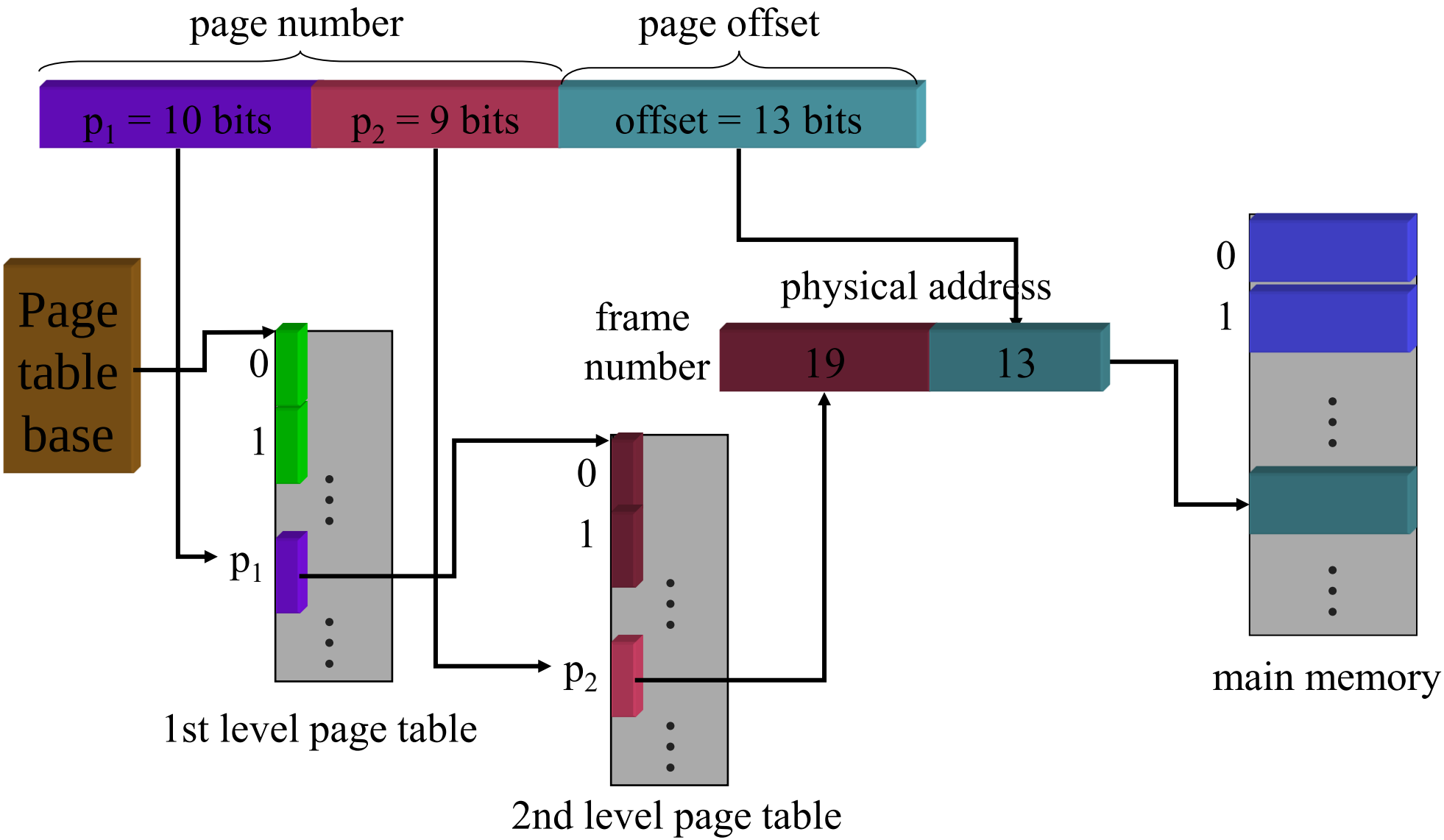
- Total number of bits indexing 1st and 2nd level table is constant for a given page size and logical address length. **Why?**
- More bits in 1st level: fine granularity at 2nd level
- Fewer bits in 1st level: maybe less wasted space?

Two-level paging: address translation

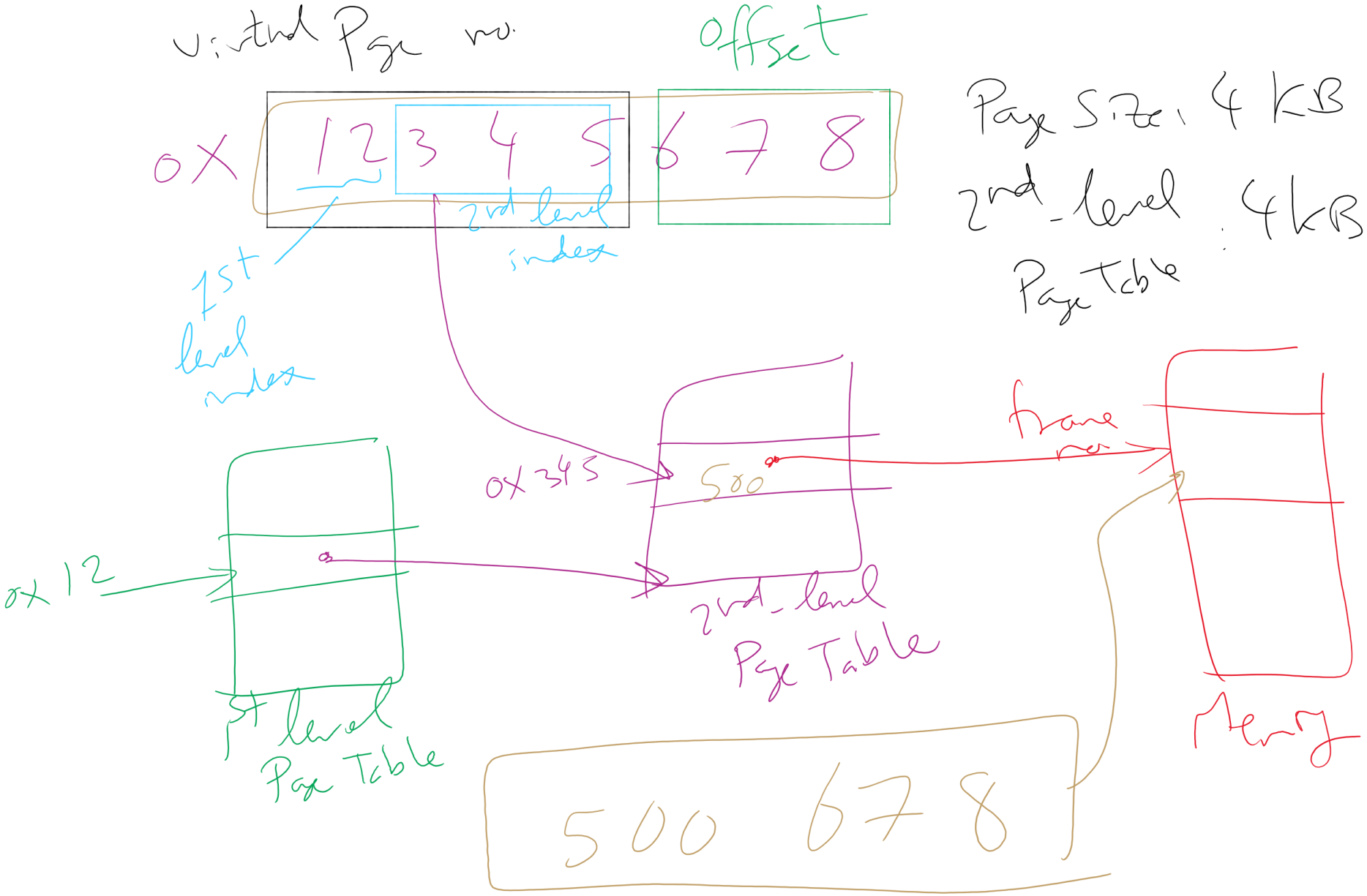
- 8 KB pages
- 32-bit logical address
- p_1 is an index into the 1st level page table
- p_2 is an index into the 2nd level page table pointed to by p_1



2-level address translation example

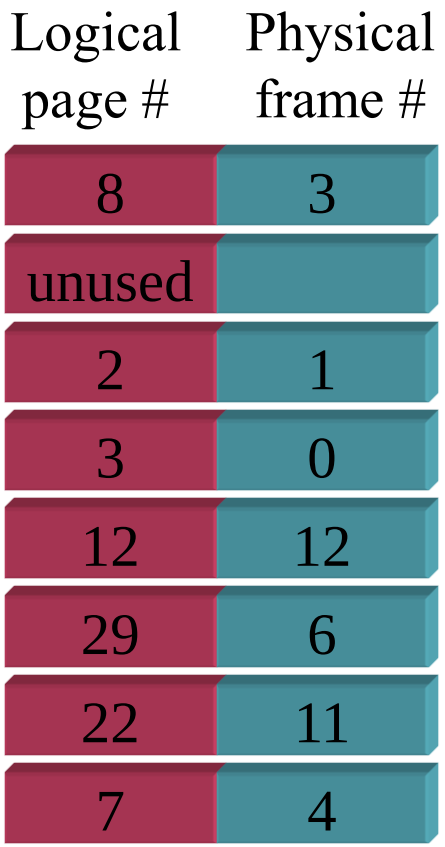


Address Translation: 2-level Page Table



Translation Lookaside Buffer (TLB)

- Search the TLB for the desired logical page number
  - Search entries in parallel
  - Use standard cache techniques
- If desired logical page number is found, get frame number from TLB
- If desired logical page number isn't found
  - Get frame number from page table in memory
  - Replace an entry in the TLB with the logical & physical page numbers from this reference

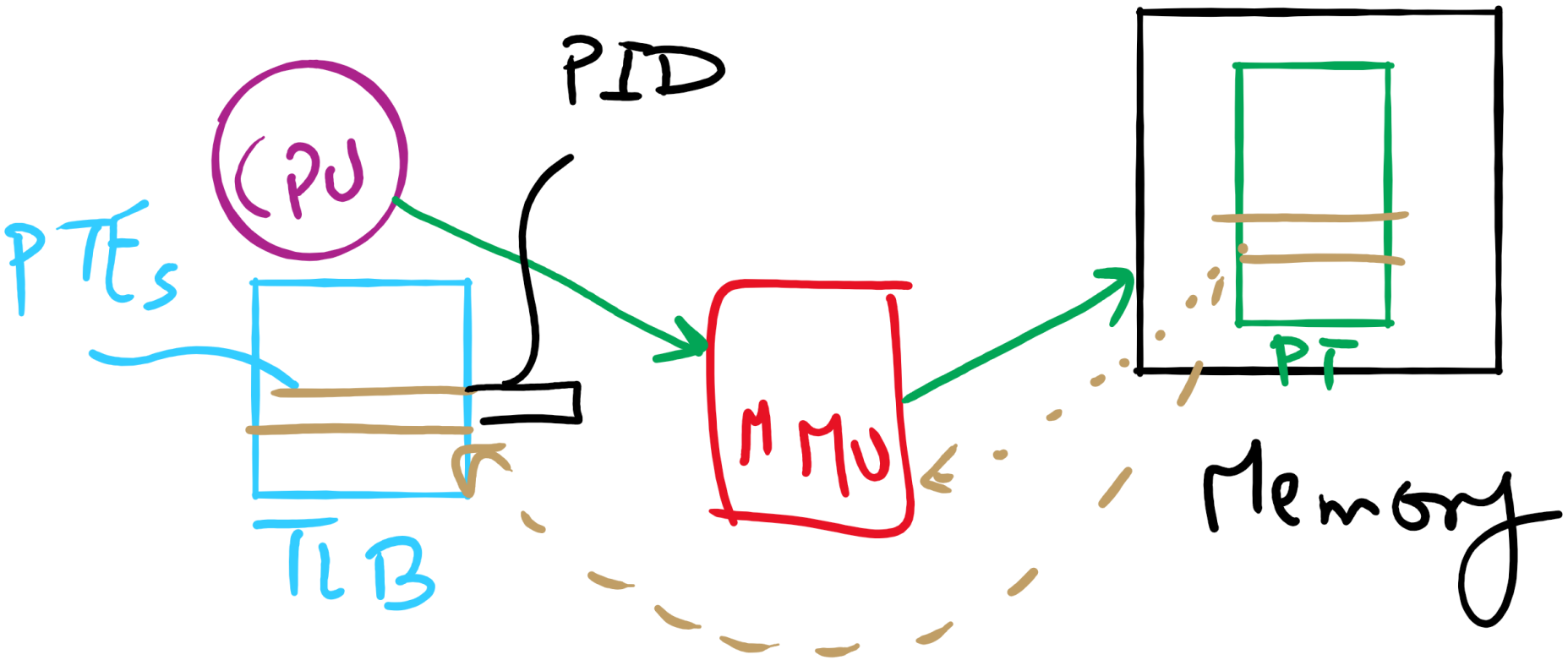


Example TLB

Handling TLB misses

- If PTE isn't found in TLB, OS needs to do the lookup in the page table
- Lookup can be done in hardware or software
- Hardware TLB replacement
  - CPU hardware does page table lookup
  - Can be faster than software
  - Less flexible than software, and more complex hardware
- Software TLB replacement
  - OS gets TLB exception
  - Exception handler does page table lookup & places the result into the TLB
  - Program continues after return from exception
  - Larger TLB (lower miss rate) can make this feasible

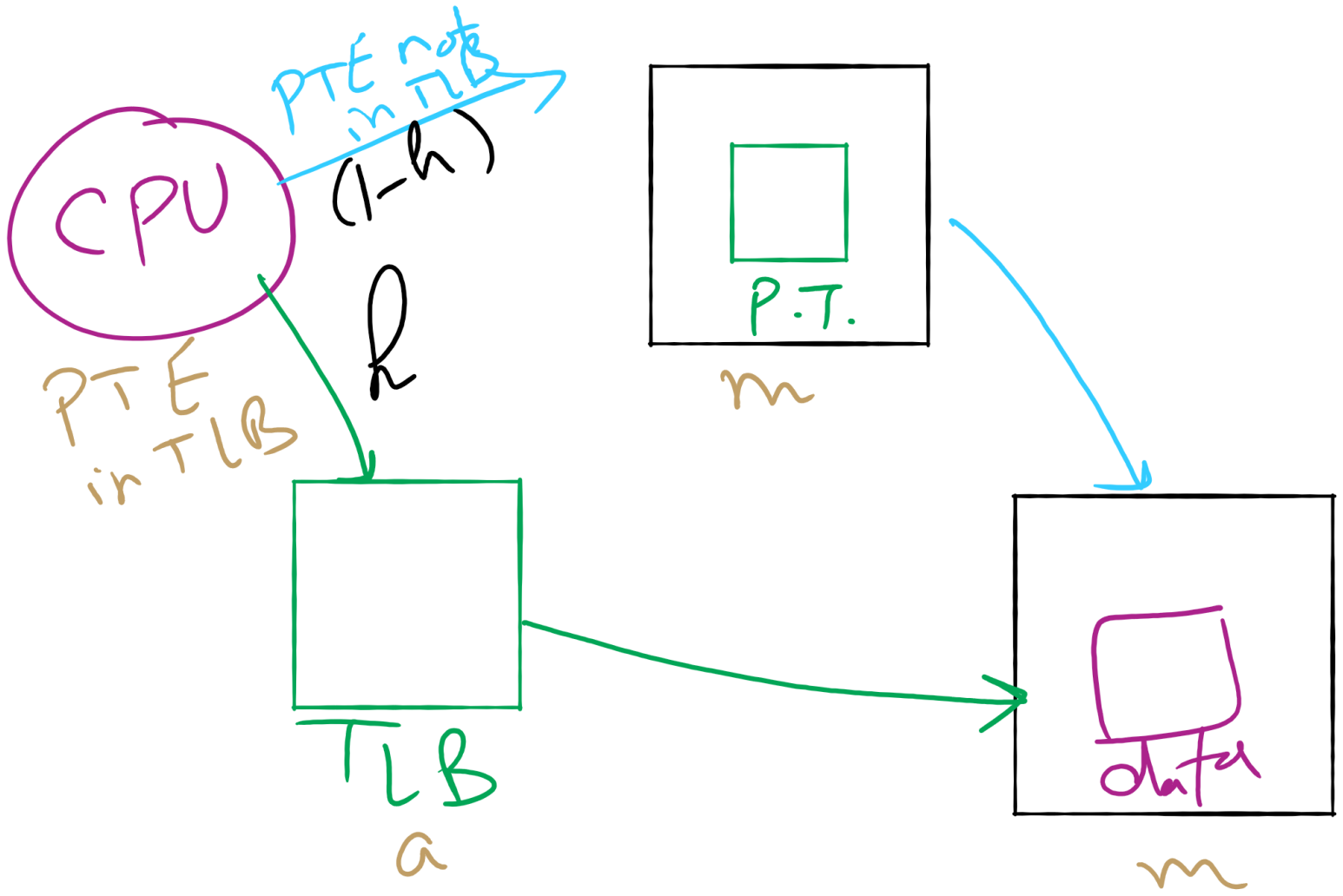
TLB



How long do memory accesses take?

- Assume the following times:
  - TLB lookup time = a (often zero - overlapped in CPU)
  - Memory access time = m
- Hit ratio (h) is percentage of time that a logical page number is found in the TLB
  - Larger TLB usually means higher h
  - TLB structure can affect h as well
- Effective access time (an average) is calculated as:
  - EAT = (m + a)h + (m + m + a)(1-h)
  - EAT =a + (2-h)m
- Interpretation
  - Reference always requires TLB lookup, 1 memory access
  - TLB misses also require an additional memory reference

Effective Access Time



$$EAT = h(a+m) + (1-h)(a+m+m)$$

EAT Calculation

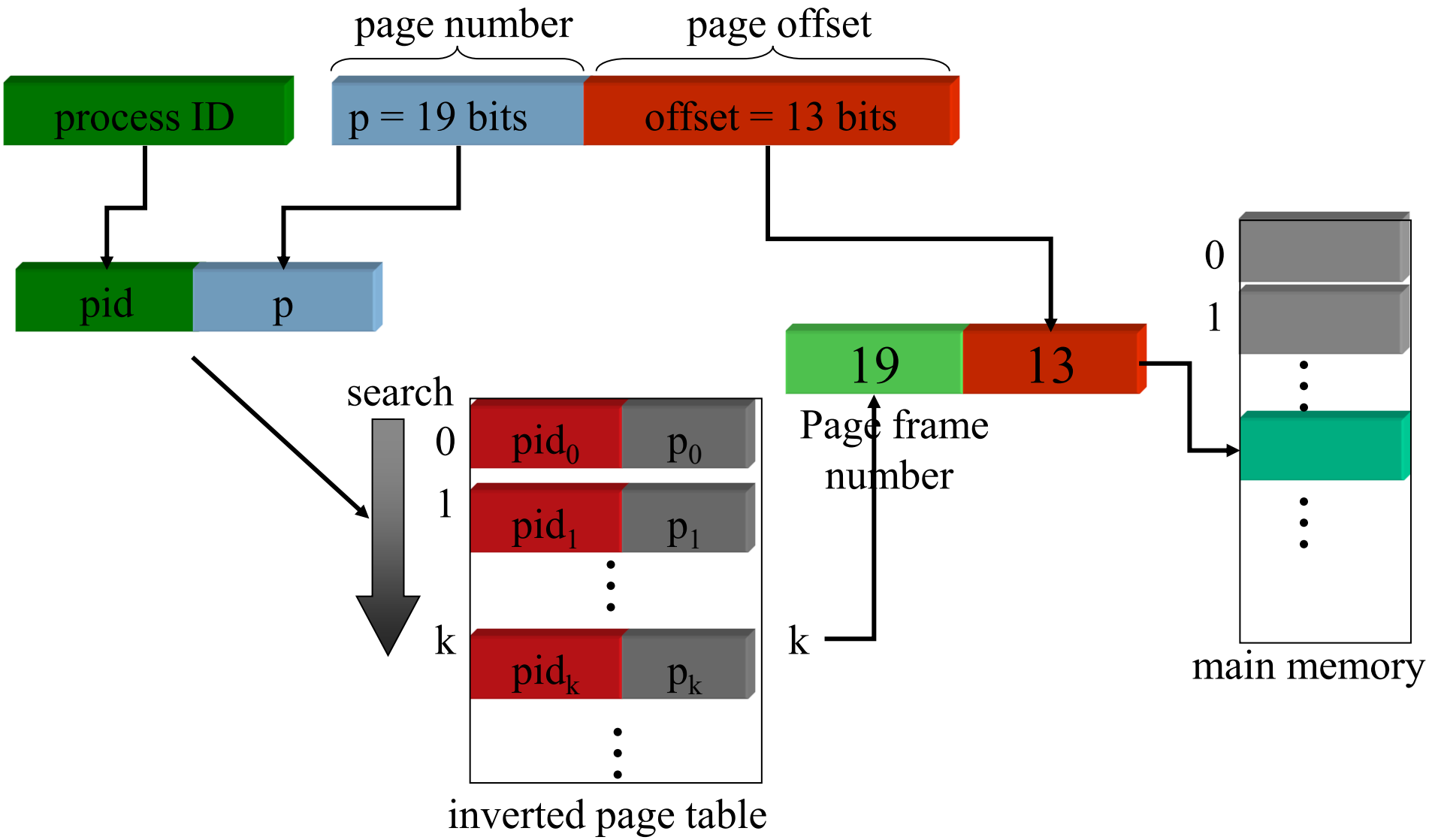
$$EAT = h(a+m) + (1-h)(a+m+m)$$

2-level Page Table → $2m$

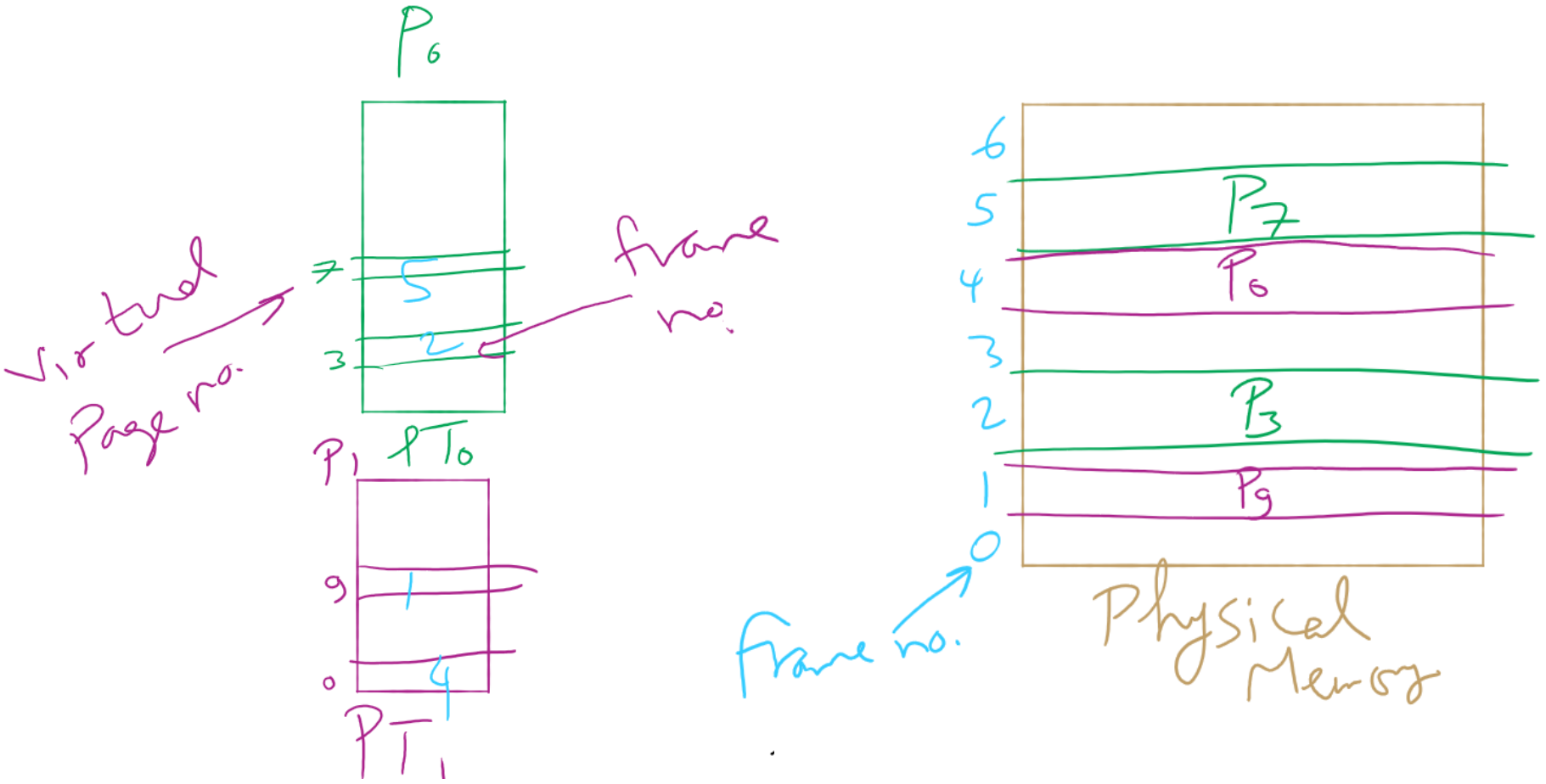
Solution 2: Inverted page table

- One entry for each **frame** in memory
- One table for the entire system
- PTE contains
  - Virtual address pointing to this frame
  - Information about the process that owns this page
- Search page table by
  - Hashing the virtual page number and process ID
  - Starting at the entry corresponding to the hash result
  - Search until either the entry is found or a limit is reached
- Page frame number is **index** of PTE in the table
- Improve performance by using more advanced hashing algorithms

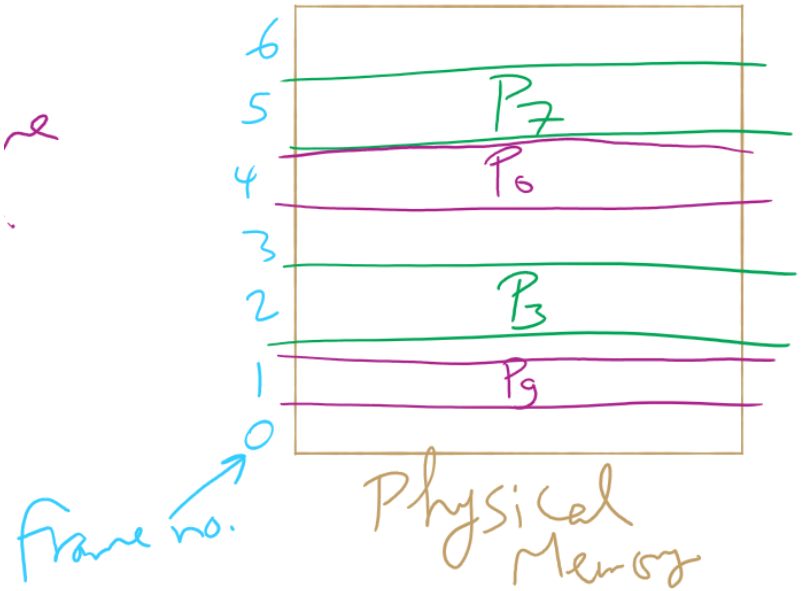
Inverted page table architecture

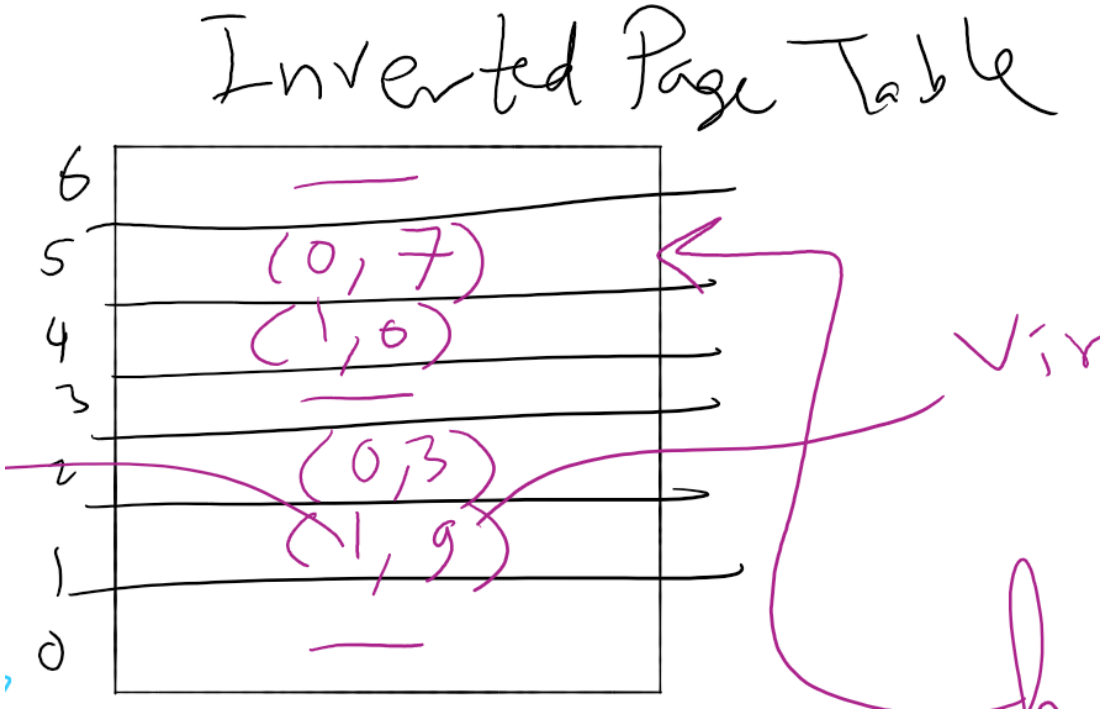


Inverted Page Table Example

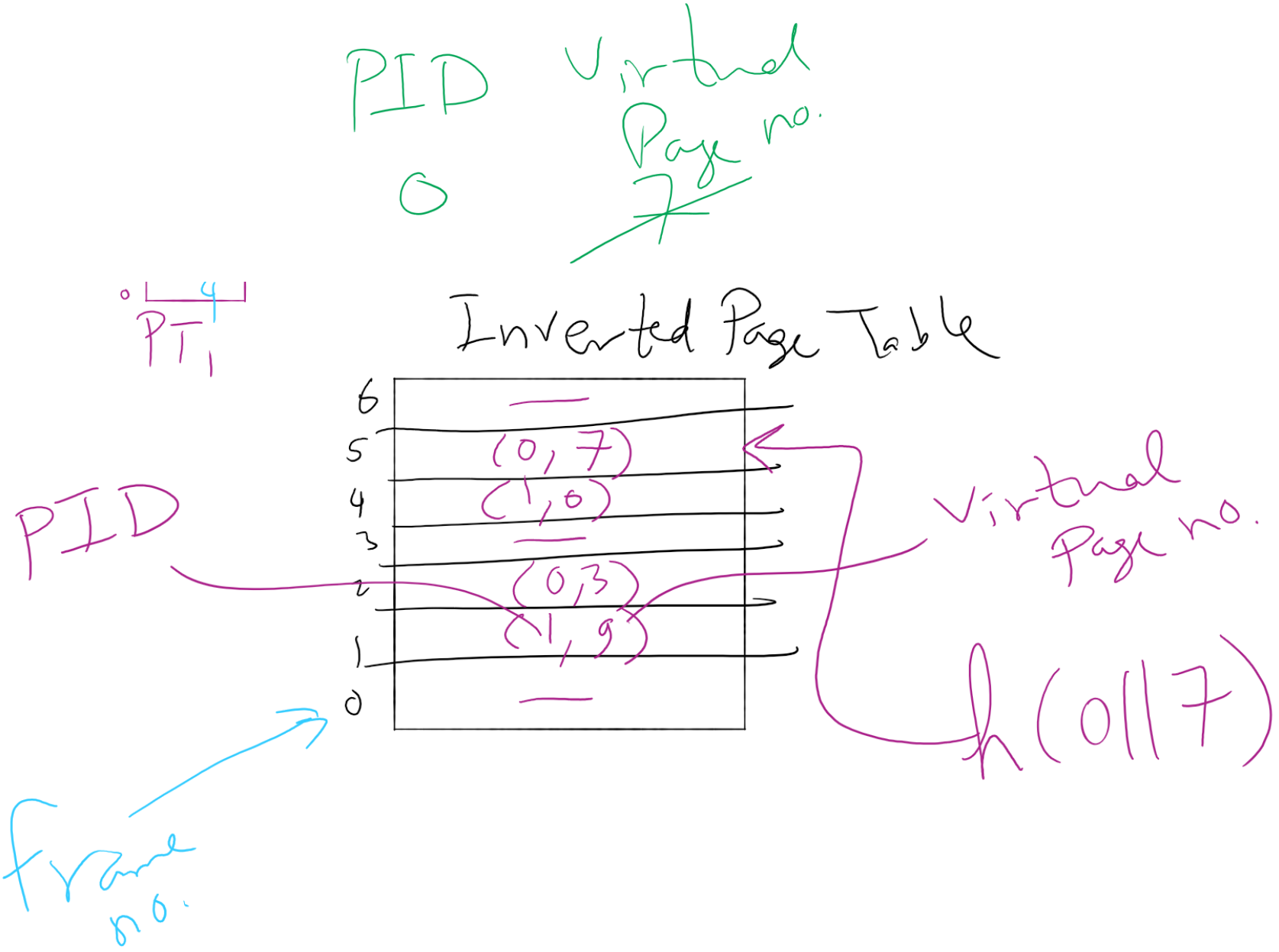


Inverted Page Table Example





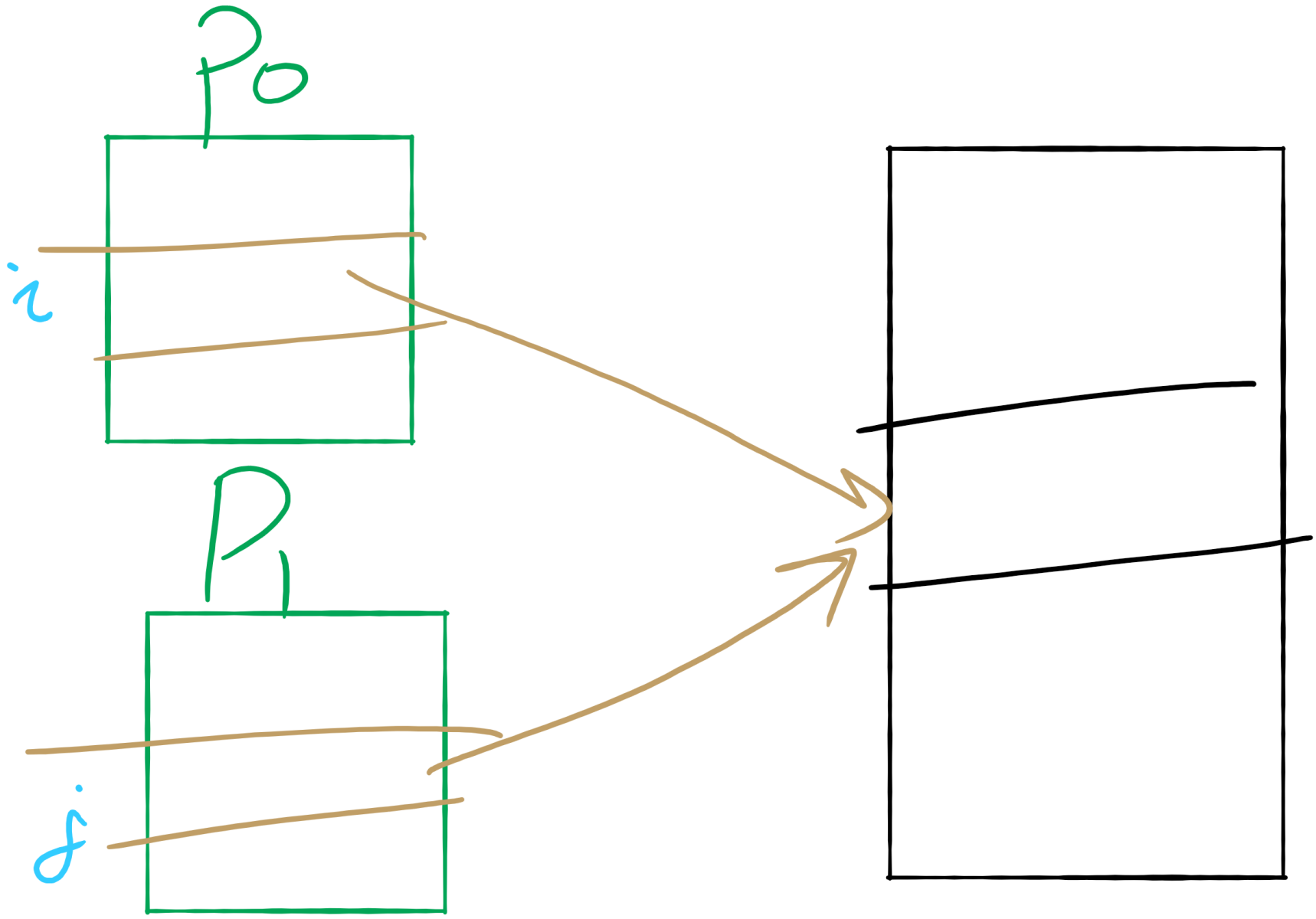
Inverted Page Table Example



How big should a page be?

- Smaller pages have advantages
  - Less internal fragmentation
  - Better fit for various data structures, code sections
  - Less unused physical memory (some pages have 20 useful bytes and the rest isn't needed currently)
- Larger pages are better because
  - Less overhead to keep track of them
    - Smaller page tables
    - TLB can point to more memory (same number of pages, but more memory per page)
    - Faster paging algorithms (fewer table entries to look through)
  - More efficient to transfer larger pages to and from disk

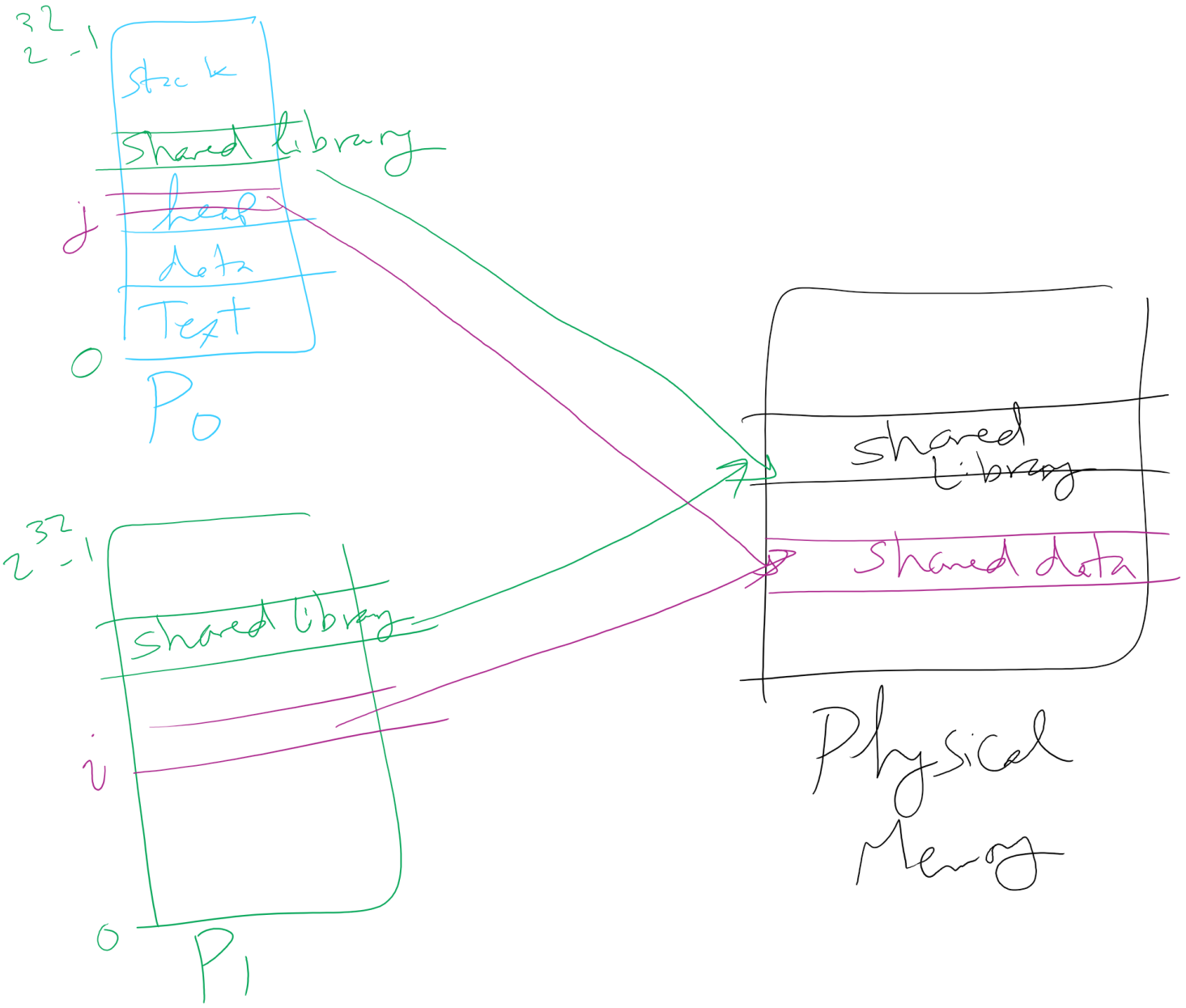
Sharing Pages



Sharing pages

- Processes can share pages
  - Entries in page tables point to the same physical page frame
  - Easier to do with code: no problems with modification
- Virtual addresses in different processes can be…
  - The same: easier to exchange pointers, keep data structures consistent
  - Different: may be easier to actually implement
    - Not a problem if there are only a few shared regions
    - Can be very difficult if many processes share regions with each other

Page Sharing



Implementation issues

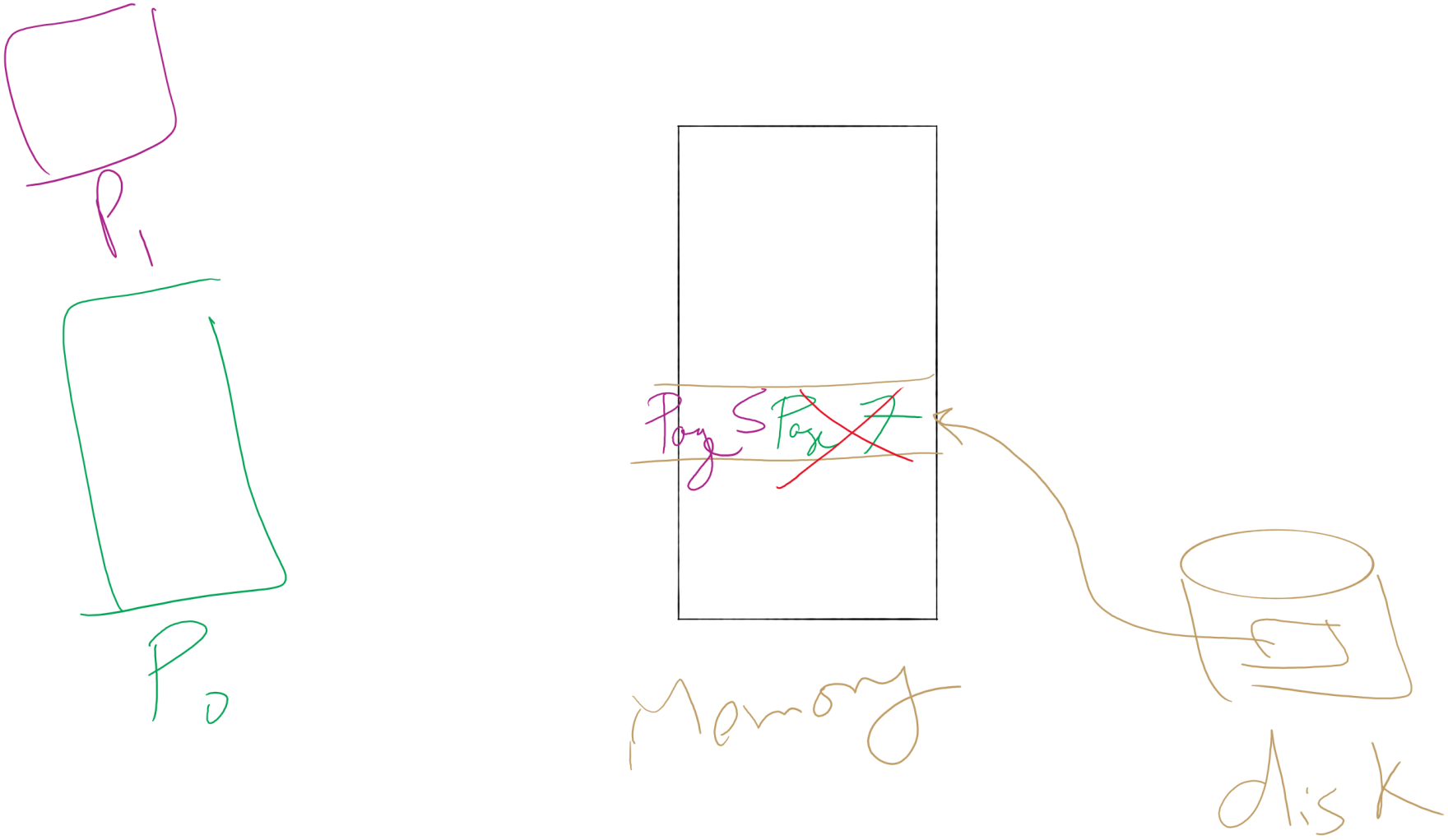
Four times when OS involved with paging

- Process creation
  - Determine program size
  - Create page table
- During process execution
  - Reset the MMU for new process
  - Flush the TLB (or reload it from saved state)
- Page fault time
  - Determine virtual address causing fault
  - Swap target page out, needed page in
- Process termination time
  - Release page table
  - Return pages to the free pool

How is a page fault handled?

- Hardware causes a page fault
- General registers saved (as on every exception)
- OS determines which virtual page needed
  - Actual fault address in a special register
  - Address of faulting instruction in register
    - Page fault was in fetching instruction, or
    - Page fault was in fetching operands for instruction
    - OS must figure out which…
- OS checks validity of address
  - Process killed if address was illegal
- OS finds a place to put new page frame
- If frame selected for replacement is dirty, write it out to disk
- OS requests the new page from disk
- Page tables updated
- Faulting instruction backed up so it can be restarted
- Faulting process scheduled
- Registers restored
- Program continues

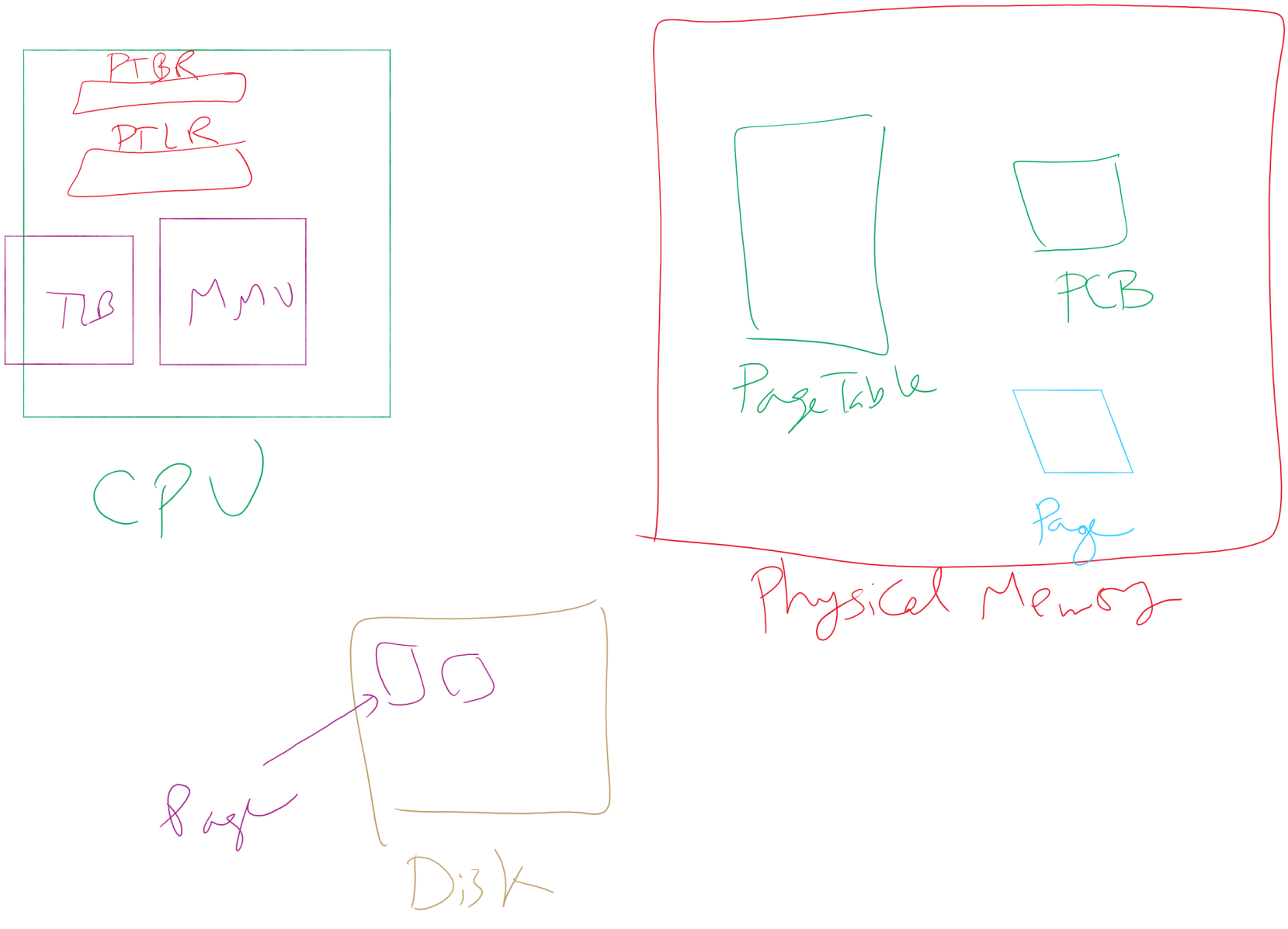
Page locking



Locking pages in memory

- Virtual memory and I/O occasionally interact
- P1 issues call for read from device into buffer
  - While it's waiting for I/O, P2 runs
  - P2 has a page fault
  - P1's I/O buffer might be chosen to be paged out
    - This can create a problem because an I/O device is going to write to the buffer on P1's behalf
- Solution: allow some pages to be *locked* into memory
  - Locked pages are immune from being replaced
  - Pages only stay locked for (relatively) short periods

Map of MMU, TLB, Page Table, etc.



When are dirty pages written to disk?

- On demand (when they're replaced)
  - Fewest writes to disk
  - Slower: replacement takes twice as long (must wait for disk write ***and*** disk read)
- Periodically (in the background)
  - Background process scans through page tables, writes out dirty pages that are pretty old
- Background process also keeps a list of pages ready for replacement
  - Page faults handled faster: no need to find space on demand
  - Cleaner may use the same structures discussed earlier (clock, etc.)

Control overall page fault rate

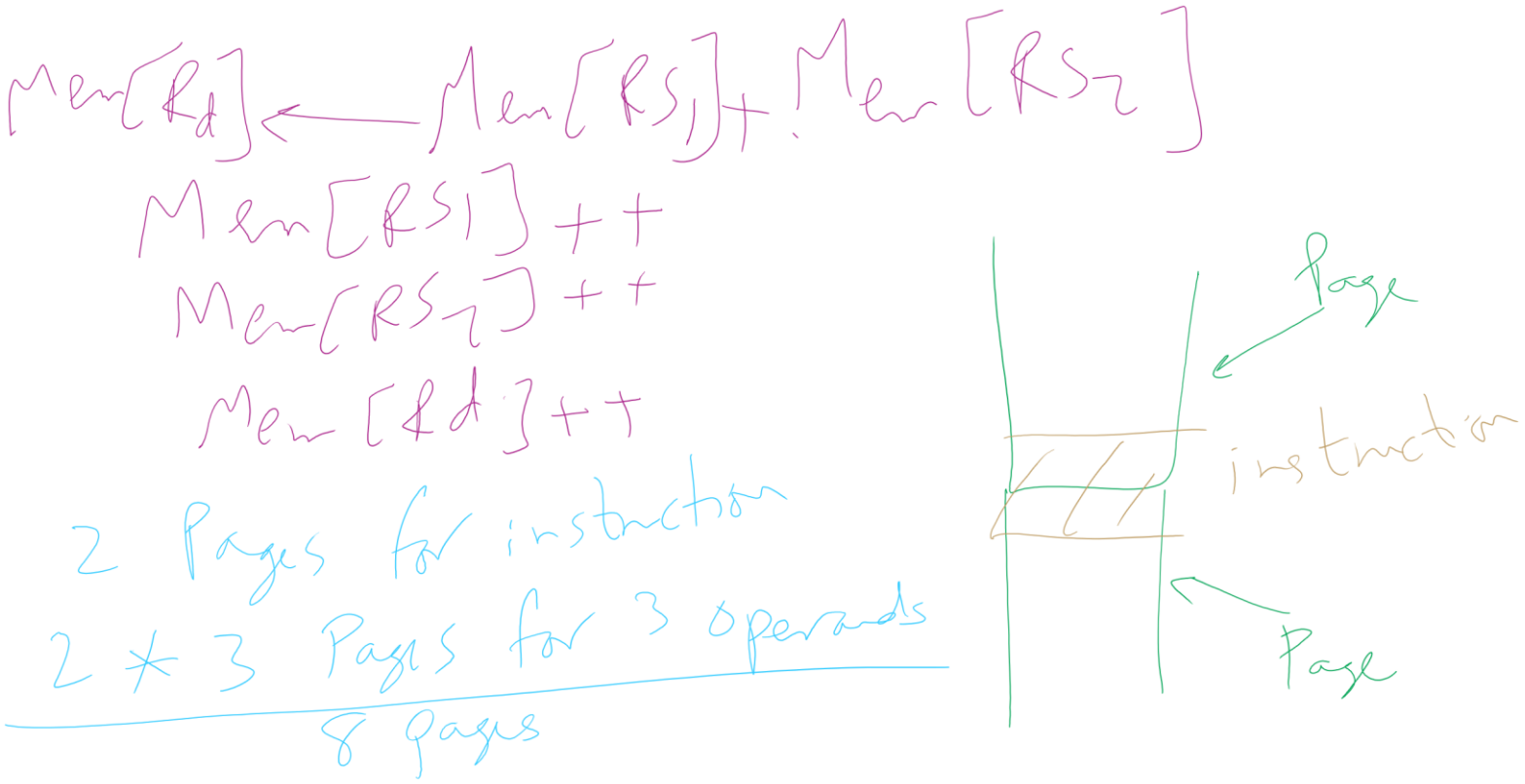
- Despite good designs, system may still thrash
- Most (or all) processes have high page fault rate
  - Some processes need more memory, …
  - but no processes need less memory (and could give some up)
- Problem: no way to reduce page fault rate
- Solution :
  Reduce number of processes competing for memory
  - Swap one or more to disk, divide up pages they held
  - Reconsider degree of multiprogramming

Backing up an instruction

- Problem: page fault happens in the middle of instruction execution
  - Some changes may have already happened
  - Others may be waiting for VM to be fixed
- Solution: undo all of the changes made by the instruction
  - Restart instruction from the beginning
  - This is easier on some architectures than others
- Example: LW R1, 12(R2)
  - Page fault in fetching instruction: nothing to undo
  - Page fault in getting value at 12(R2): restart instruction

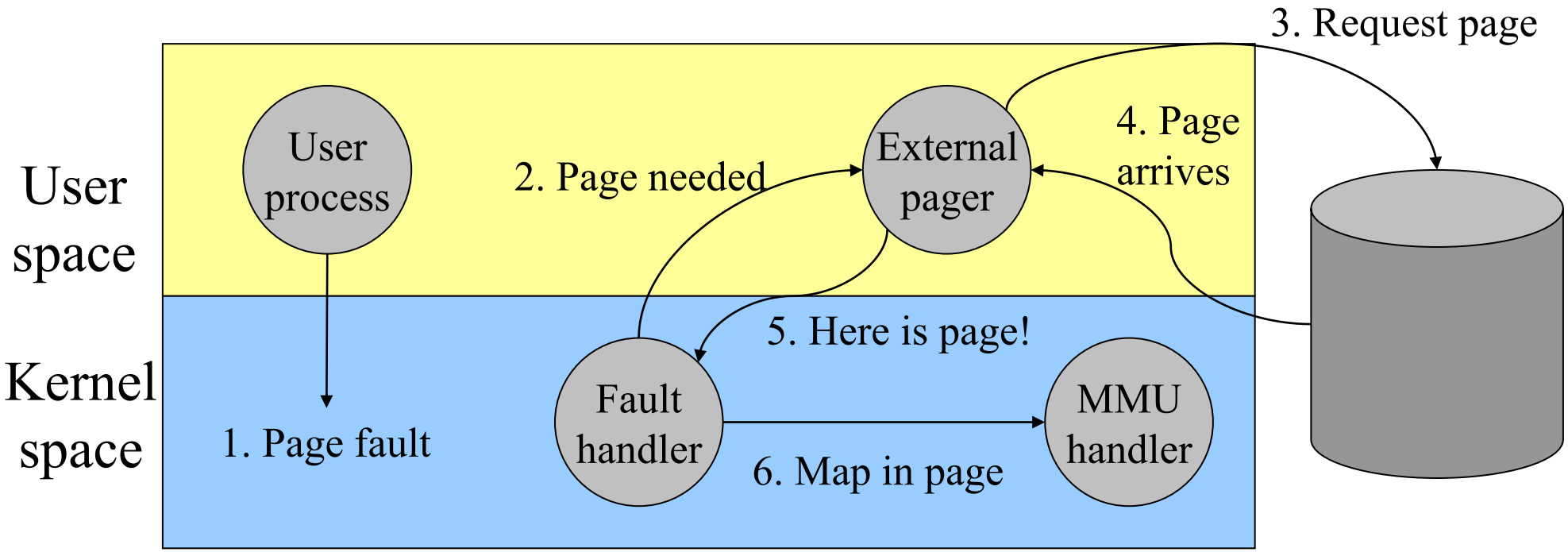
Minimum memory allocation to a process

- Example: ADD (Rd)+,(Rs1)+,(Rs2)+
  - Page fault in writing to (Rd): may have to undo an awful lot…



Separating policy and mechanism

- Mechanism for page replacement has to be in kernel
  - Modifying page tables
  - Reading and writing page table entries
- Policy for deciding which pages to replace could be in user space
  - More flexibility



Separating Policy and Mechanism for Page Replacement