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
  - Lab 3 is due on Tuesday 3/28 at 11:59 pm
  - Project 3 is due Friday 4/7 at 11:59 pm

Previous lecture …

- How to simulate page replacement algorithms
  - FIFO

This lecture …

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

FIFO Example 1

FIFO with 4 frames

Belady's anomaly

- Reduce the number of page faults by supplying more memory
  - Use previous reference string and FIFO algorithm
  - Add another page to physical memory (total 4 pages)
- More page faults (10 vs. 9), not fewer!
  - This is called *Belady's anomaly*
  - Adding more pages shouldn't result in worse performance!
- Motivated the study of paging algorithms

CLOCK Simulation

Modeling more replacement algorithms

- Paging system characterized by:
  - Reference string of executing process
  - Page replacement algorithm
  - Number of page frames available in physical memory (*m*)
- Model this by keeping track of all *n* pages referenced in array *M*
  - Top part of *M* has *m* pages in memory
  - Bottom part of *M* has *n-m* pages stored on disk
- Page replacement occurs when page moves from top to bottom
  - Top and bottom parts may be rearranged without causing movement between memory and disk

Example: LRU

- Model LRU replacement with
  - 8 unique references in the reference string
  - 4 pages of physical memory
- Array state over time shown below
- LRU treats list of pages like a stack

	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
		0	2	1	3	5	4	6	3	7	4	7	7	3	3	5	3	3	3	1	7	1	3	4
			0	2	1	3	5	4	6	3	3	4	4	7	7	7	5	5	5	3	3	7	1	3
				0	2	1	3	5	4	6	6	6	6	4	4	4	7	7	7	5	5	5	7	7
					0	2	1	1	5	5	5	5	5	6	6	6	4	4	4	4	4	4	5	5
						0	2	2	1	1	1	1	1	1	1	1	6	6	6	6	6	6	6	6
							0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Stack algorithms

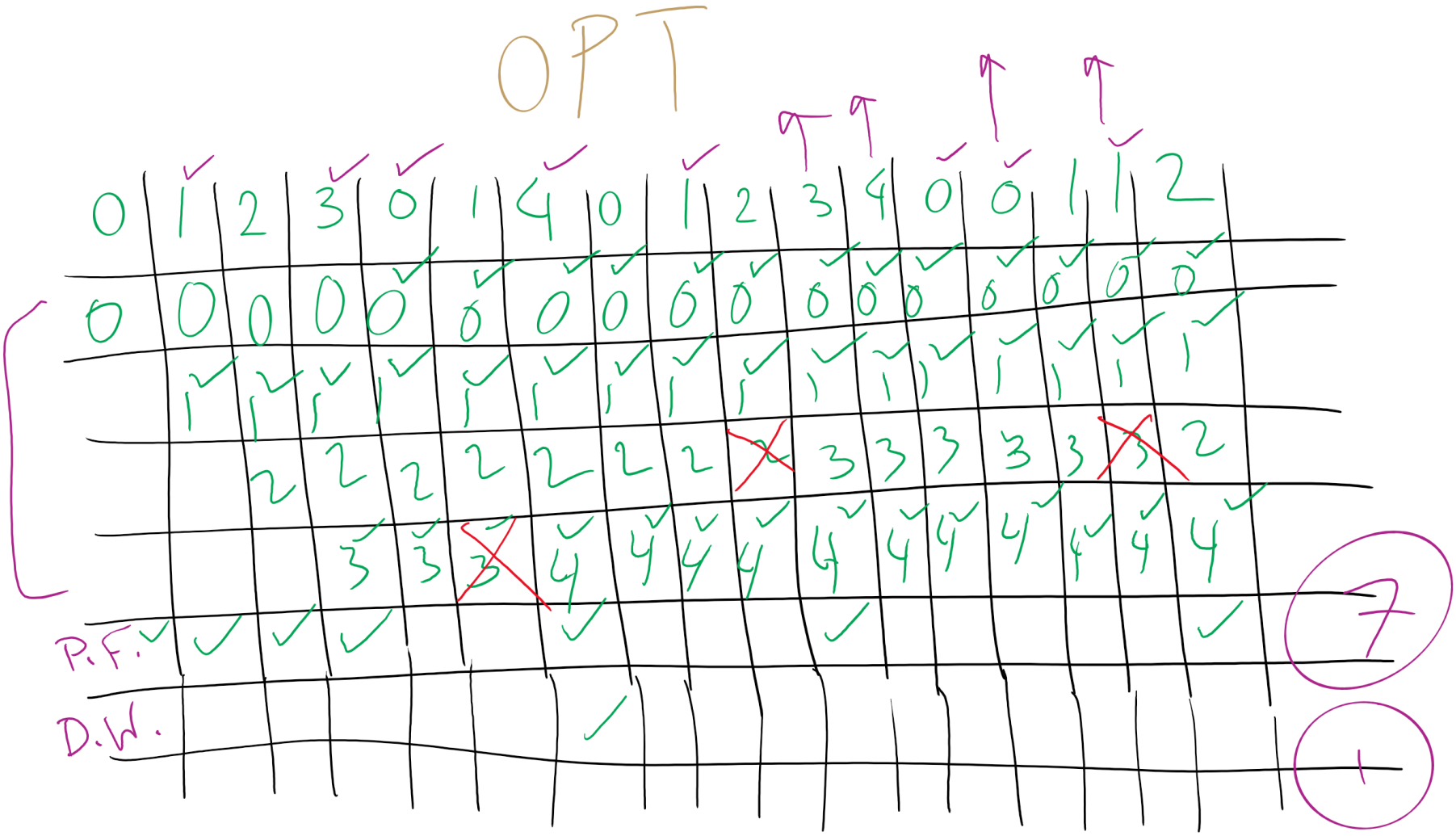
- LRU is an example of a stack algorithm
- For stack algorithms
  - Any page in memory with *m* physical pages is also in memory with *m*+1 physical pages
  - Increasing memory size is guaranteed to reduce (or at least not increase) the number of page faults
- Stack algorithms do not suffer from Belady's anomaly
- *Distance* of a reference == position of the page in the stack before the reference was made
  - Distance is ∞ if no reference had been made before
  - Distance depends on reference string and paging algorithm: might be different for LRU and optimal (both stack algorithms)

Predicting page fault rates using distance

- Distance can be used to predict page fault rates
- Make a single pass over the reference string to generate the distance string on-the-fly
- Keep an array of counts
  - Entry *j* counts the number of times distance *j* occurs in the distance string
- The number of page faults for a memory of size *m* is the sum of the counts for *j*>*m*
  - This can be done in a single pass!
  - Makes for fast simulations of page replacement algorithms
- This is why virtual memory theorists like stack algorithms!

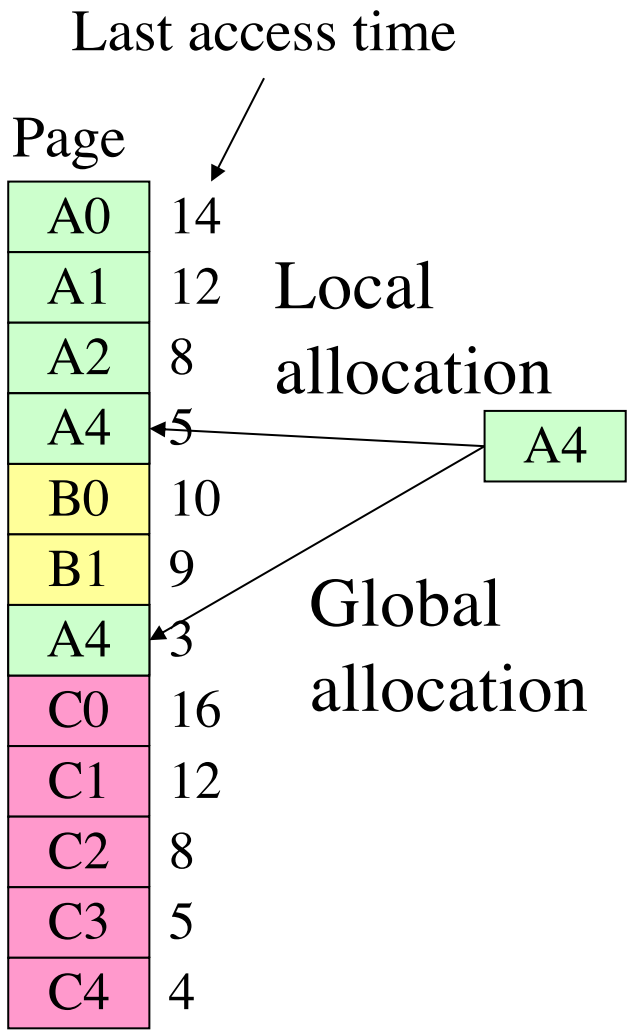
LRU

OPT



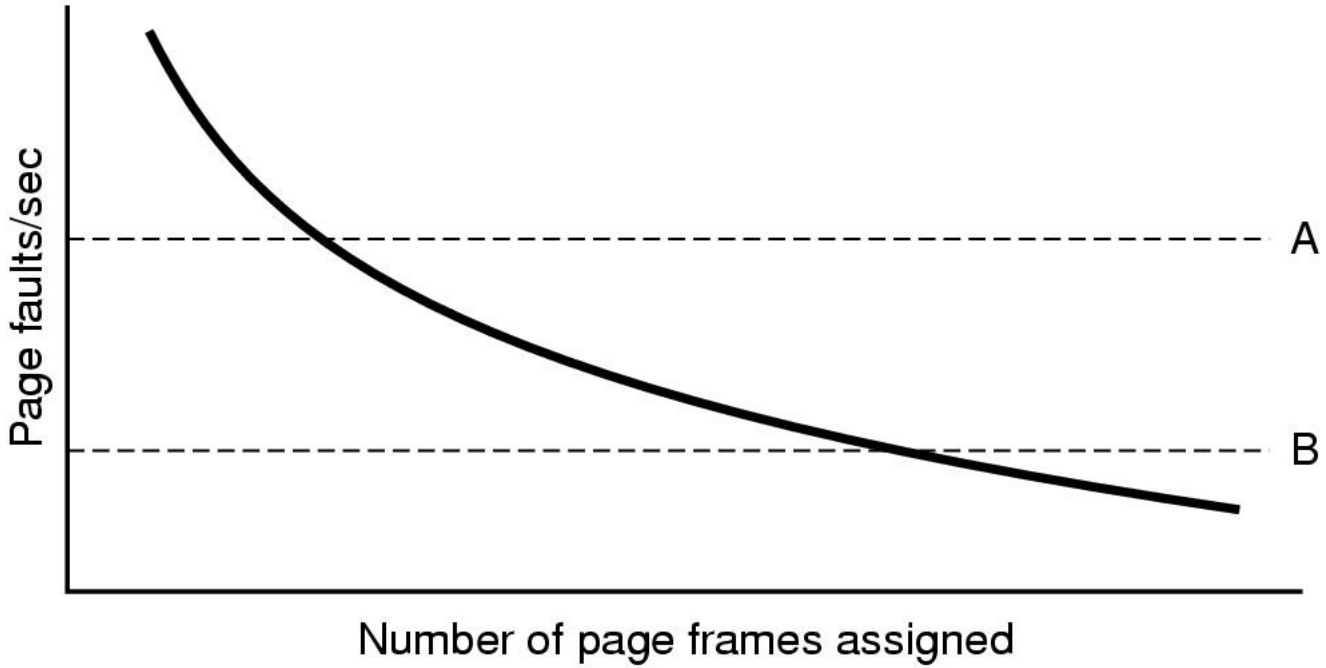
Local vs. global allocation policies

- What is the pool of pages eligible to be replaced?
  - Pages belonging to the process needing a new page
  - All pages in the system
- Local allocation: replace a page from this process
  - May be more “fair”: penalize processes that replace many pages
  - Can lead to poor performance: some processes need more pages than others
- Global allocation: replace a page from *any* process



Page fault rate vs. allocated frames

- Local allocation may be more “fair”
  - Don’t penalize other processes for high page fault rate
- Global allocation is better for overall system performance
  - Take page frames from processes that don’t need them as much
  - Reduce the overall page fault rate (even though rate for a single process may go up)



Control overall page fault rate

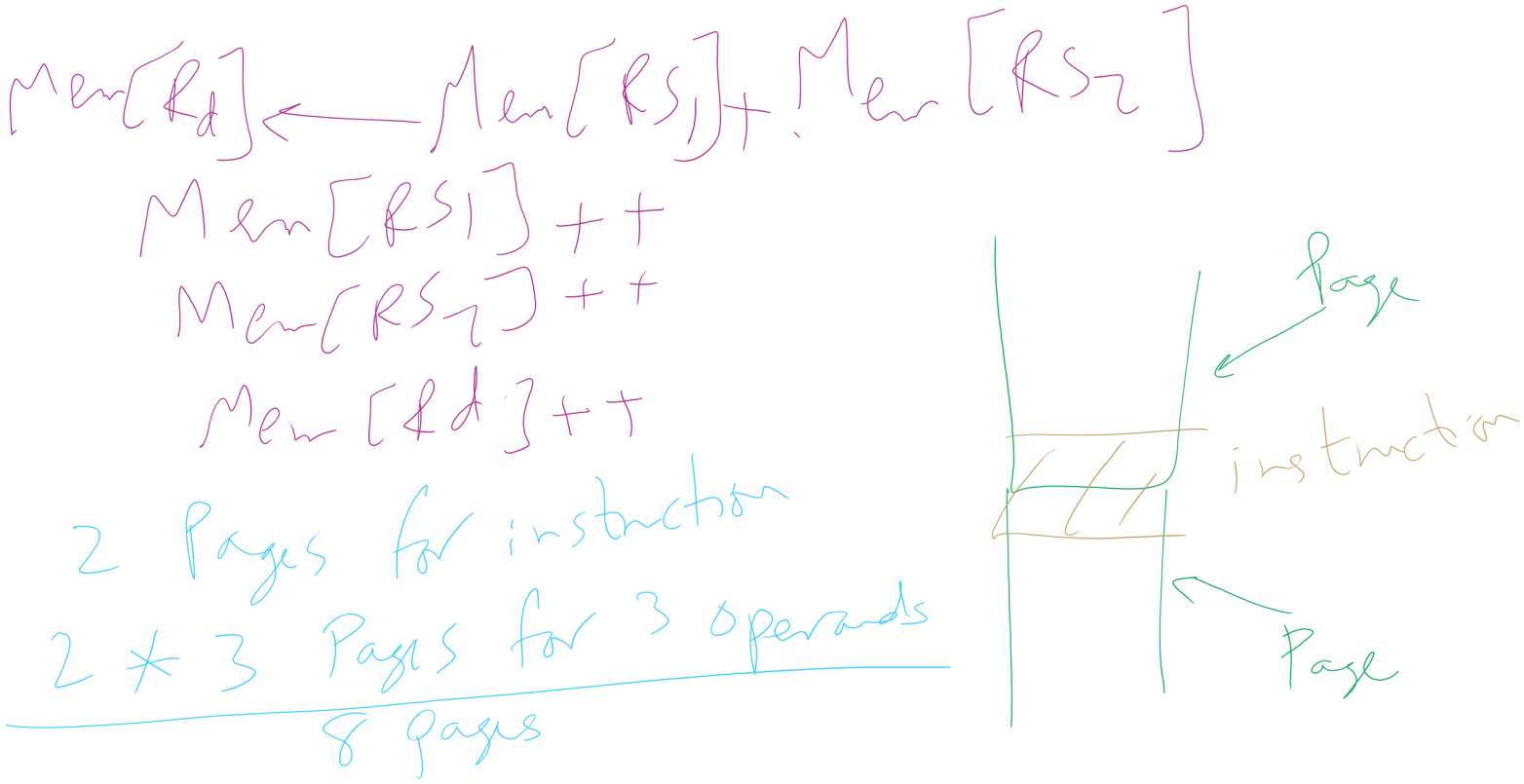
- Despite good designs, system may still thrash
- Most (or all) processes have high page fault rate
  - Some processes need more memory, …
  - but no processes need less memory (and could give some up)
- Problem: no way to reduce page fault rate
- Solution :
  Reduce number of processes competing for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

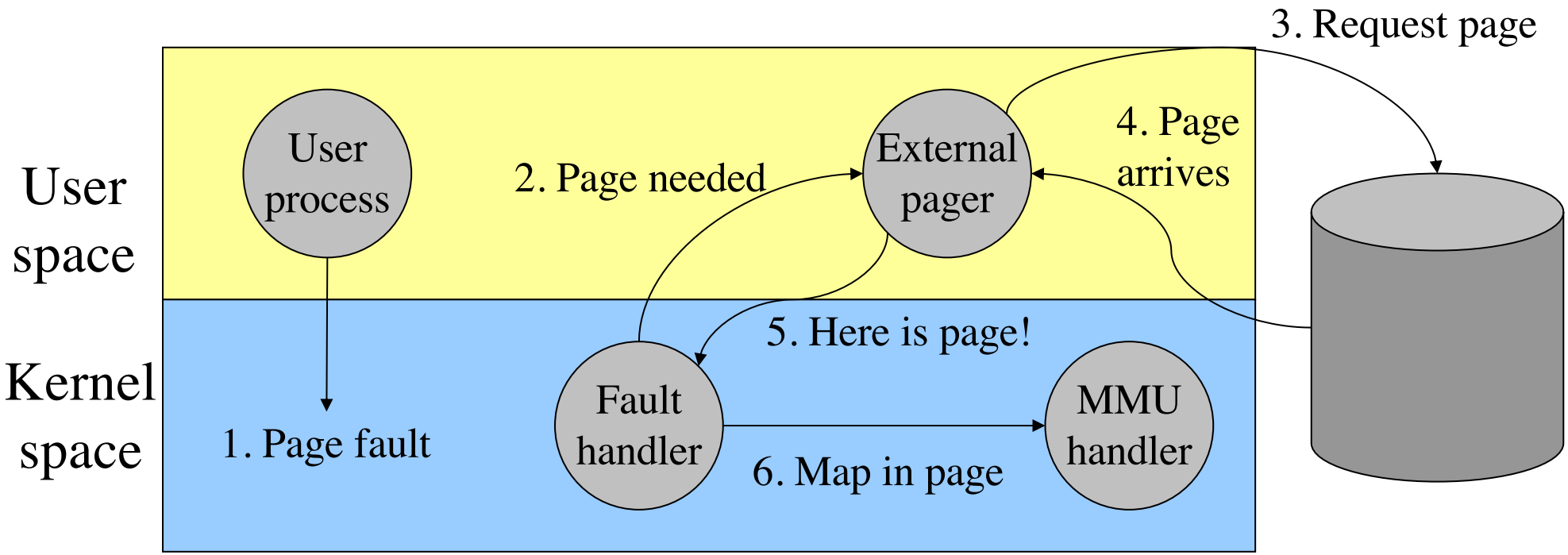
Minimum memory allocation to a process

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

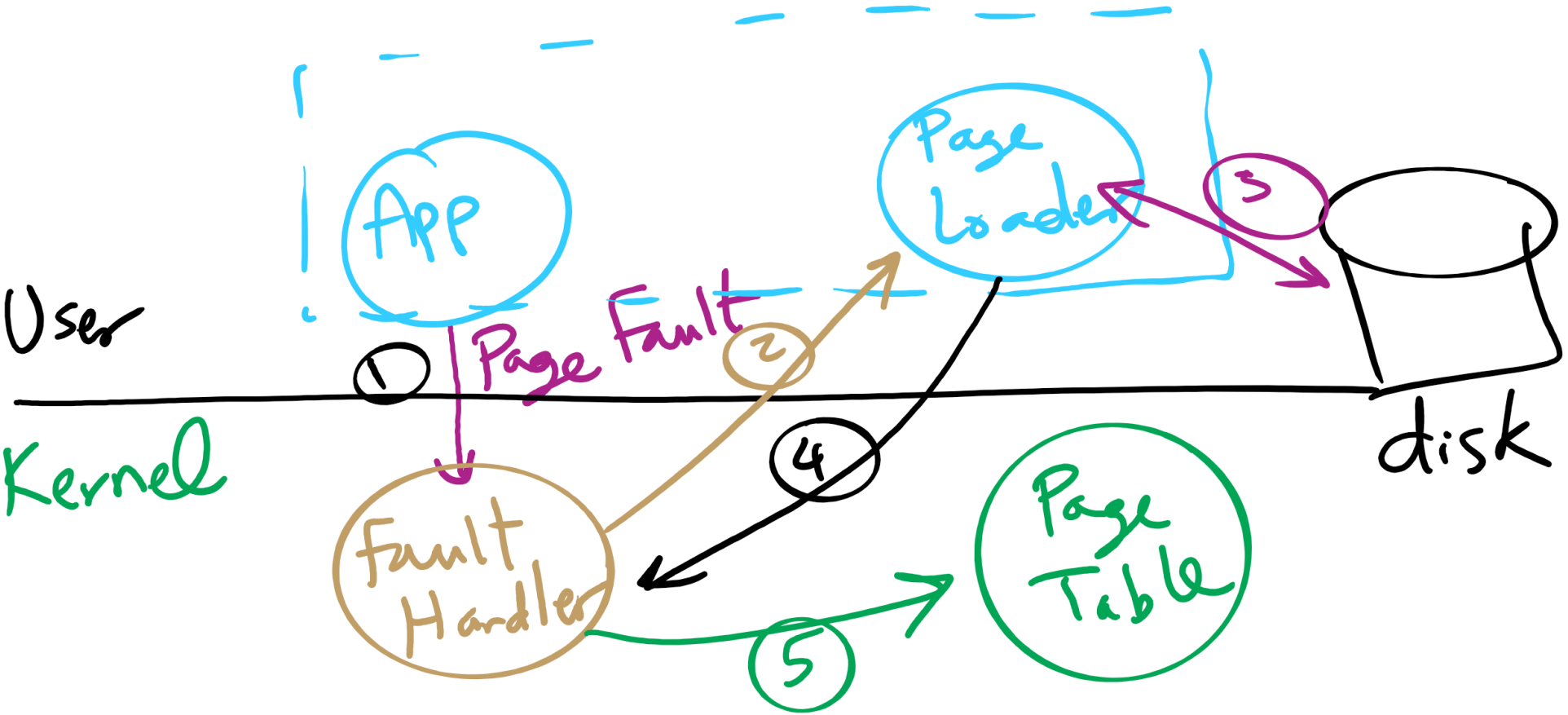


Separating policy and mechanism

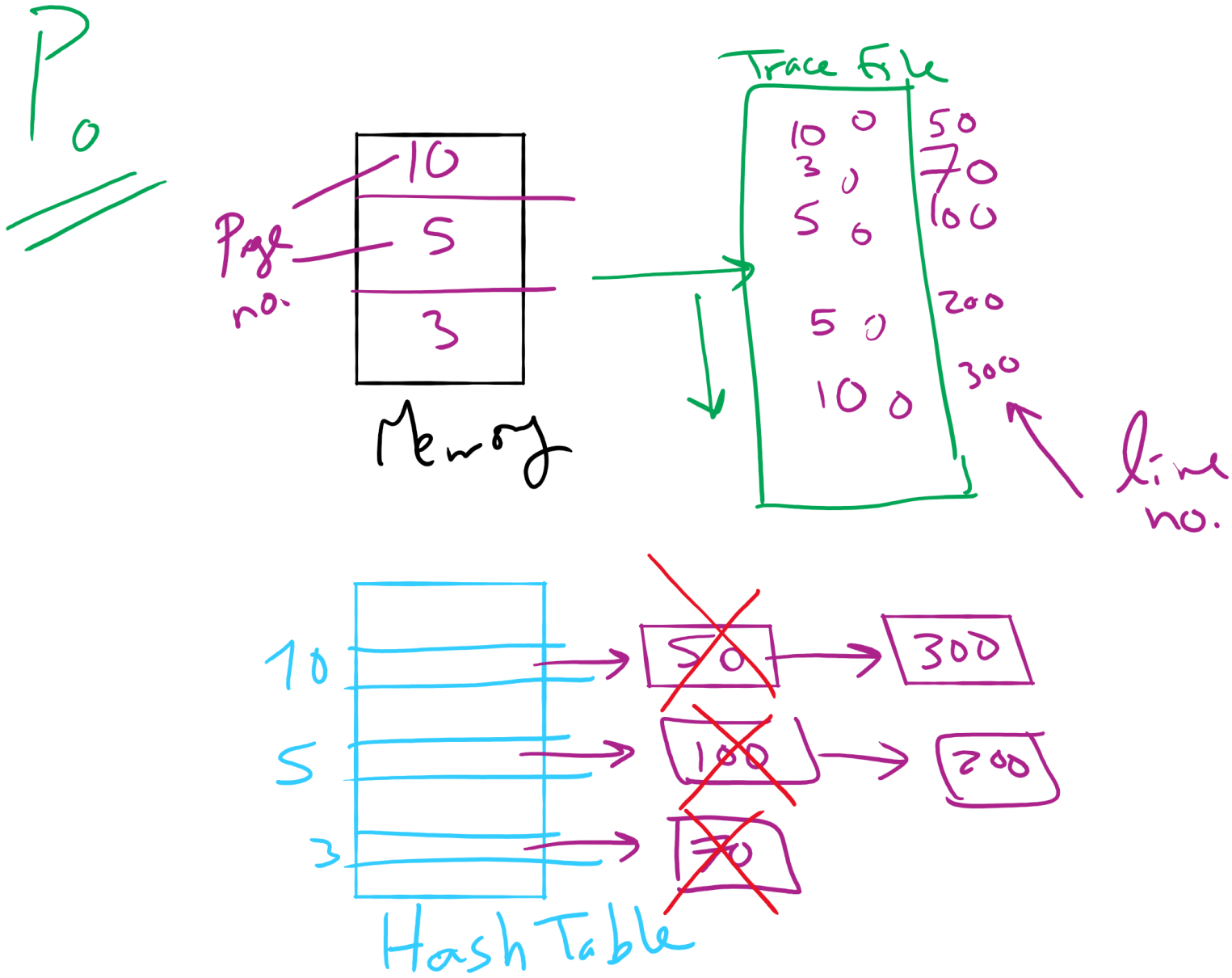
- Mechanism for page replacement has to be in kernel
  - Modifying page tables
  - Reading and writing page table entries
- Policy for deciding which pages to replace could be in user space
  - More flexibility



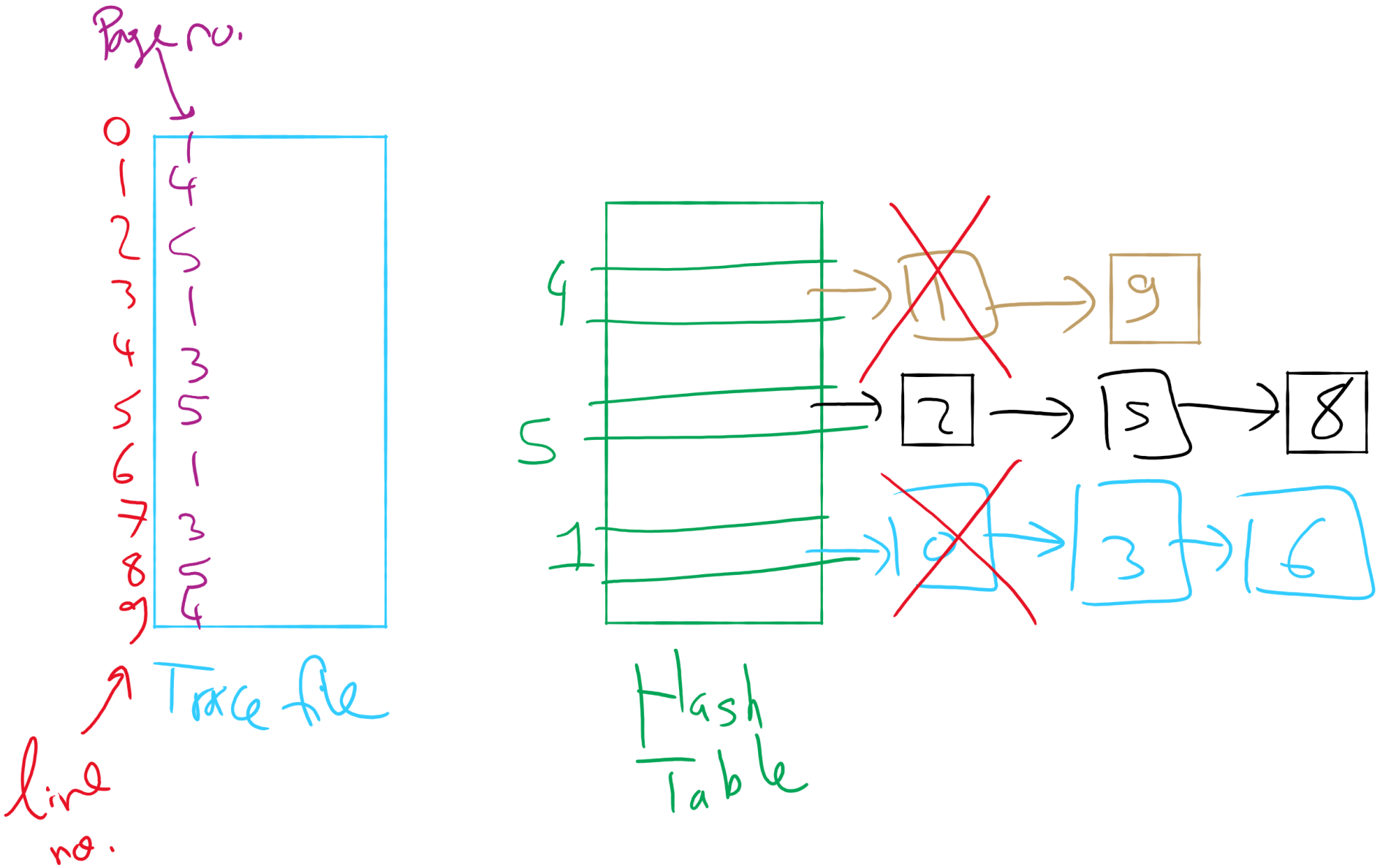
Separating Policy and Mechanism for Page Replacement



Project 3: OPT Simulation

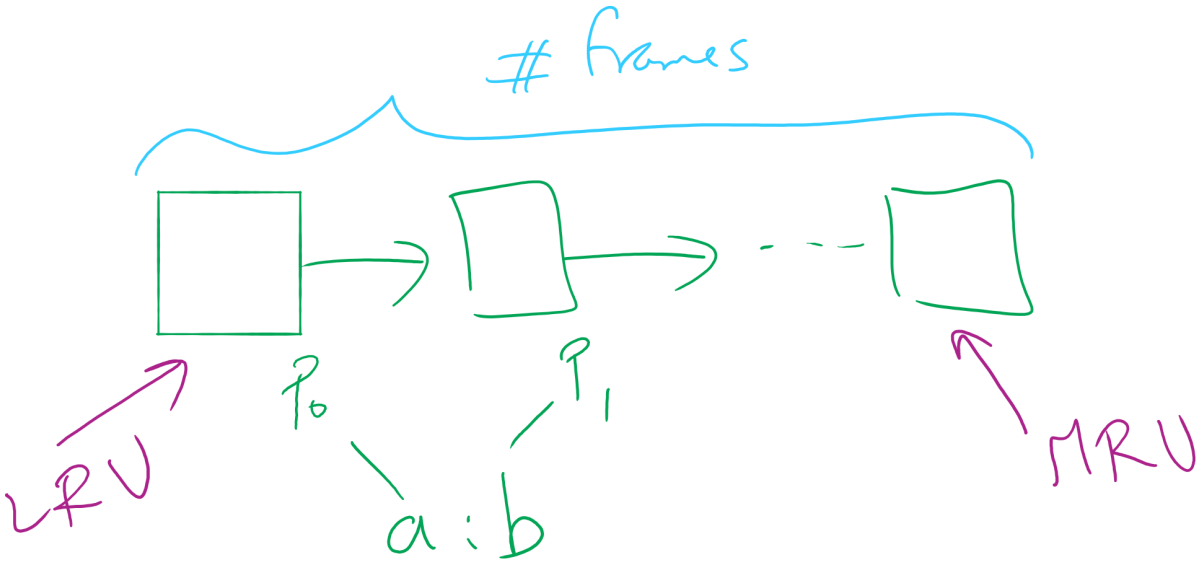


OPT Implementation Example



Project 3: LRU and miscellaneous hints

gunzip 1.trace.gz

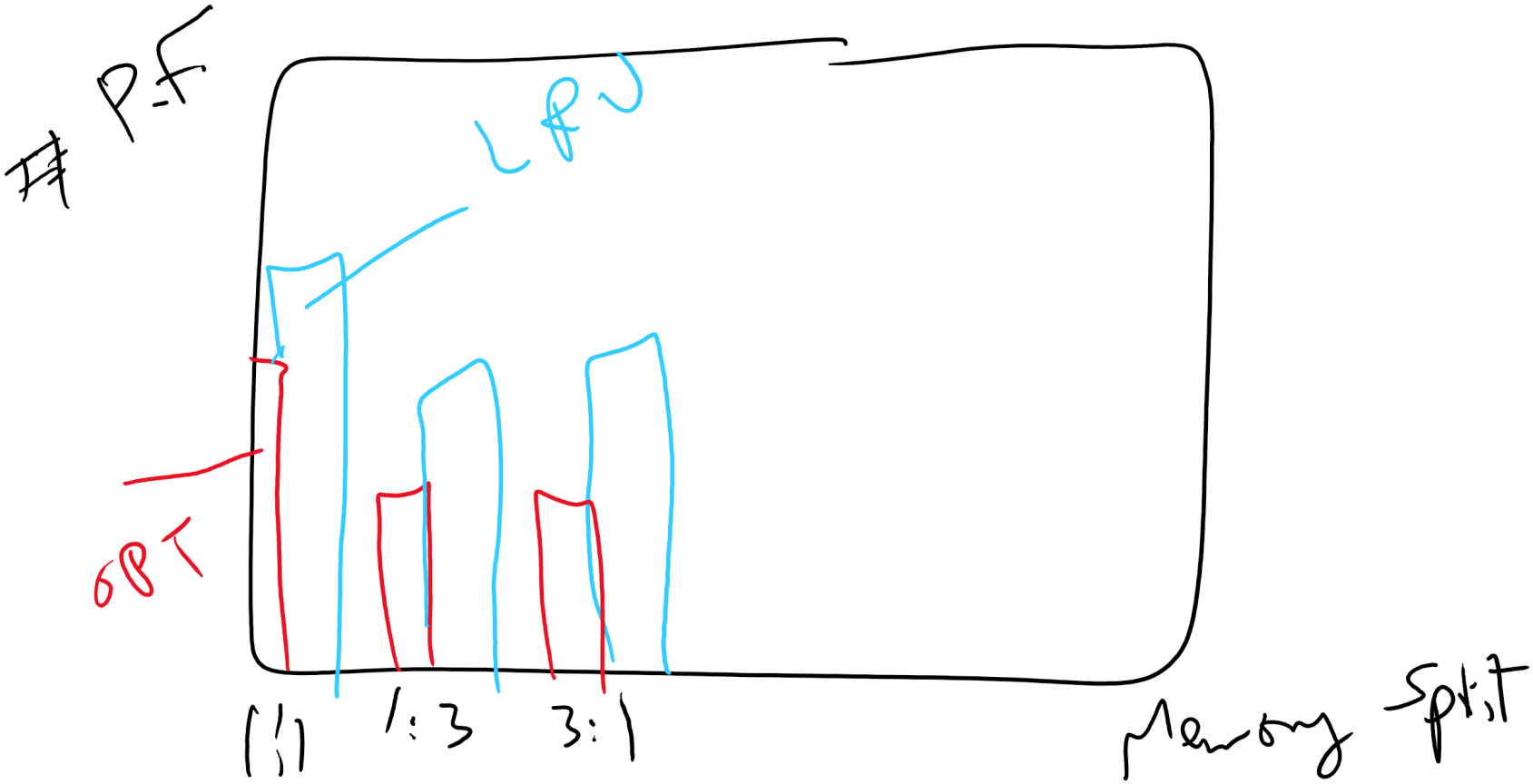


$n \div (a+b) == 0$

$Frames_0 = \left(\frac{n}{a+b}\right) * a$

$Frames_1 = n - Frames_0$

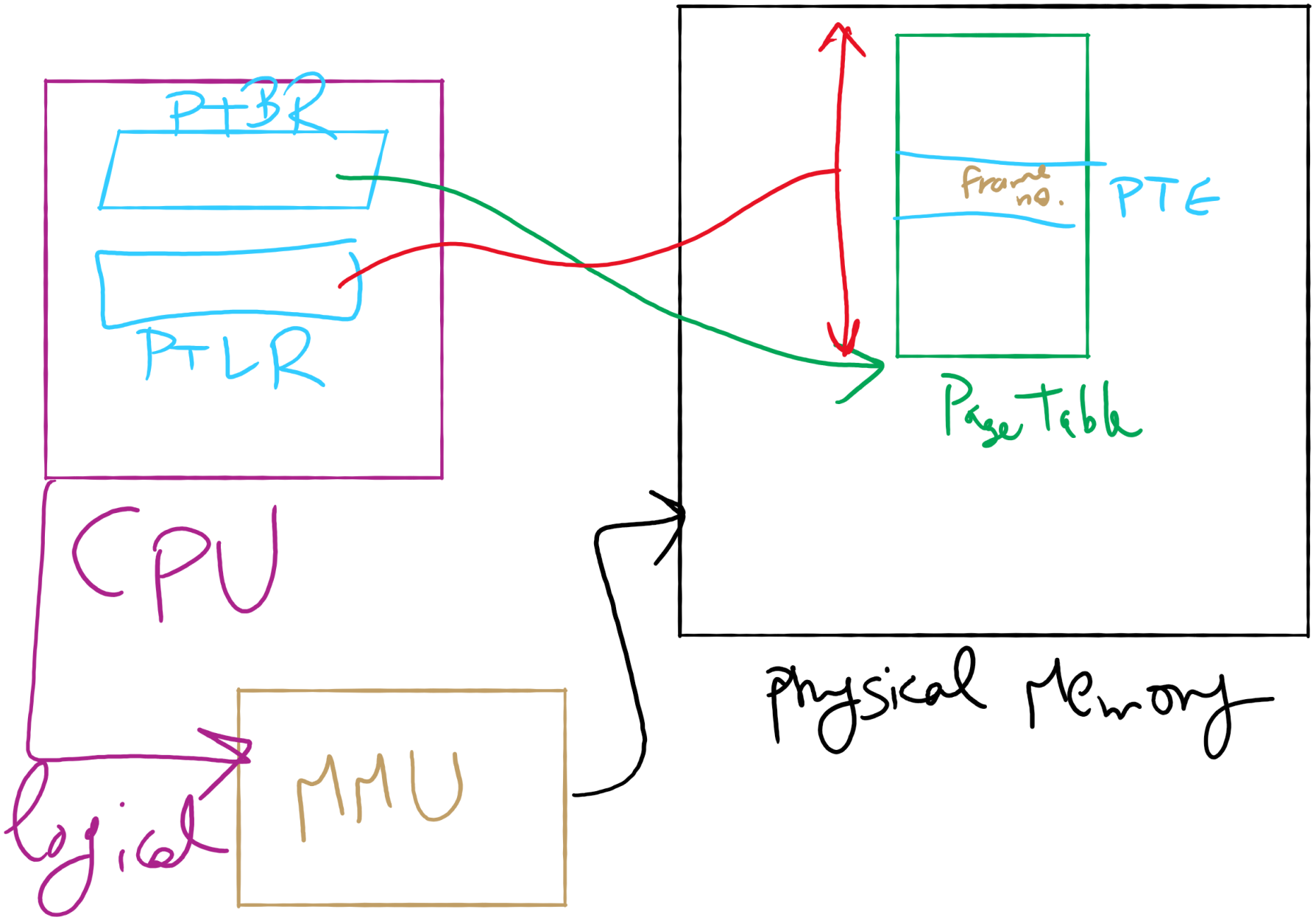
Project 3: Writeup hints



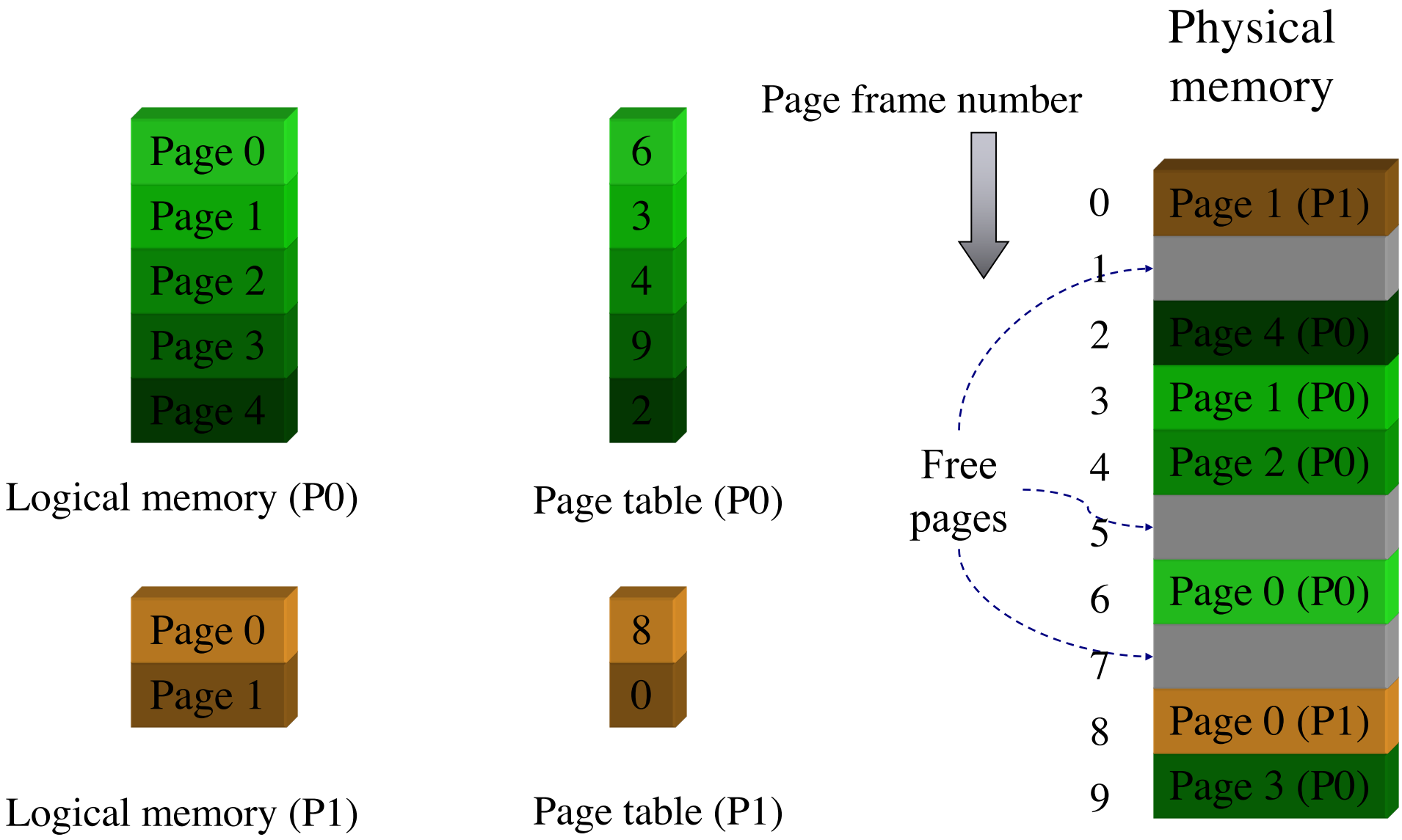
Problem of the Day

- How to keep track of larger page tables that can store more pages
- How big can a page table be?
  - 64-bit machine
  - 4 KB page size
  - How many pages?
  - How many PTE?
  - How big is a PTE?
  - How big is the page table of one process?

Address Translation Structures



Memory & paging structures

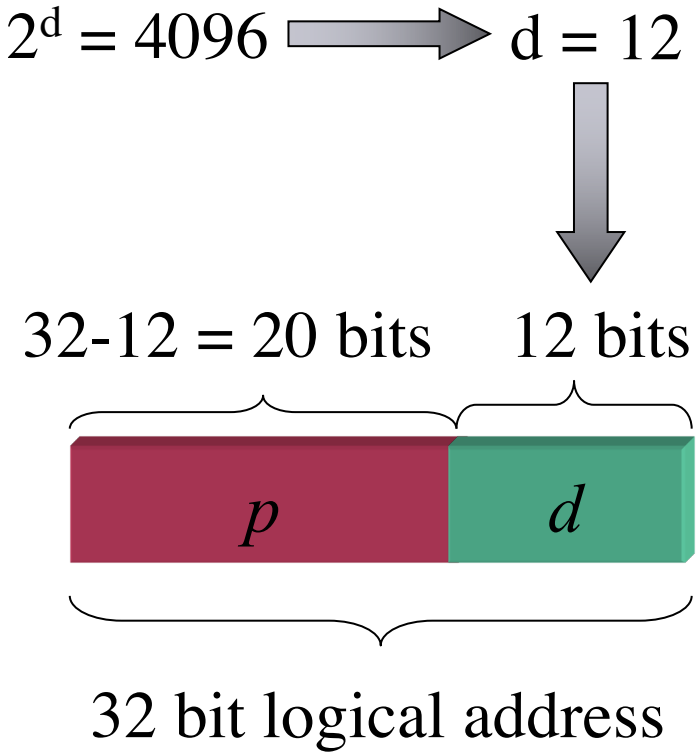


Mapping logical => physical address

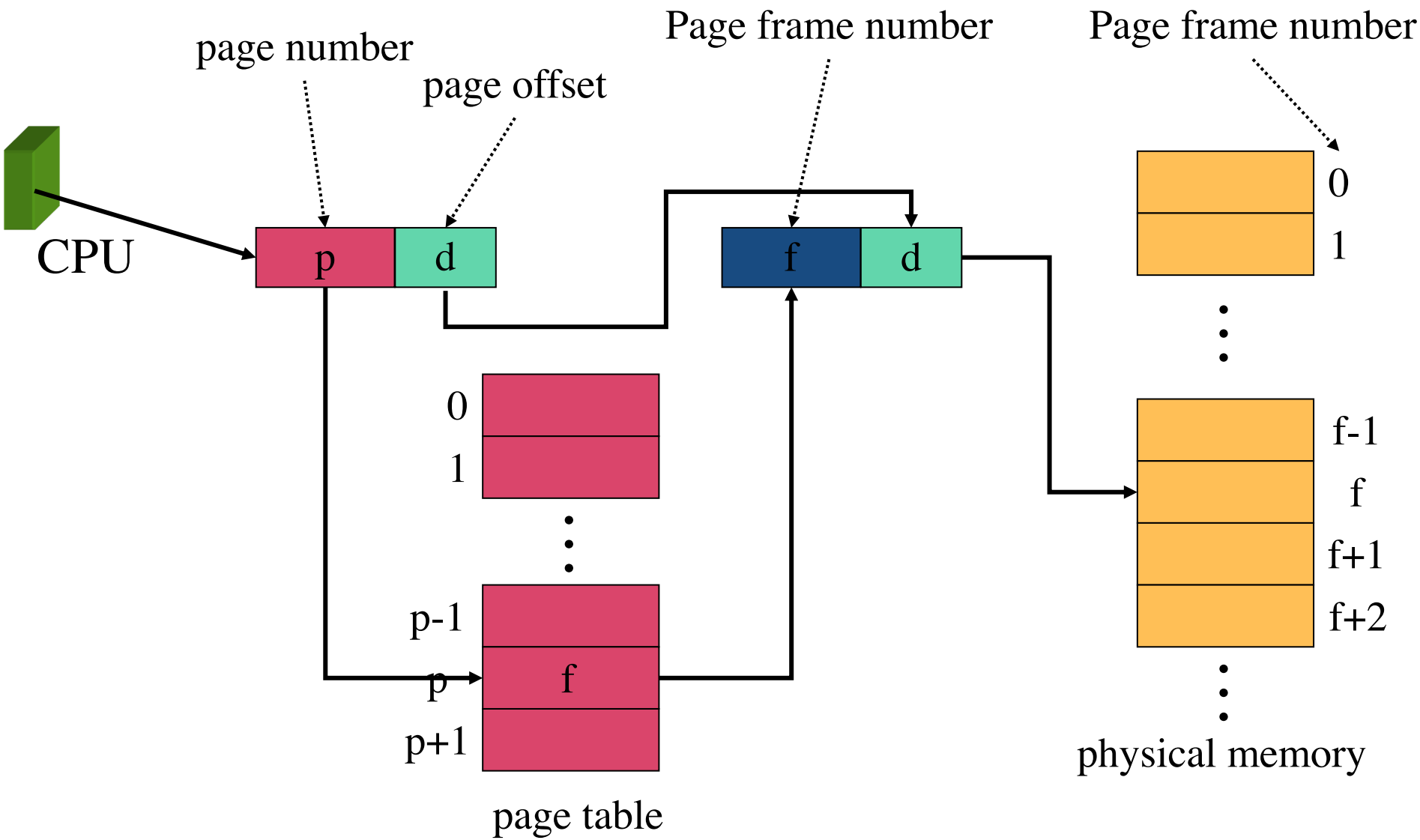
- Split address from CPU into two pieces
  - Page number (*p*)
  - Page offset (*d*)
- Page number
  - Index into page table
  - Page table contains base address of page in physical memory
- Page offset
  - Added to base address to get actual physical memory address
- Page size = 2^d bytes

Example:

- 4 KB (=4096 byte) pages
- 32 bit logical addresses



Address translation architecture



Memory Sizes

KiB = 2^{10}

MiB = 2^{20}

GiB = 2^{30}

TiB = 2^{40}

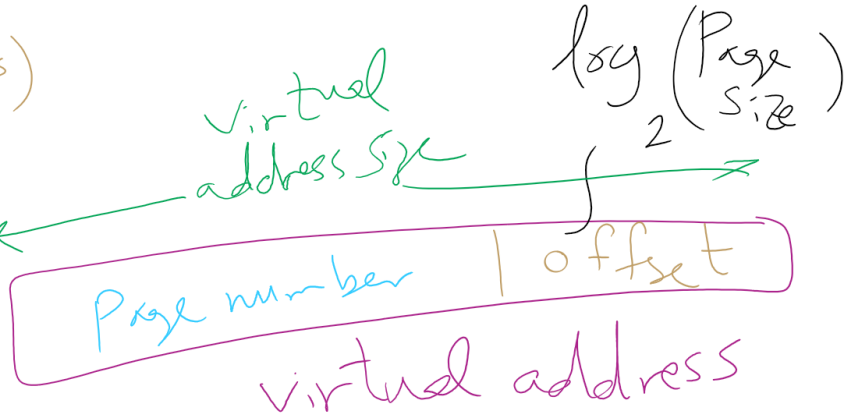
PiB = 2^{50}

Address Translation Relations

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

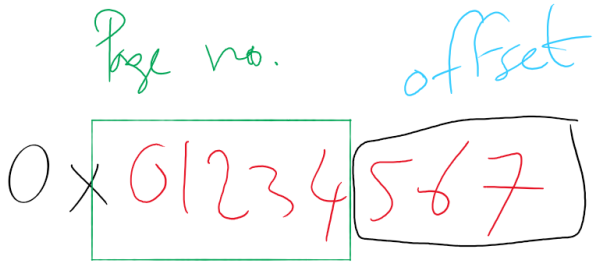
$$= 2^{(\text{Page no. size})}$$

$$\text{Page Table size} = \#PTEs \times PTE\ size$$



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

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

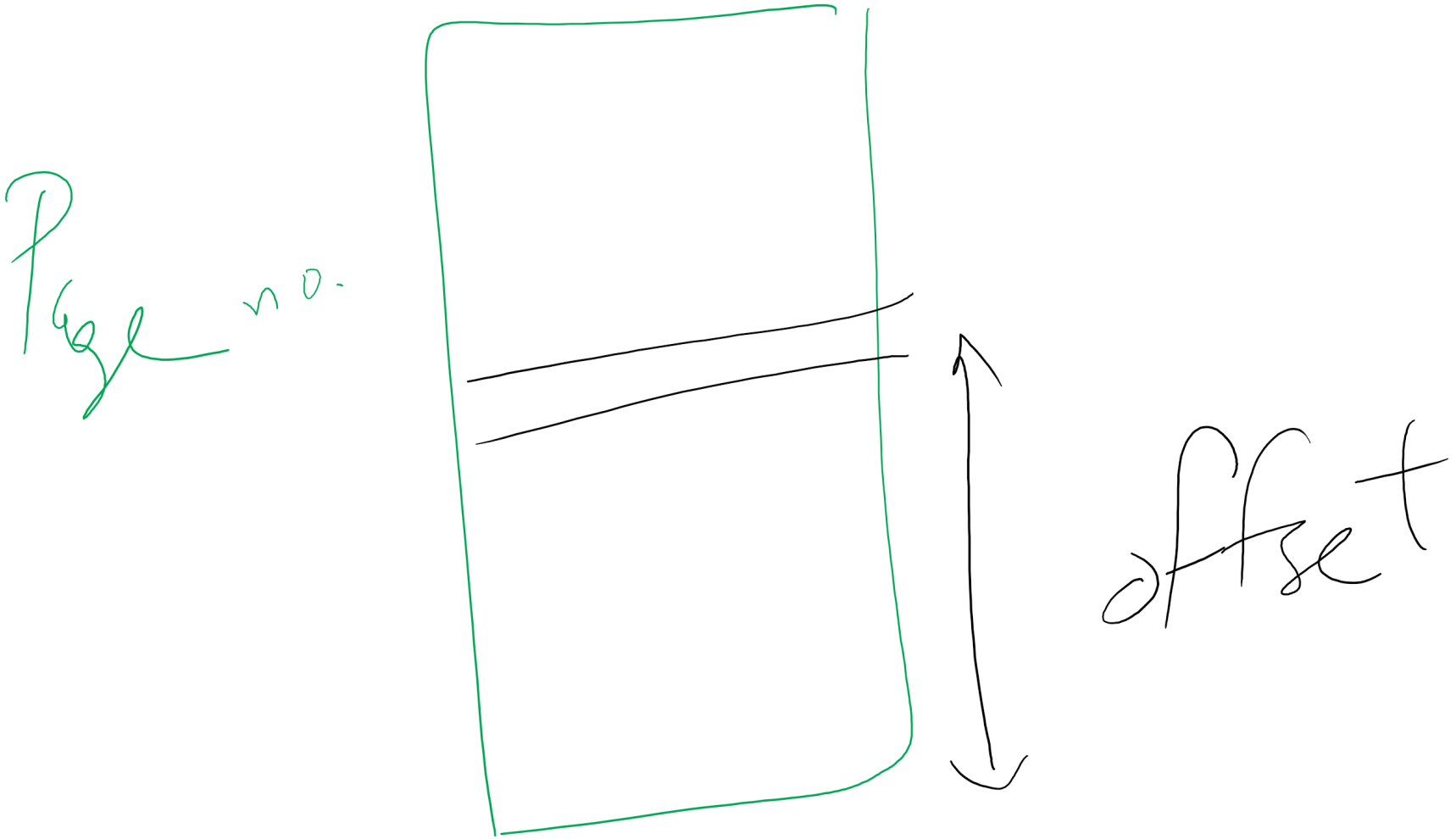


Page size = 4 KB = 2^{12} bytes

offset : 12 bits

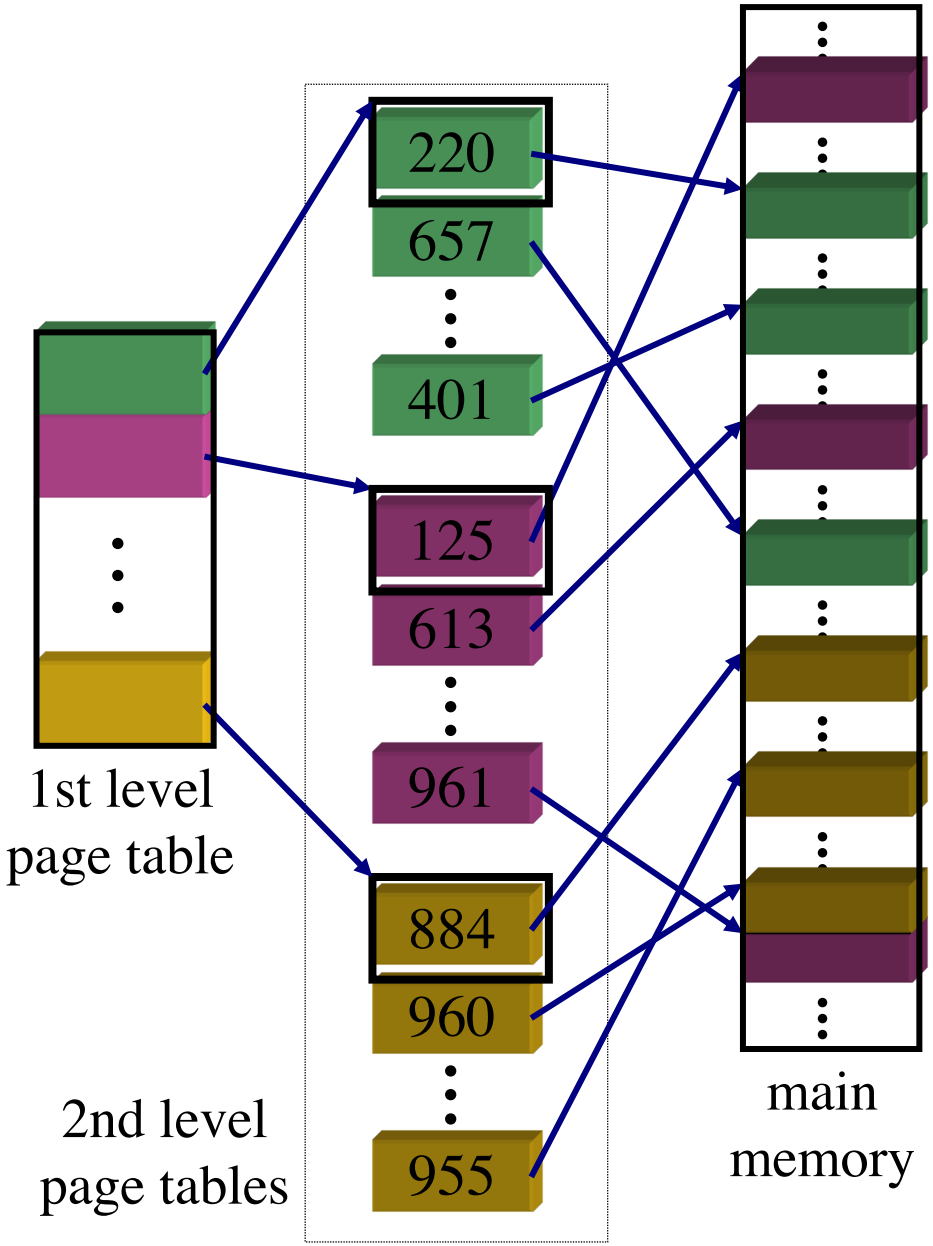
: 3 hexadecimal digits

Page number vs. Offset

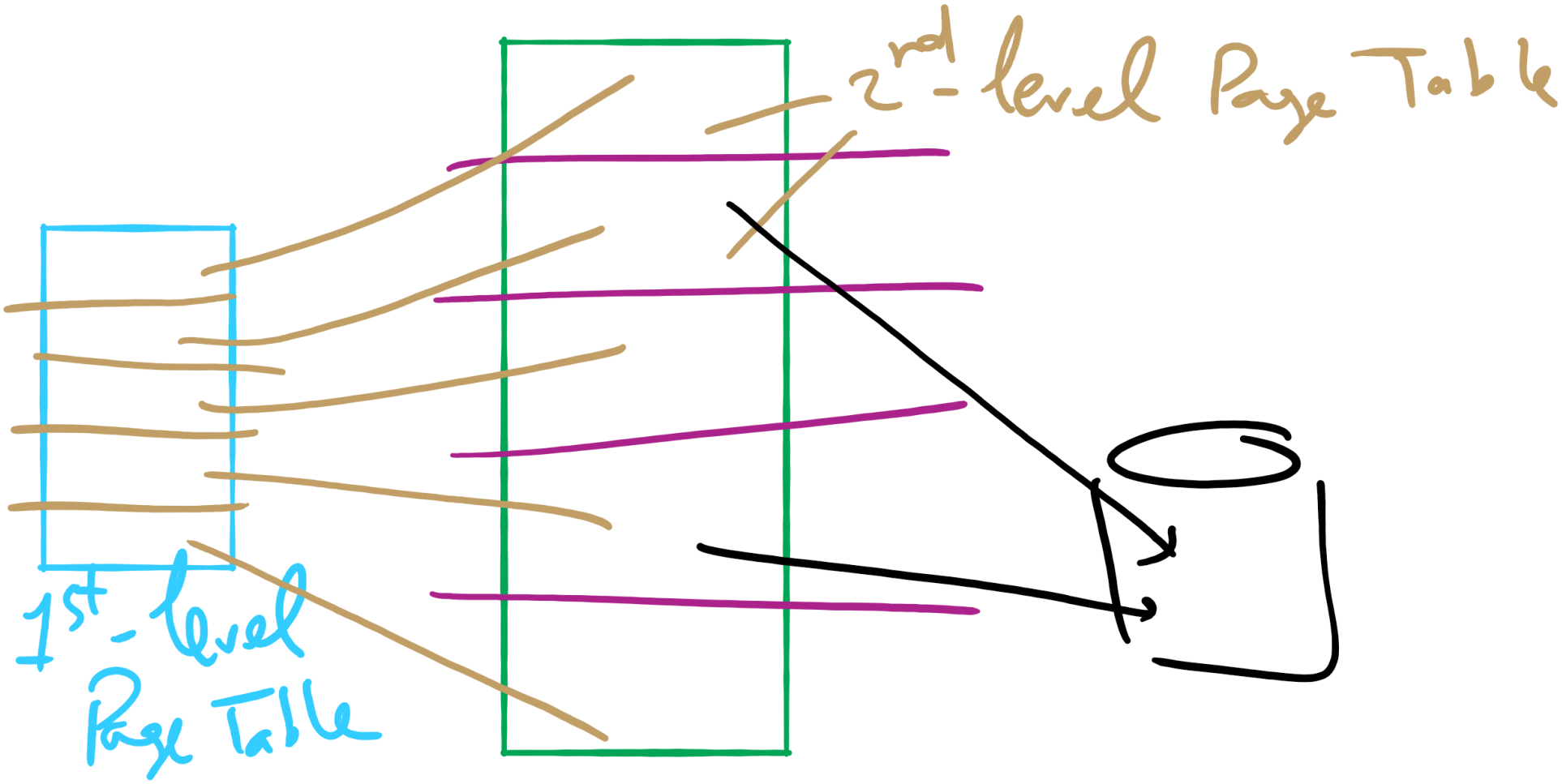


Solution 1: Two-level page tables

- Problem: page tables can be too large
  - 2^{32} bytes in 4KB pages need 1 million PTEs
- Solution: use multi-level page tables
  - "Page size" in first page table is large (megabytes)
  - PTE marked invalid in first page table needs no 2nd level page table
- 1st level page table has pointers to 2nd level page tables
- 2nd level page table has actual physical page numbers in it



Two-level Page Table

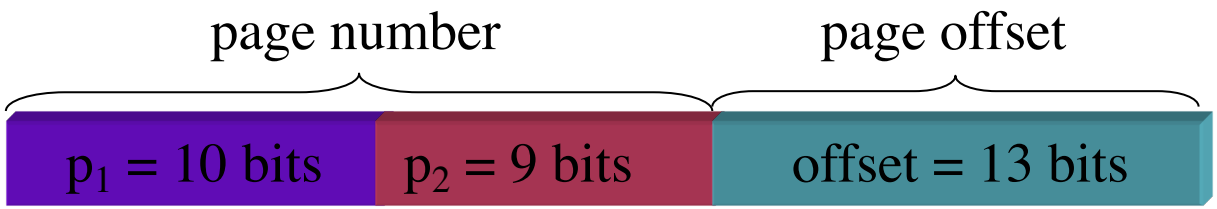


More on two-level page tables

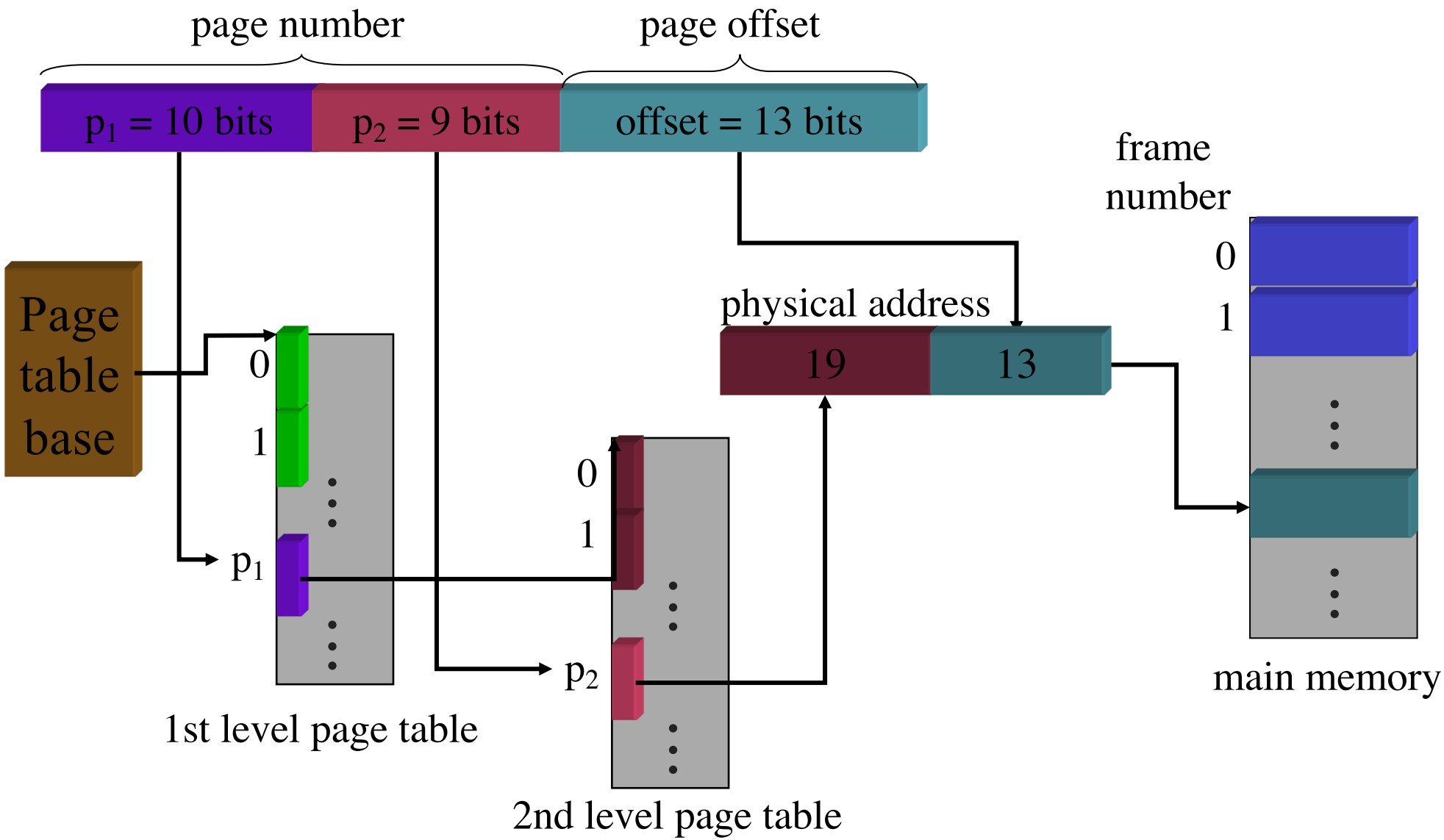
- Tradeoffs between 1st and 2nd level page table sizes
  - Total number of bits indexing 1st and 2nd level table is constant for a given page size and logical address length
  - Tradeoff between number of bits indexing 1st and number indexing 2nd level tables
    - More bits in 1st level: fine granularity at 2nd level
    - Fewer bits in 1st level: maybe less wasted space?
- All addresses in table are physical addresses
- Protection bits kept in 2nd level table

Two-level paging: example

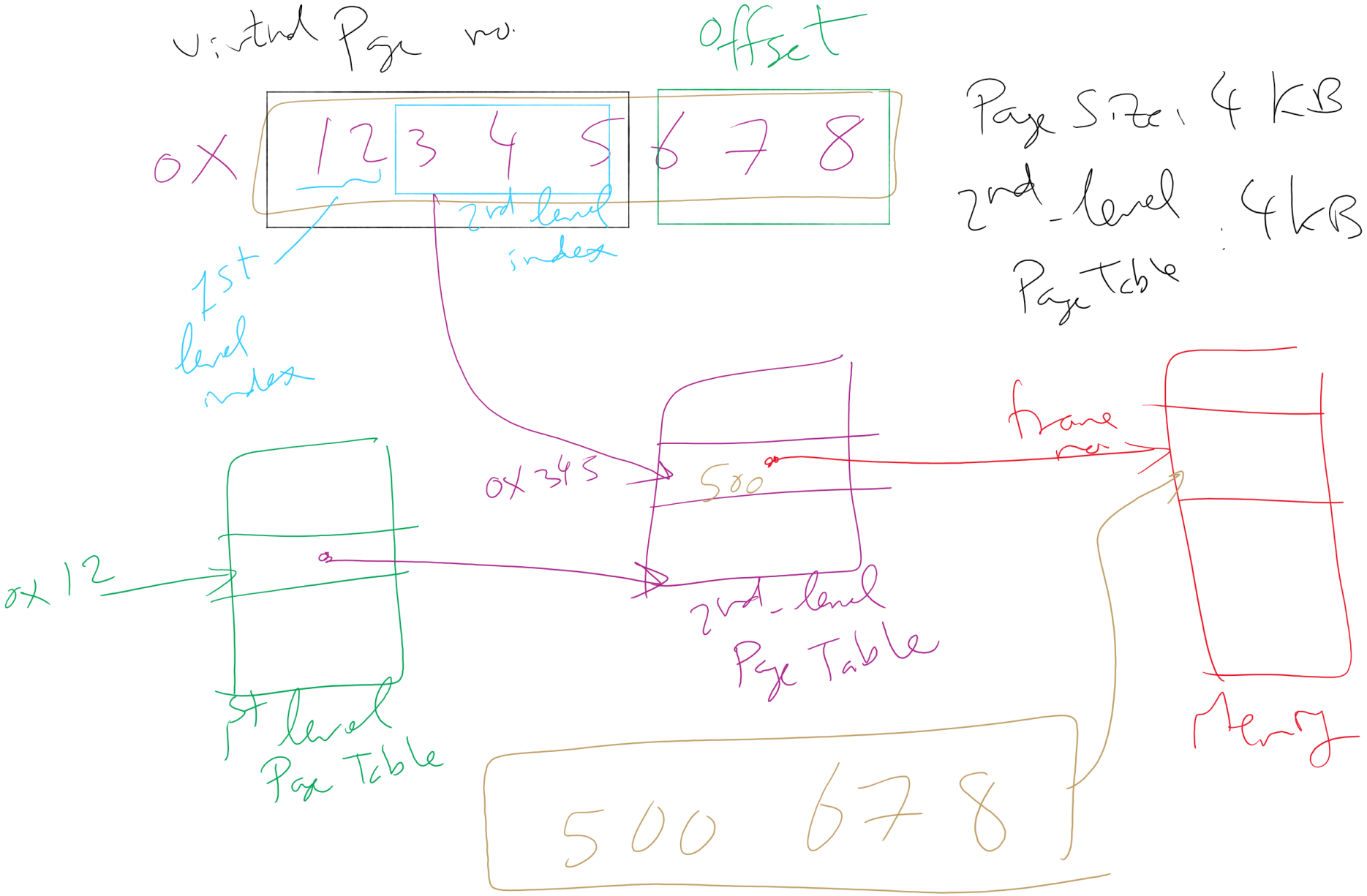
- System characteristics
  - 8 KB pages
  - 32-bit logical address divided into 13 bit page offset, 19 bit page number
- Page number divided into:
  - 10 bit page number
  - 9 bit page offset
- Logical address looks like this:
  - p_1 is an index into the 1st level page table
  - p_2 is an index into the 2nd level page table pointed to by p_1



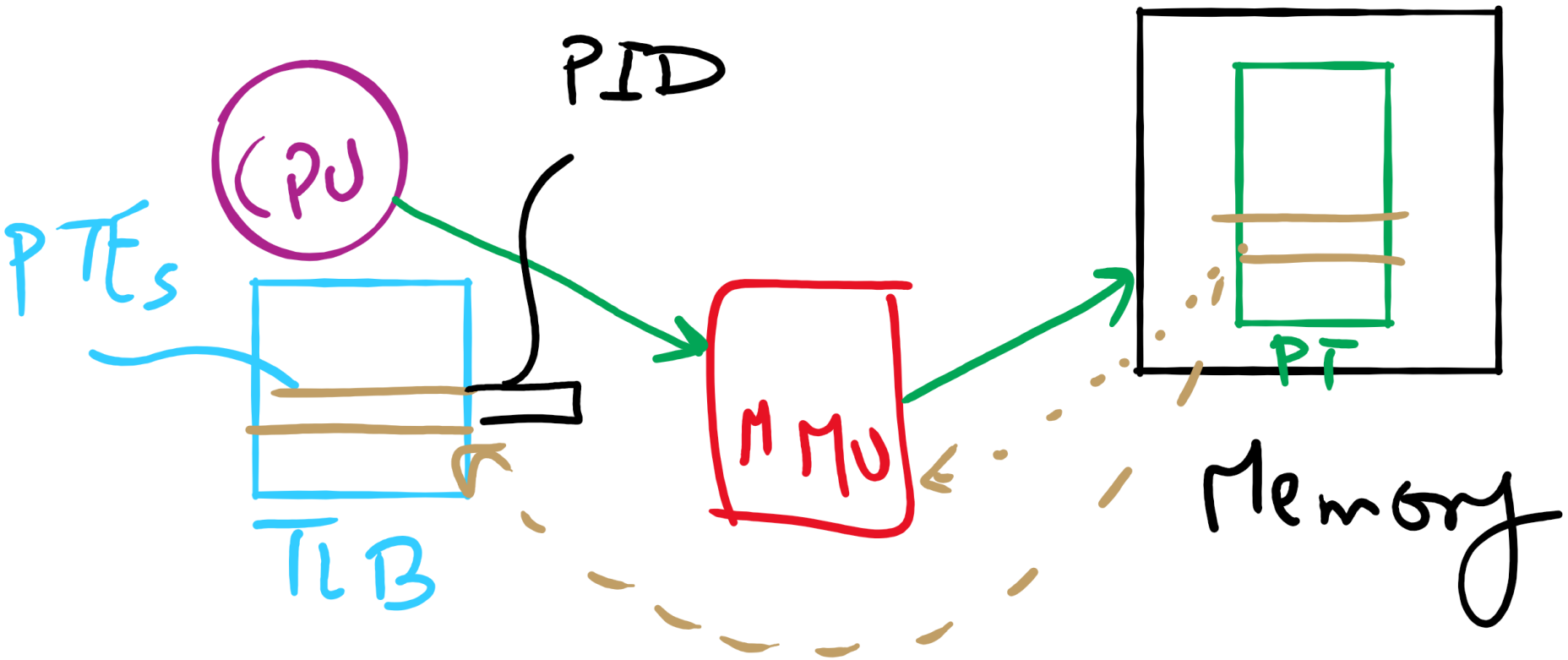
2-level address translation example



Address Translation: 2-level Page Table

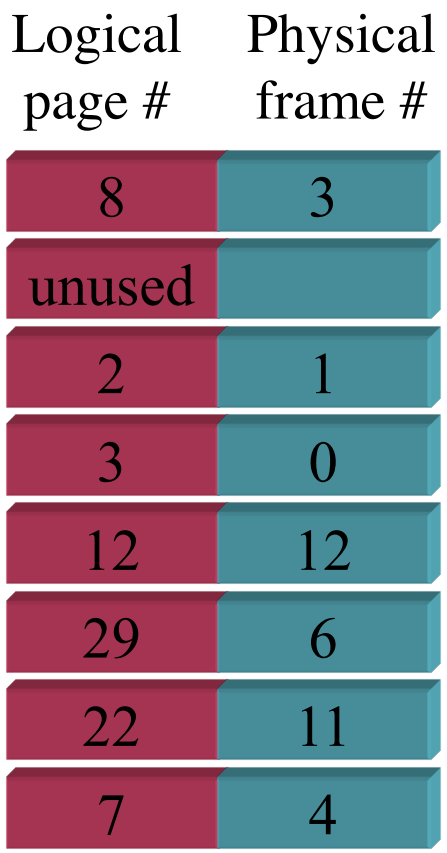


TLB



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

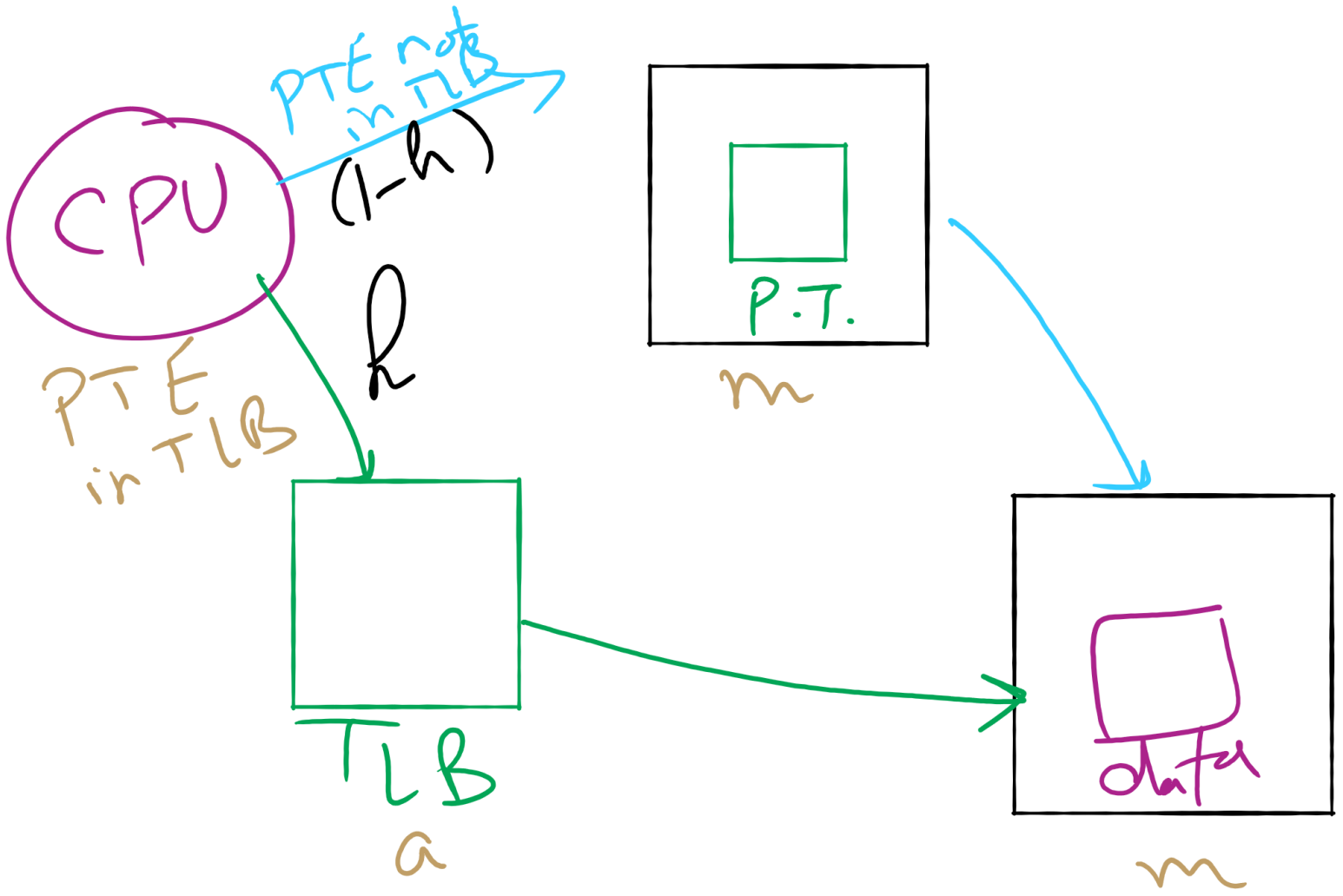
Handling TLB misses

- 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

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

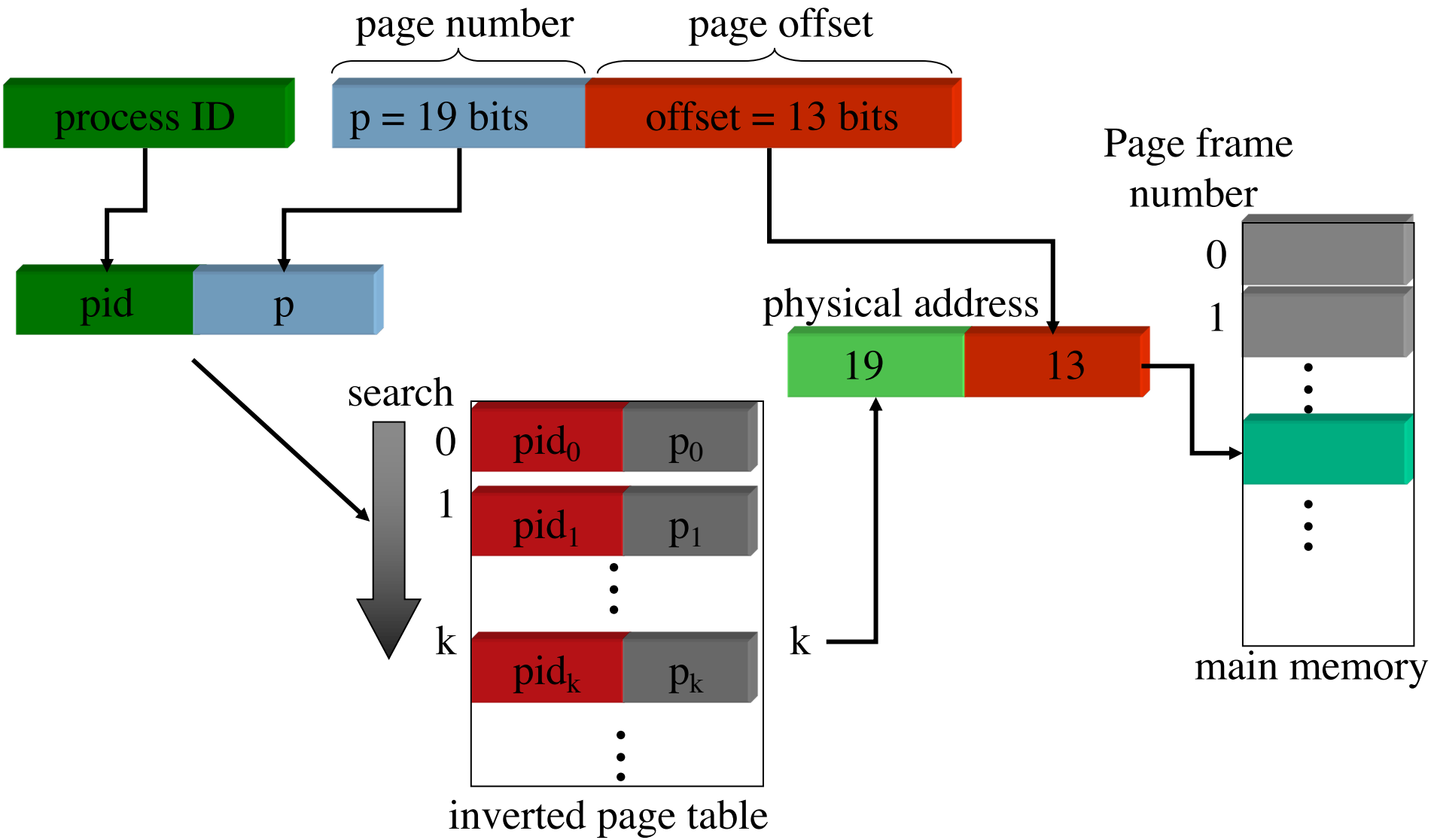
$$EAT = h(a+m) + (1-h)(a+m+m)$$

2-level Page Table → $2m$

Solution 2: Inverted page table

- Reduce page table size further: keep one entry for each frame in memory
- 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
- Improve performance by using more advanced hashing algorithms

Inverted page table architecture



Inverted Page Table Example