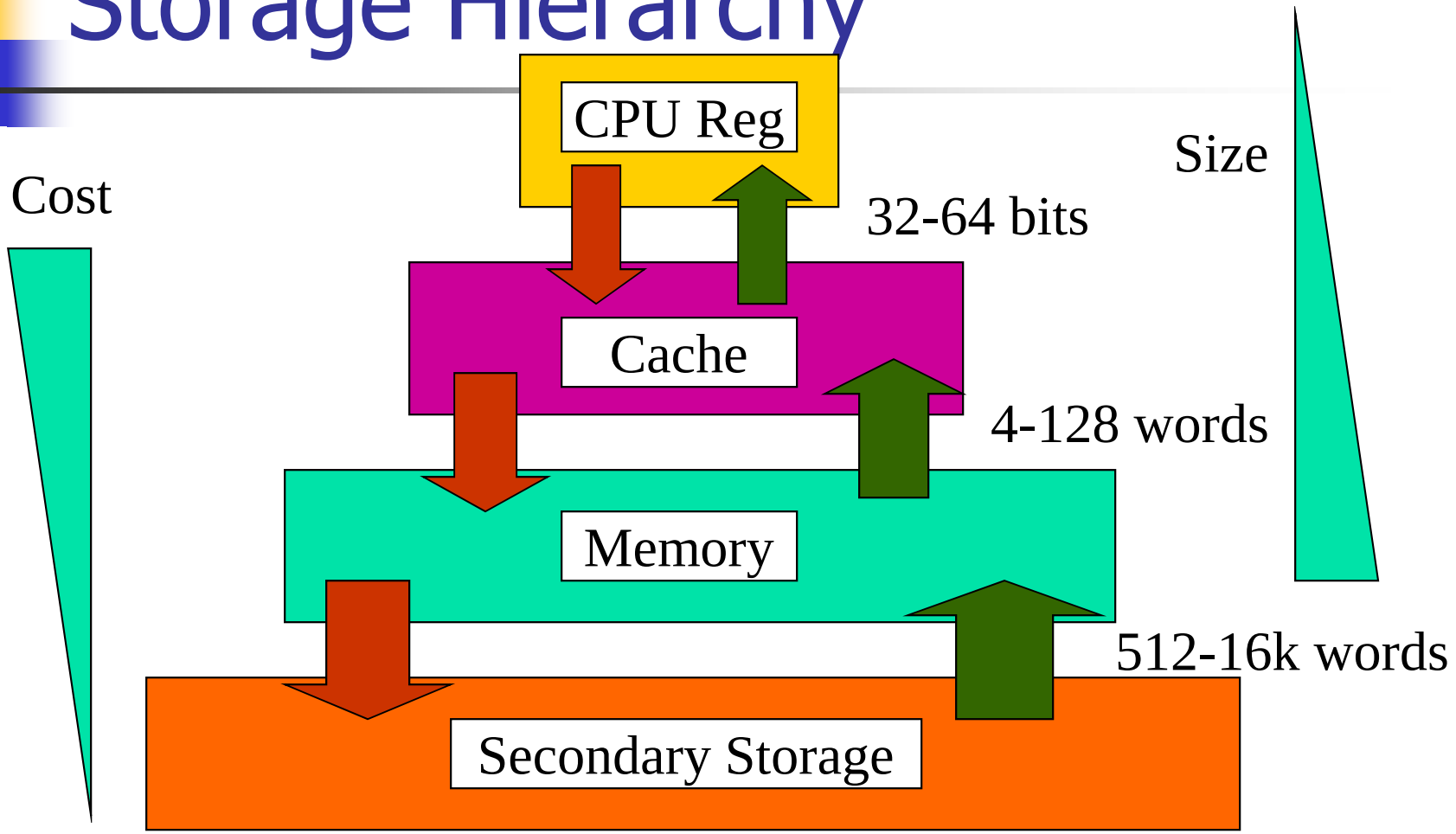
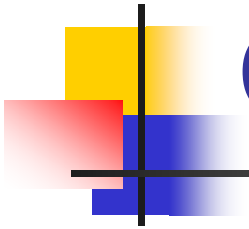


Virtual Memory Management

Storage Hierarchy

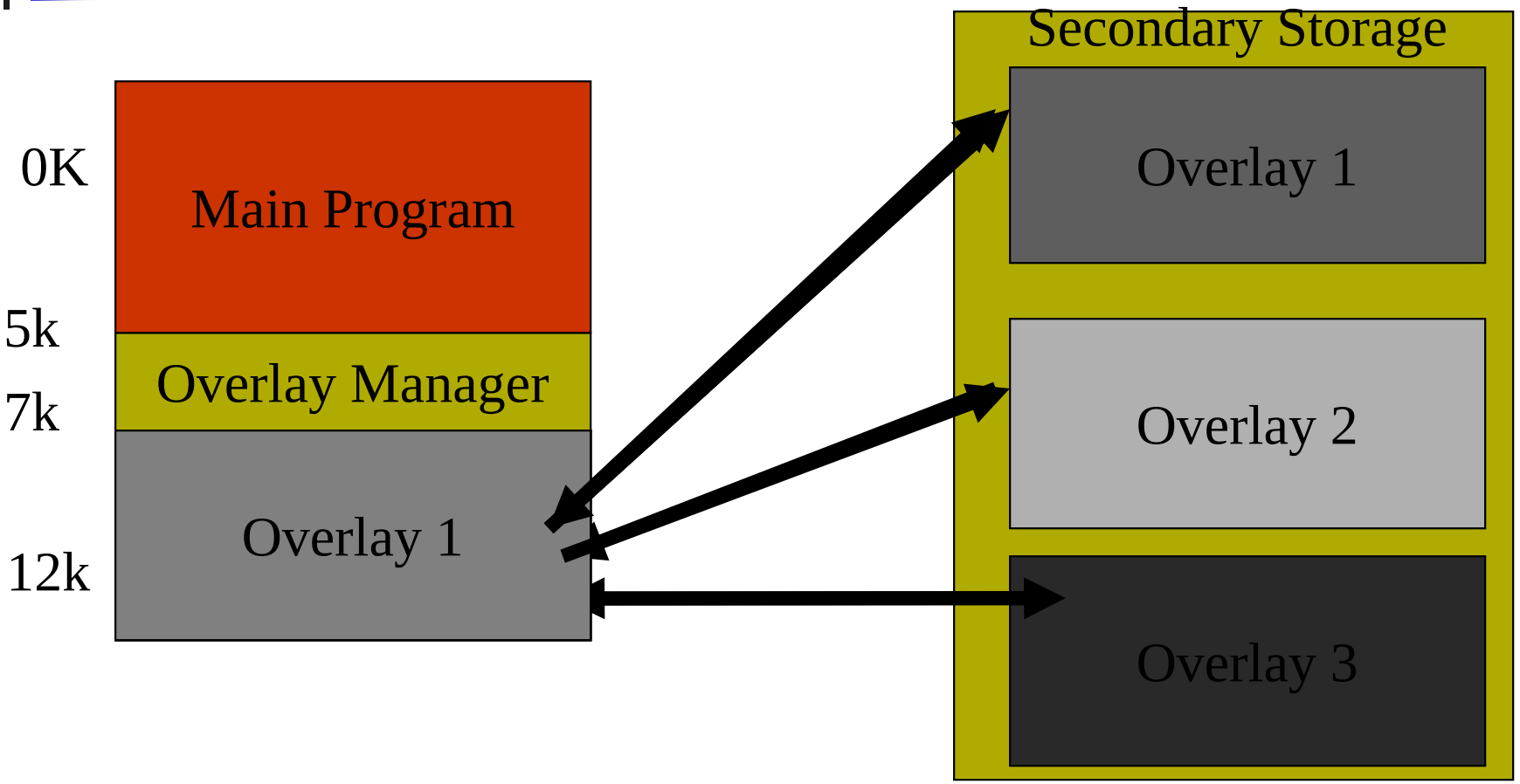




General Memory Problem

- We have a limited expensive *fast* resource (memory) and an *abundant* slow resource (disk)
- Make them look like an *abundant fast* resource.

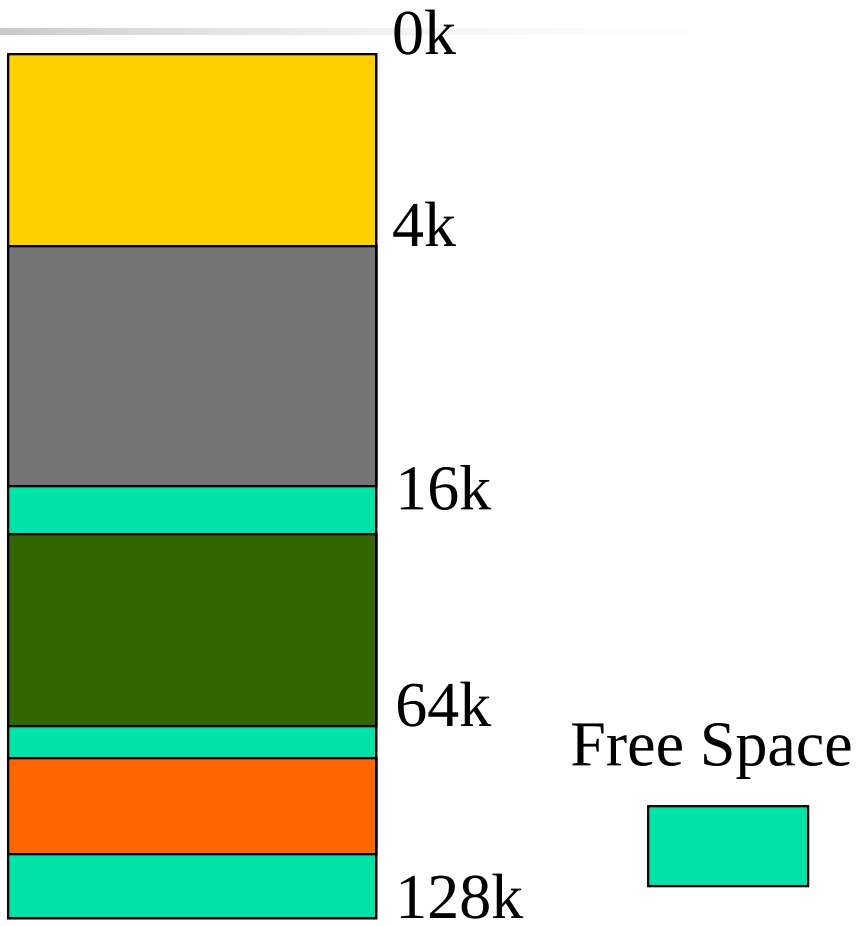
History: Overlays



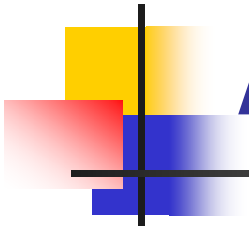
Used when process memory requirement exceeds the physical memory space

History: Multiprogramming with Fixed Partitions

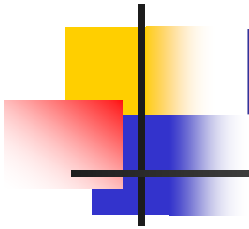
- Divide memory into *n* fixed (possibly unequal) partitions
- Problem:
  - Fragmentation



History: Fixed Partition Allocation



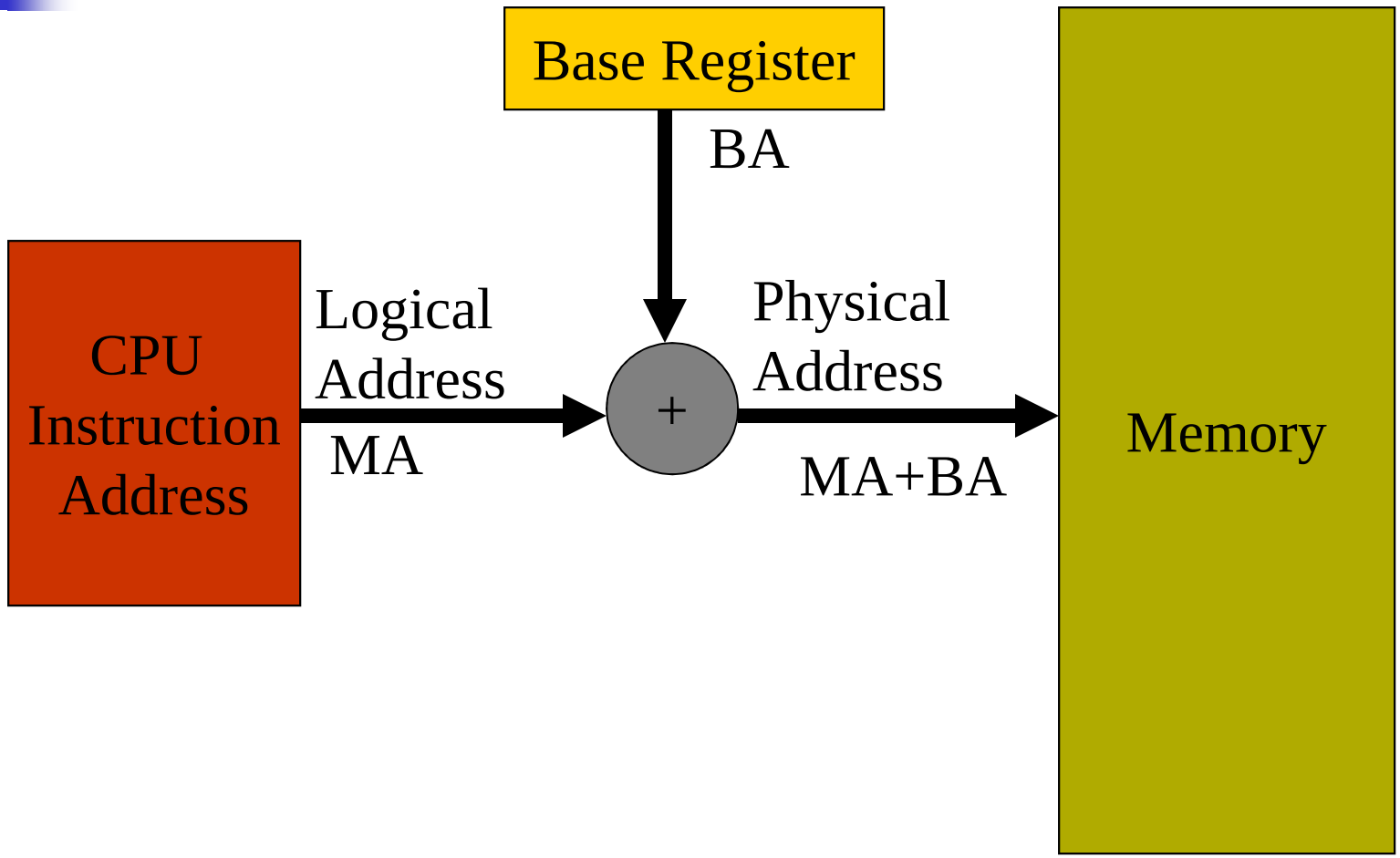
- Separate input queue for each partition
  - Sorting incoming jobs into separate queues
  - Inefficient utilization of memory
    - when the queue for a large partition is empty but the queue for a small partition is full. Small jobs have to wait to get into memory even though plenty of memory is free.
- One single input queue for all partitions.
  - Allocate a partition where the job fits in.
    - Best Fit, Worst Fit, First Fit



History: Relocation

- Correct starting address when a program should start in the memory
- Different jobs will run at different addresses
  - When a program is linked, the linker must know at what address the program will begin in memory.
- Logical addresses
  - Logical address space , range (0 to max)
  - Physical addresses, Physical address space range (R+0 to R+max) for base value R.
  - User program never sees the real physical addresses
- Relocation register
  - Mapping requires hardware with the base register

History: Relocation Register

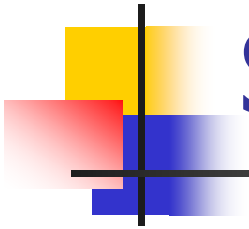


History: Variable Partition Allocation

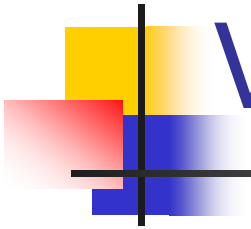


Memory wasted by External Fragmentation

History: Storage Placement Strategies



- **Best Fit**
  - Use the hole whose size is equal to the need, or if none is equal, the hole that is larger but closest in size.
  - Problem: Creates small holes that can't be used.
- **First Fit**
  - Use the first available hole whose size is sufficient to meet the need.
  - Problem: Creates average size holes.
- **Next Fit**.
  - Minor variation of first fit: search from the last hole used.
  - Problem: slightly worse performance than first fit.
- **Worst Fit**.
  - Use the largest available hole.
  - Problem: Gets rid of large holes making it difficult to run large programs.

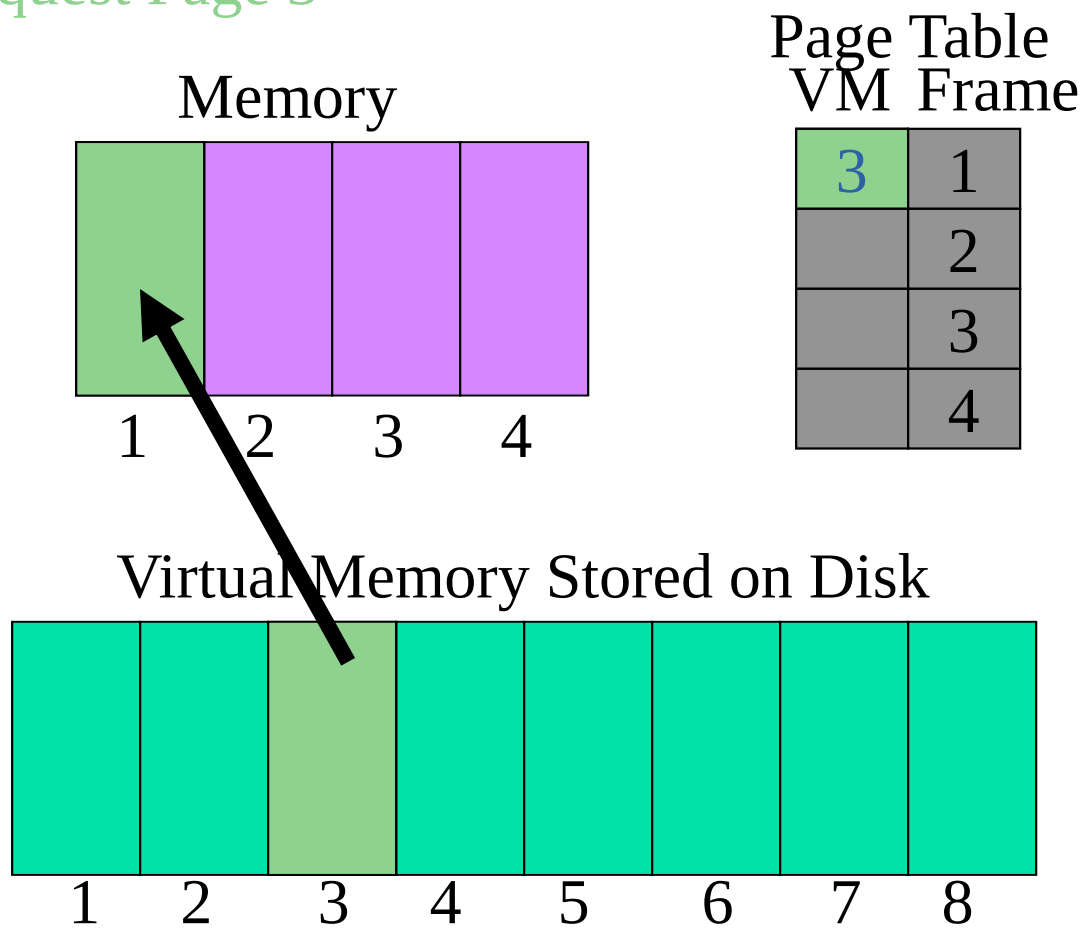


Virtual Memory

- Provide user with virtual memory that is as big as user needs
- Store virtual memory on disk
- Cache parts of virtual memory being used in real memory
- Load and store cached virtual memory without user program intervention

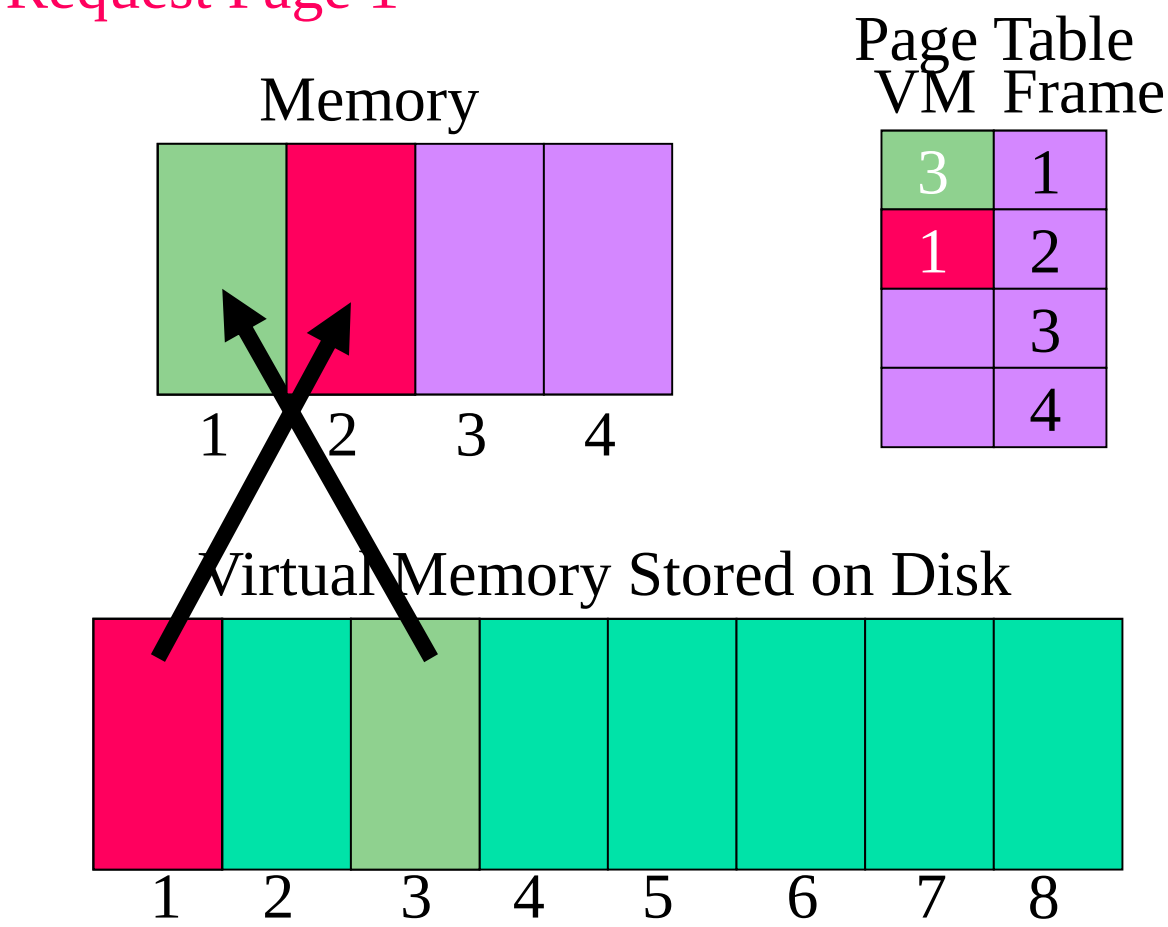
Paging

Request Page 3



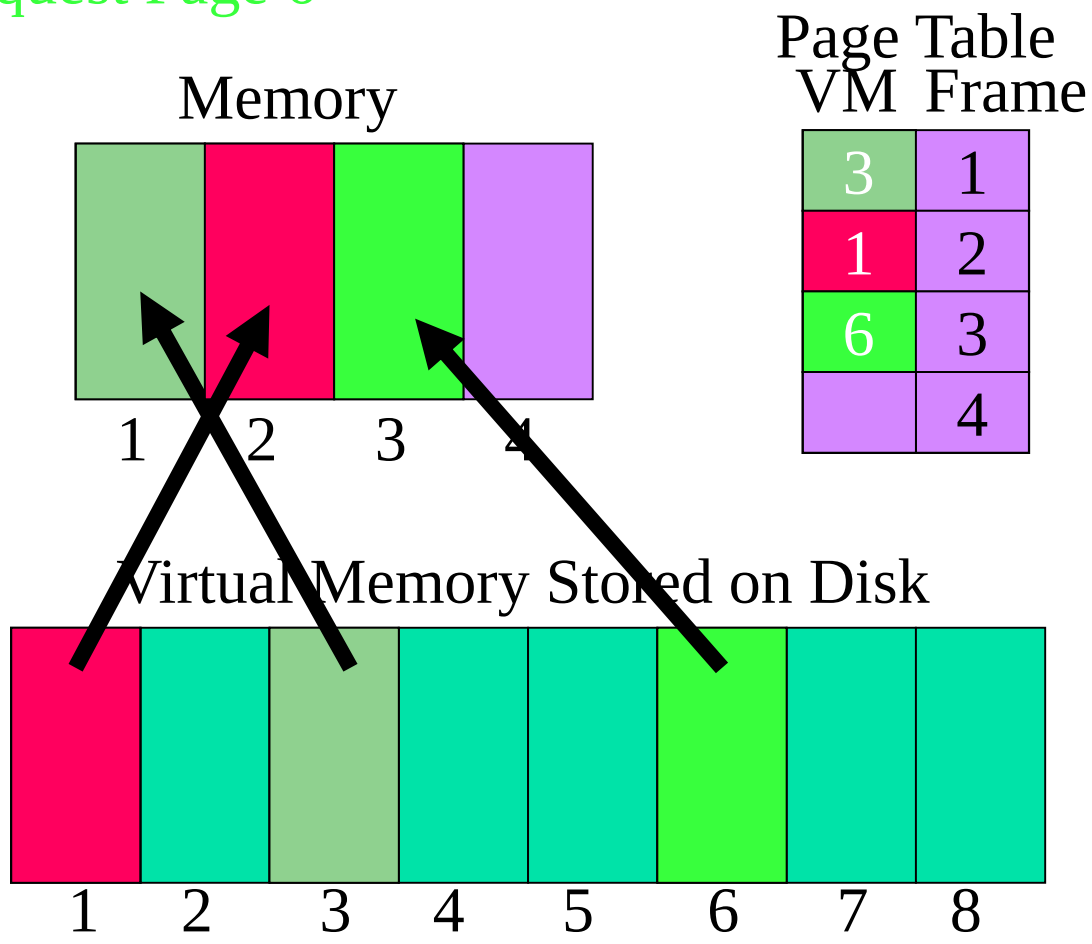
Paging

Request Page 1



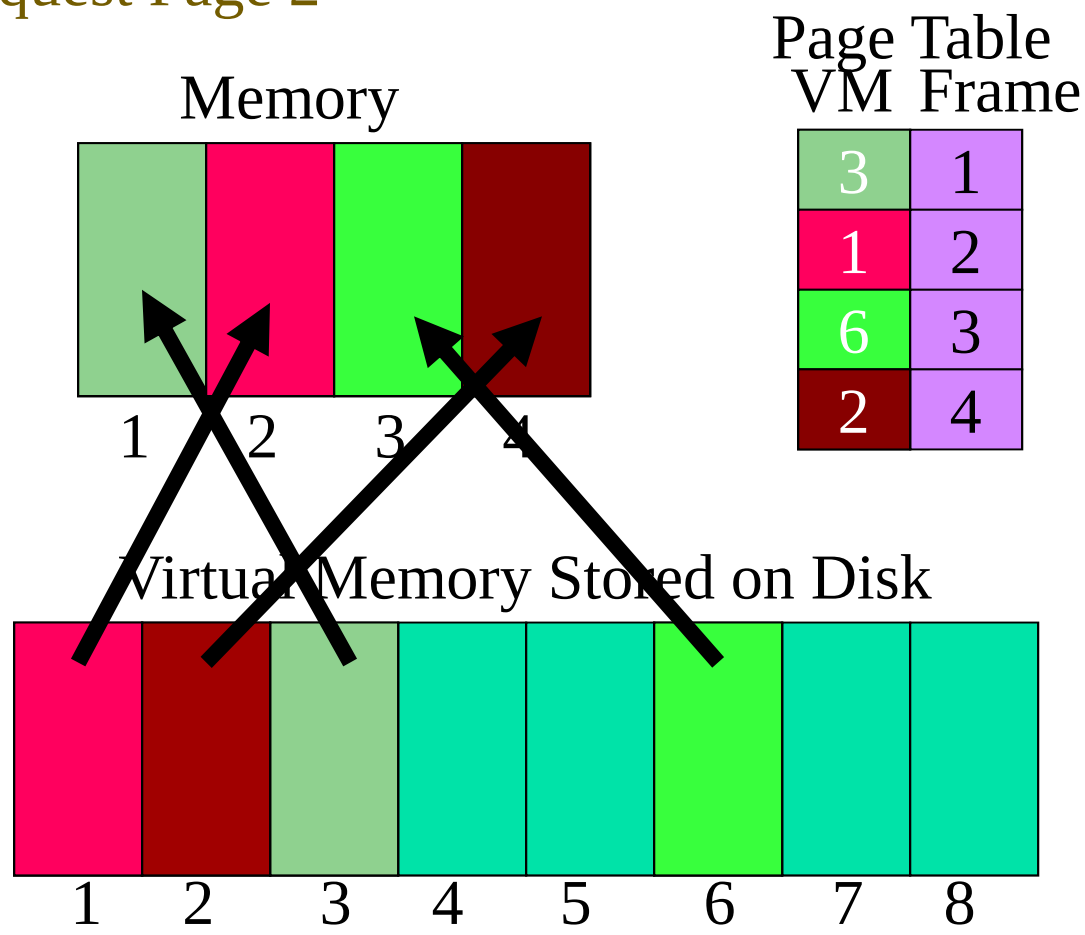
Paging

Request Page 6



Paging

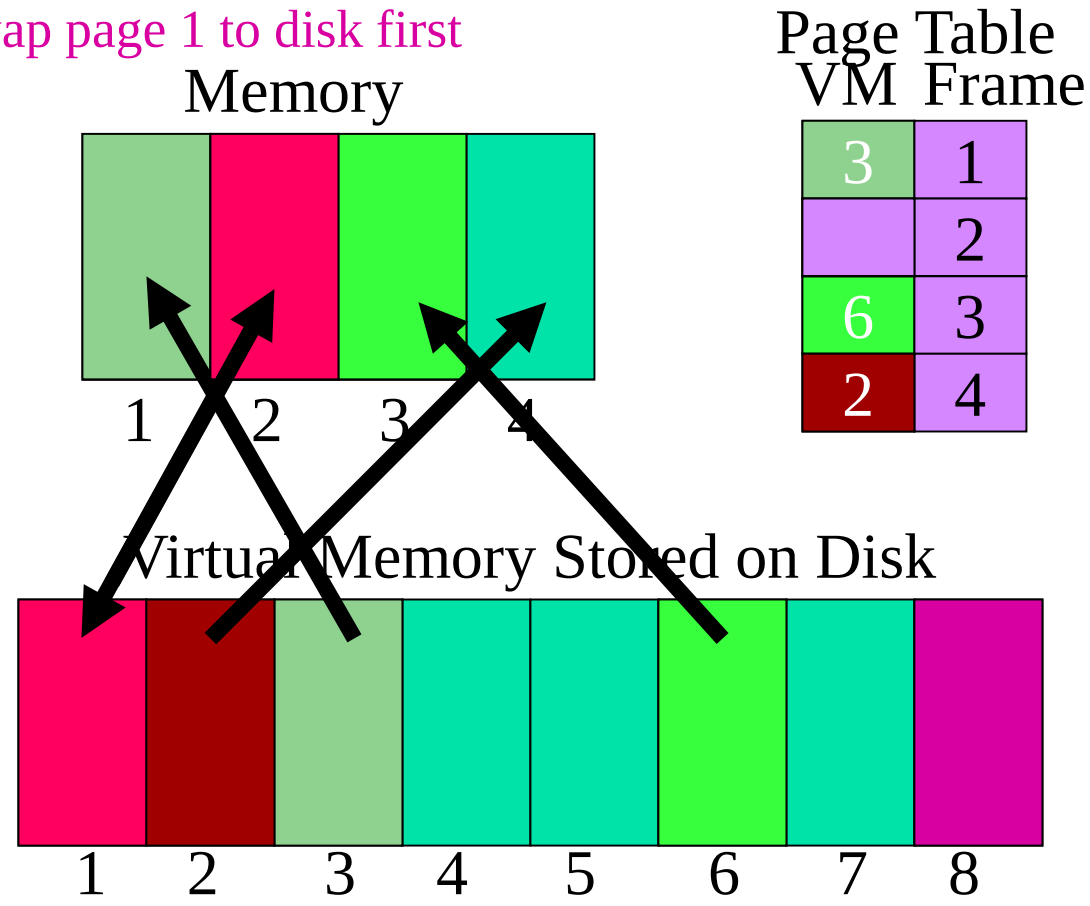
Request Page 2



Paging

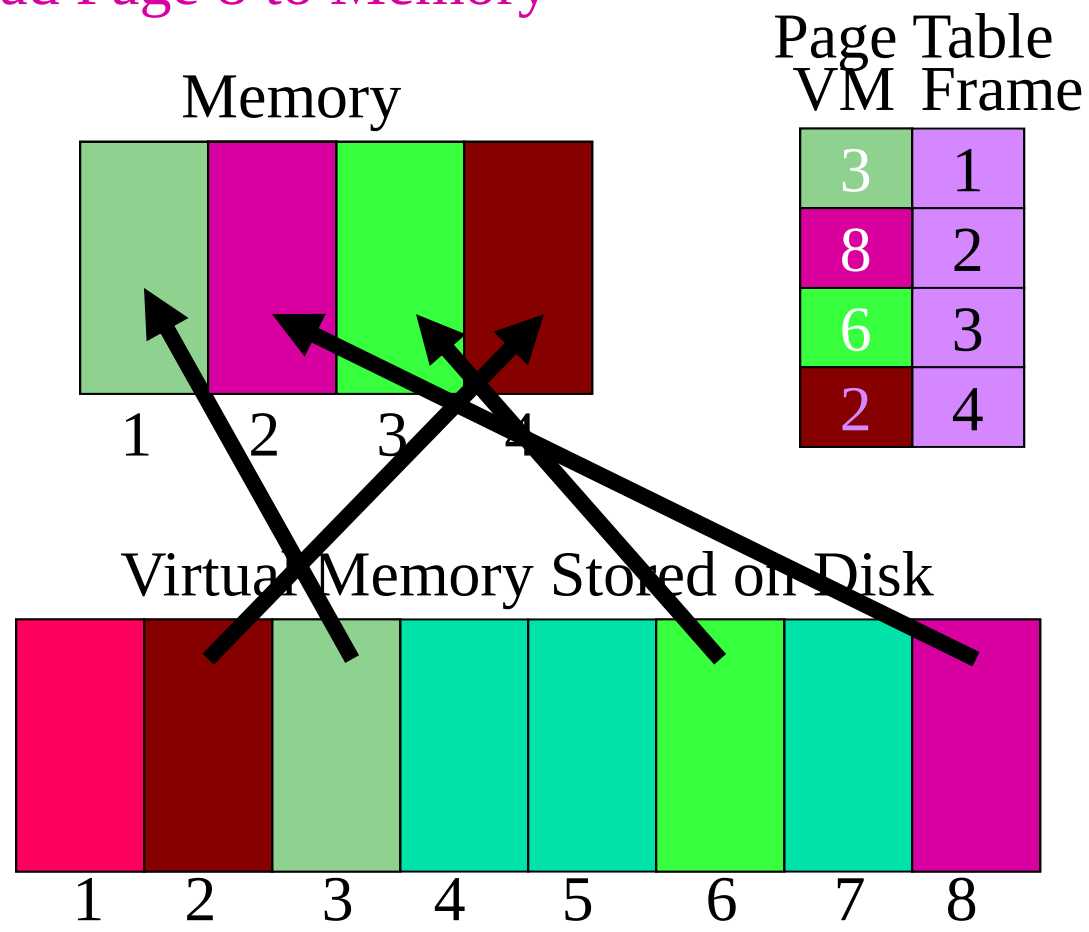
Request Page 8

Swap page 1 to disk first

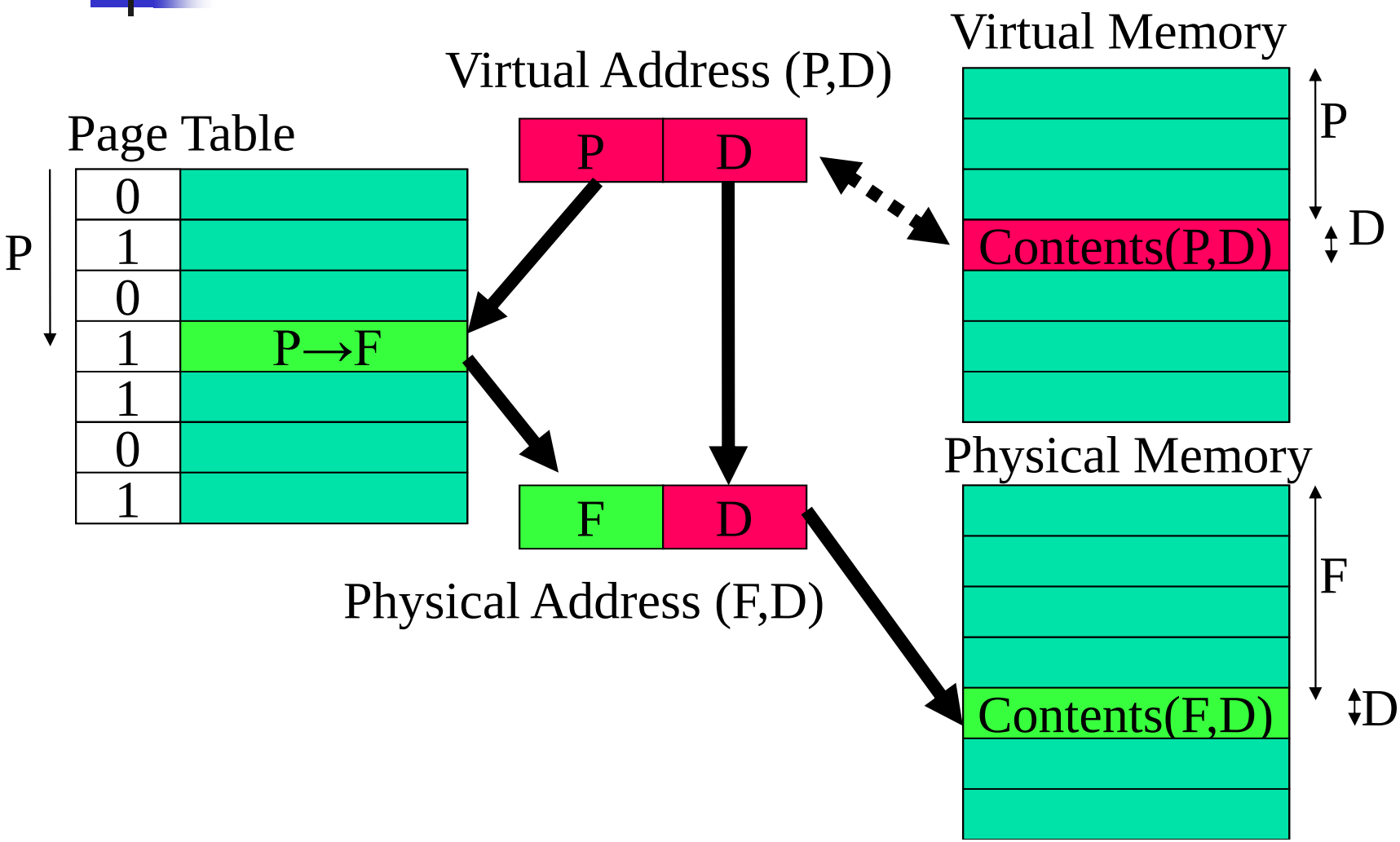


Paging

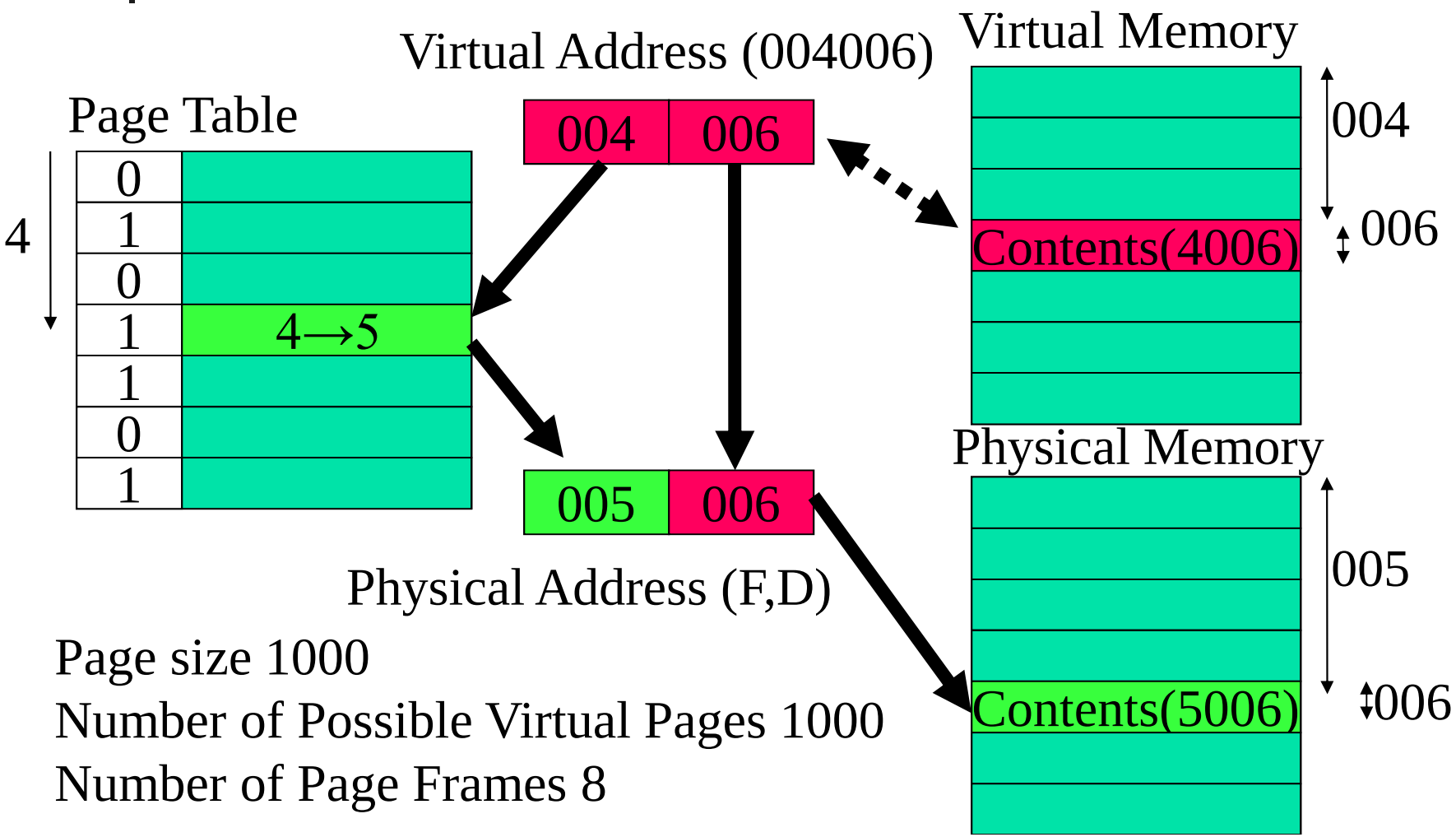
Load Page 8 to Memory



Page Mapping Hardware



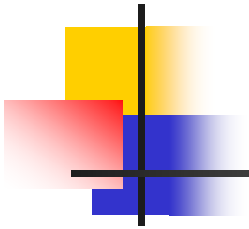
Page Mapping Hardware



Page size 1000

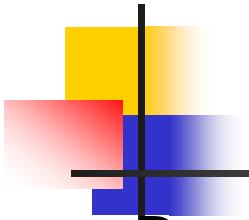
Number of Possible Virtual Pages 1000

Number of Page Frames 8



Page Fault

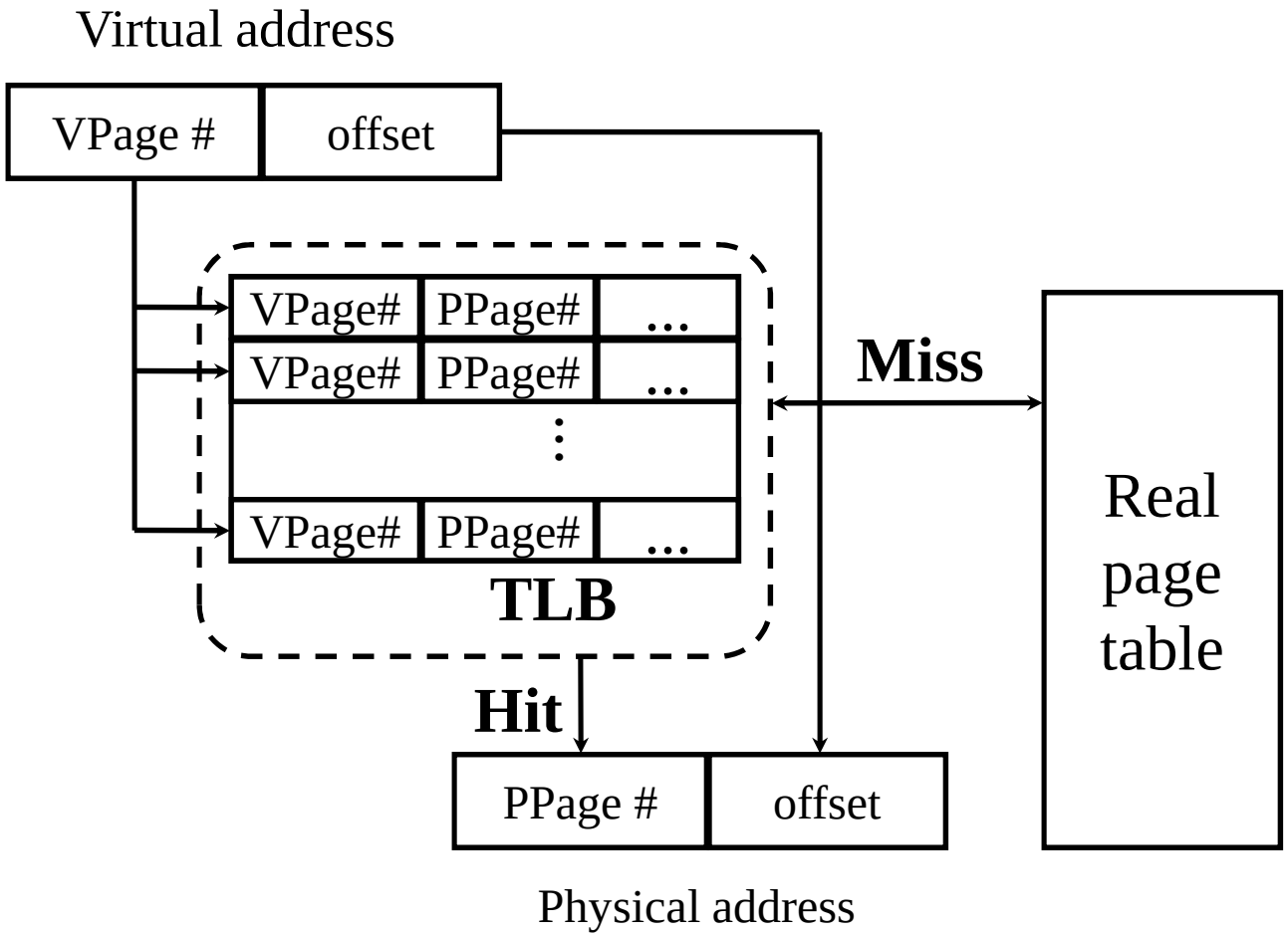
- Access a virtual page that is not mapped into any physical page
  - A fault is triggered by hardware
- Page fault handler (in OS's VM subsystem)
  - Find if there is any free physical page available
    - If no, evict some resident page to disk (swapping space)
  - Allocate a free physical page
  - Load the faulted virtual page to the prepared physical page
  - Modify the page table

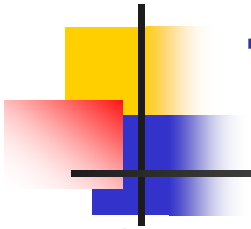


Paging Issues

- Page size is 2^n
  - usually 512 bytes, 1 KB, 2 KB, 4 KB, or 8 KB
  - E.g. 32 bit VM address may have 2^{20} (1 MB) pages with 4k (2^{12}) bytes per page
- Page table:
  - 2^{20} page entries take 2^{22} bytes (4 MB)
  - Must map into real memory
  - Page Table base register must be changed for context switch
- *No* external fragmentation; internal fragmentation on last page *only*

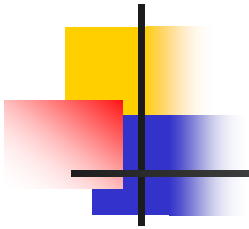
Optimization: Translation Lookaside Buffer (TLB)





TLB Function

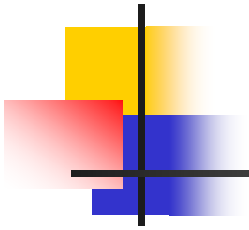
- If a virtual address is presented to MMU, the hardware checks TLB by comparing all entries simultaneously (in parallel).
- If match is valid, the page is taken from TLB without going through page table.
- If match is not valid
  - MMU detects miss and does a page table lookup.
  - It then evicts one page out of TLB and replaces it with the new entry, so that next time that page is found in TLB.



Issues

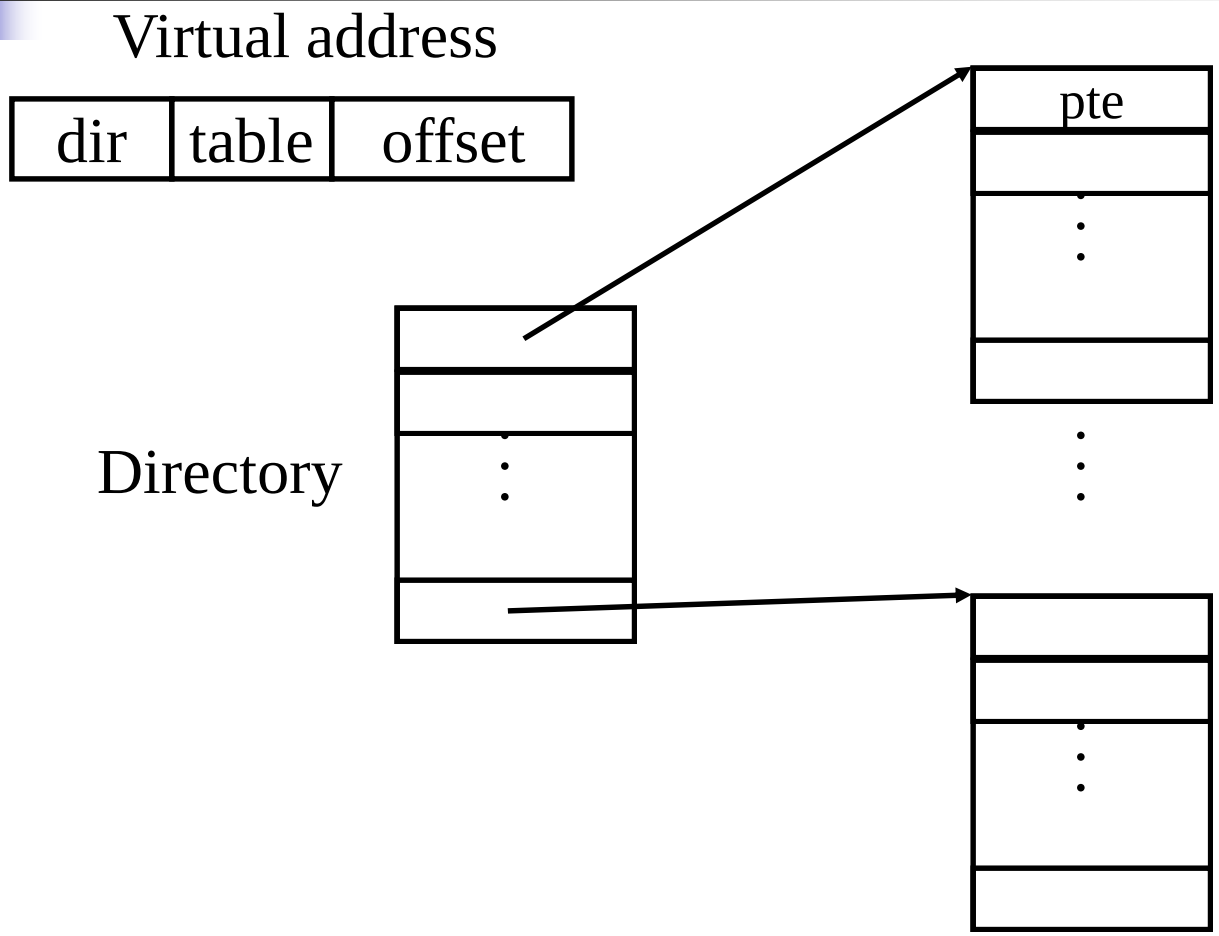
- What TLB entry to be replaced?
  - Random
  - Least Recently Used (LRU)
- What happens on a context switch?
  - Invalidate the entire TLB contents
- What happens when changing a page table entry?
  - Change the entry in memory
  - Invalidate the TLB entry

Effective Access Time



- TLB lookup time = σ time unit
- Memory cycle = m µs
- TLB Hit ratio = η
- Effective access time
  - $Eat = (m + \sigma)\,\eta + (2m + \sigma)(1 - \eta)$
  - $Eat = 2m + \sigma - m\,\eta$

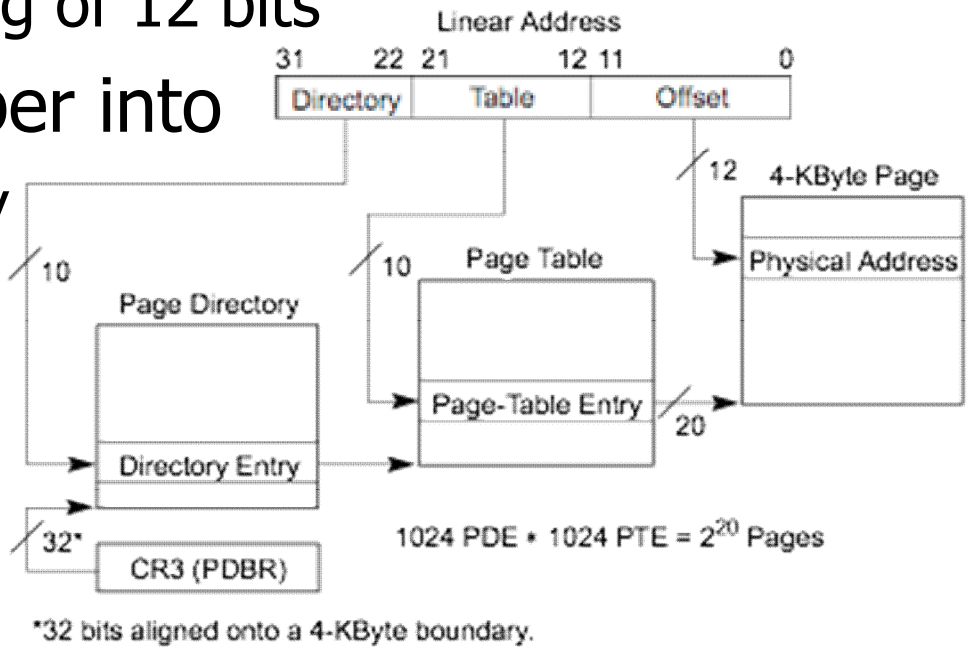
Multilevel Page Tables



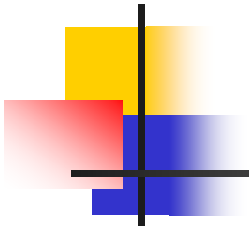
What does this buy us? Sparse address spaces and easier paging

Example Addressing on a Multilevel Page Table System

- A logical address (on 32-bit x86 with 4k page size) is divided into
  - A page number consisting of 20 bits
  - A page offset consisting of 12 bits
- Divide the page number into
  - A 10-bit page directory
  - A 10-bit page number

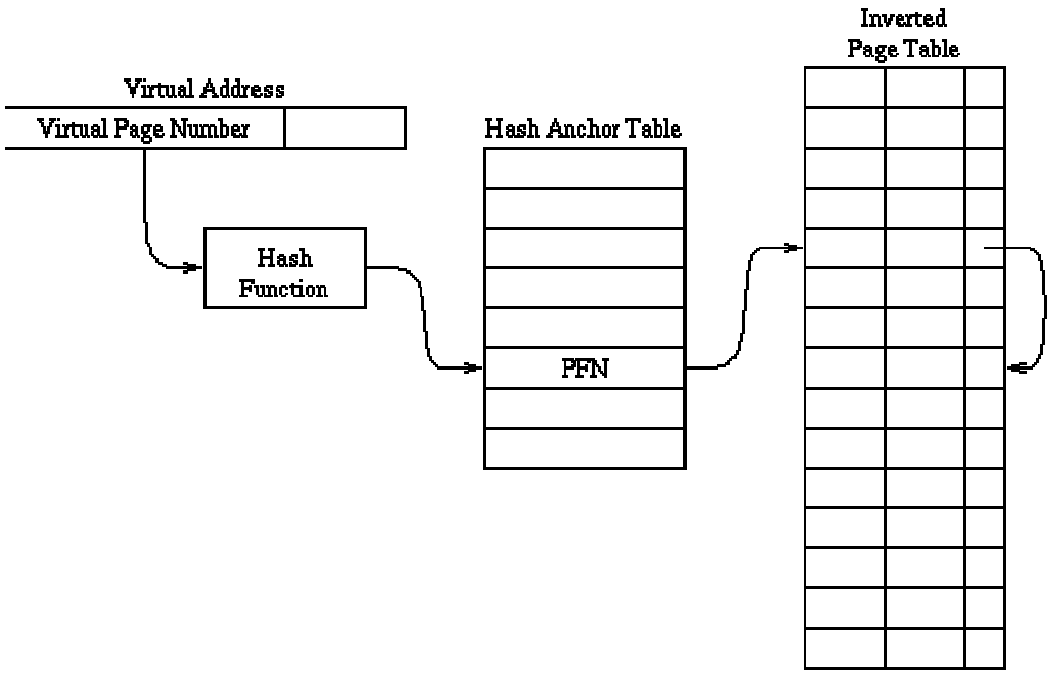


Multilevel Paging and Performance



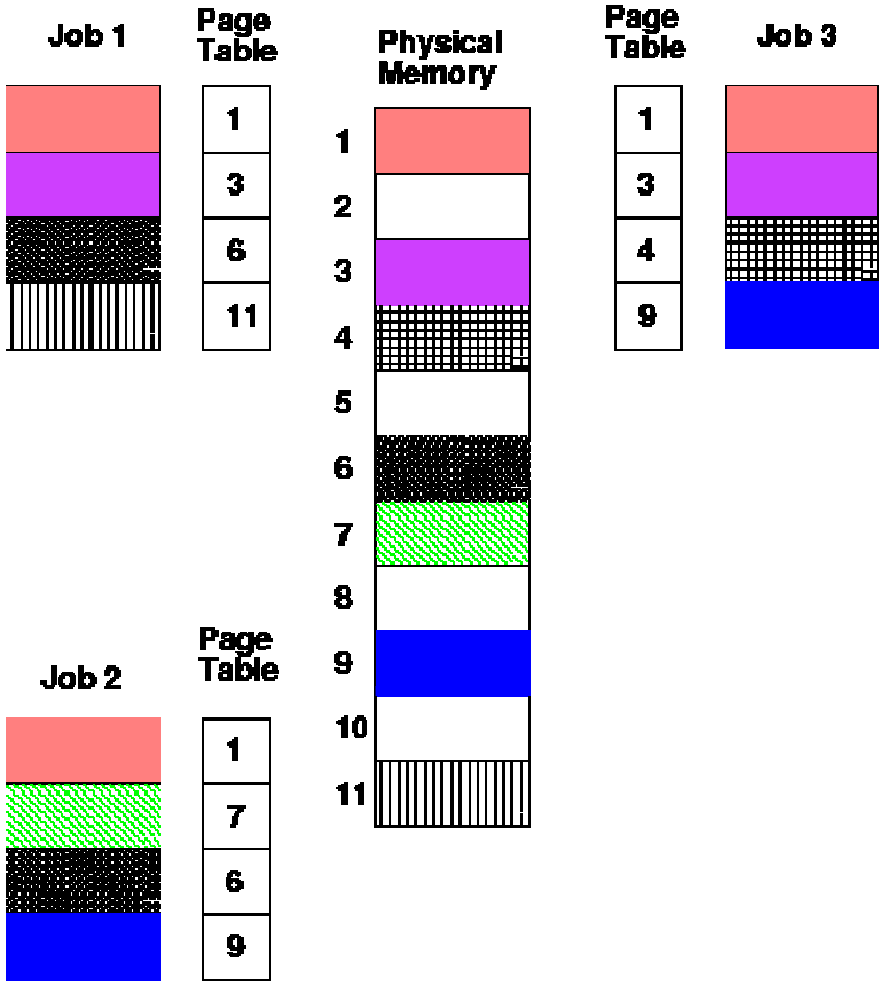
- Since each level is stored as a separate table in memory, converting a logical address to a physical one with a n-level page table may take n+1 memory accesses. Why?

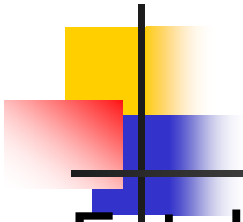
Inverted Page Tables



- Hash the process ID and virtual page number to get an index into the HAT.
- Look up a Physical Frame Number in the HAT.
- Look at the inverted page table entry, to see if it is the right process ID and virtual page number. If it is, you're done.
- If the PID or VPN does not match, follow the pointer to the next link in the hash chain. Again, if you get a match then you're done; if you don't, then you continue. Eventually, you will either get a match or you will find a pointer that is marked invalid. If you get a match, then you've got the translation; if you get the invalid pointer, then you have a miss.

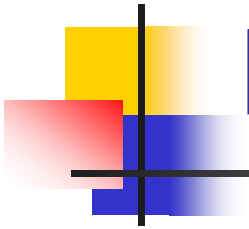
Shared Pages





Paging Policies

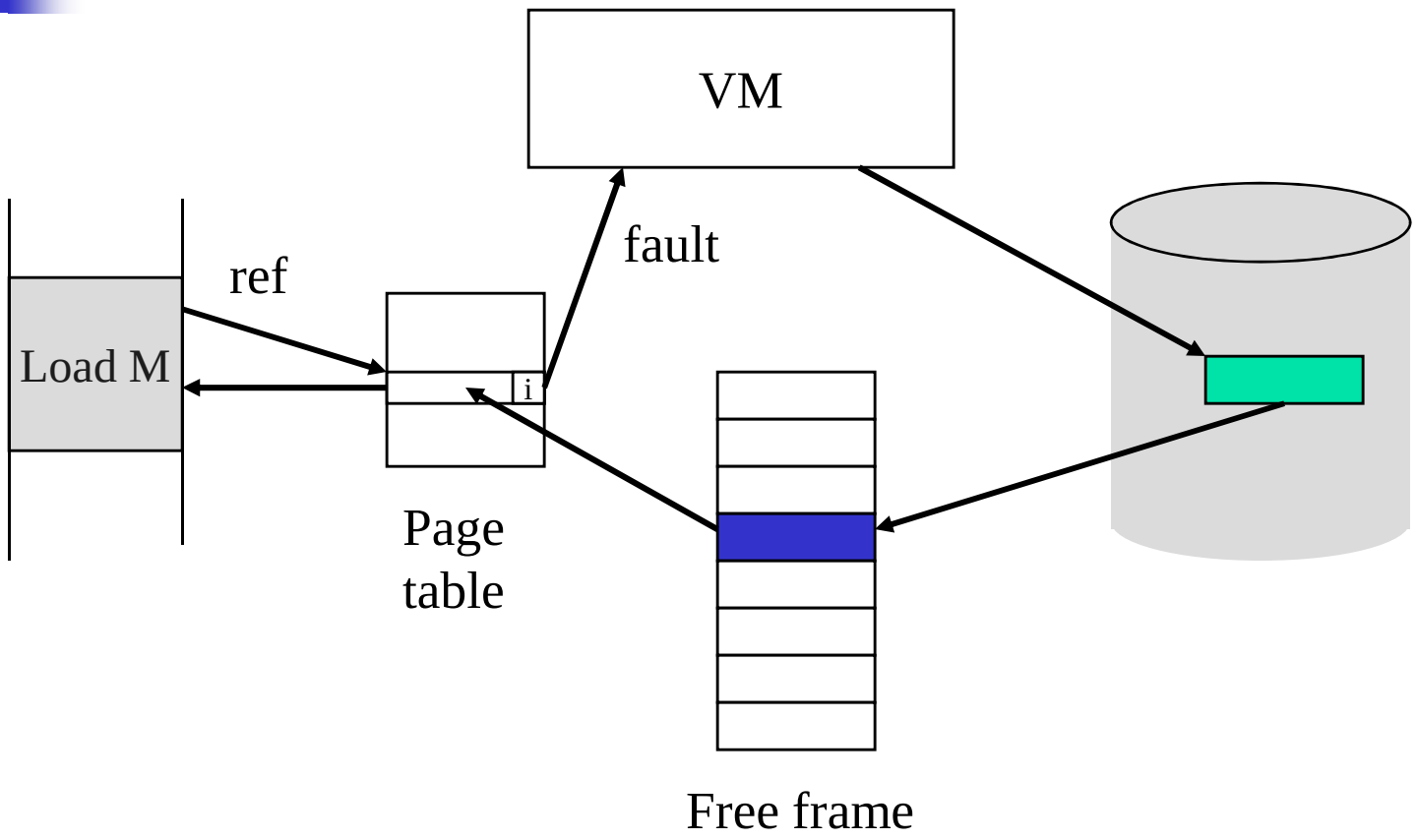
- Fetch Strategies
  - When should a page be brought into primary (main) memory from secondary (disk) storage.
- Placement Strategies
  - When a page is brought into primary storage, where is it to be put?
- Replacement Strategies
  - Which page in primary storage is to be removed when some other page or segment is to be brought in and there is not enough room.

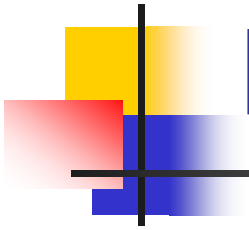


Fetch: Demand Paging

- Algorithm never brings a page into primary memory until its needed.
  1. Page fault
  2. Check if a valid virtual memory address. Kill job if not.
  3. Find a free page frame.
  4. Map address into disk block and fetch disk block into page frame. Suspend user process.
  5. When disk read finished, add vm mapping for page frame.
  6. Restart instruction.

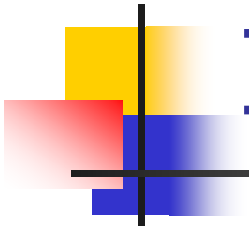
Demand Paging Example





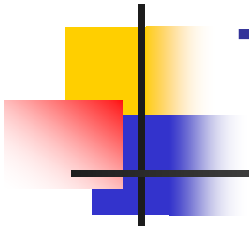
Page Replacement

1. Find location of page on disk
2. Find a free page frame
    1. If free page frame use it
    2. Otherwise, select a page frame using the page replacement algorithm
    3. Write the selected page to the disk and update any necessary tables
3. Read the requested page from the disk.
4. Restart instruction.



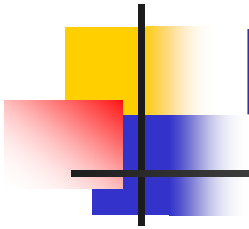
Issue: Eviction

- Hopefully, kick out a less-useful page
  - Dirty pages require writing, clean pages don't
    - Hardware has a dirty bit for each page frame indicating this page has been updated or not
  - Where do you write? To "swap space" on disk.
- Goal: kick out the page that's least useful
- Problem: how do you determine utility?
  - Heuristic: temporal locality exists
  - Kick out pages that aren't likely to be used again



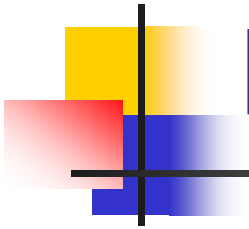
Terminology

- **Reference string**: the memory reference sequence generated by a program.
- **Paging** – moving pages to (from) disk
- **Optimal** – the best (theoretical) strategy
- **Eviction** – throwing something out
- **Pollution** – bringing in useless pages/lines



Page Replacement Strategies

- **The Principle of Optimality**
  - Replace the page that will not be used the most time in the future.
- **Random page replacement**
  - Choose a page randomly
- **FIFO - First in First Out**
  - Replace the page that has been in primary memory the longest
- **LRU - Least Recently Used**
  - Replace the page that has not been used for the longest time
- **LFU - Least Frequently Used**
  - Replace the page that is used least often
- **Second Chance**
  - An approximation to LRU.



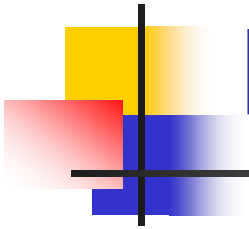
Principle of Optimality

- Description:
  - Assume that each page can be labeled with the number of instructions that will be executed before that page is first referenced, i.e., we would know the future reference string for a program.
  - Then the optimal page algorithm would choose the page with the highest label to be removed from the memory.
- Impractical because it needs to know future references

Optimal Example

12 references,
7 faults

Page Refs	3 Page Frames			
	Fault?	Page Contents		
A	yes	A		
B	yes	B	A	
C	yes	C	B	A
D	yes	D	B	A
A	no	D	B	A
B	no	D	B	A
E	yes	E	B	A
A	no	E	B	A
B	no	E	B	A
C	yes	C	E	B
D	yes	D	C	E
E	no	D	C	E

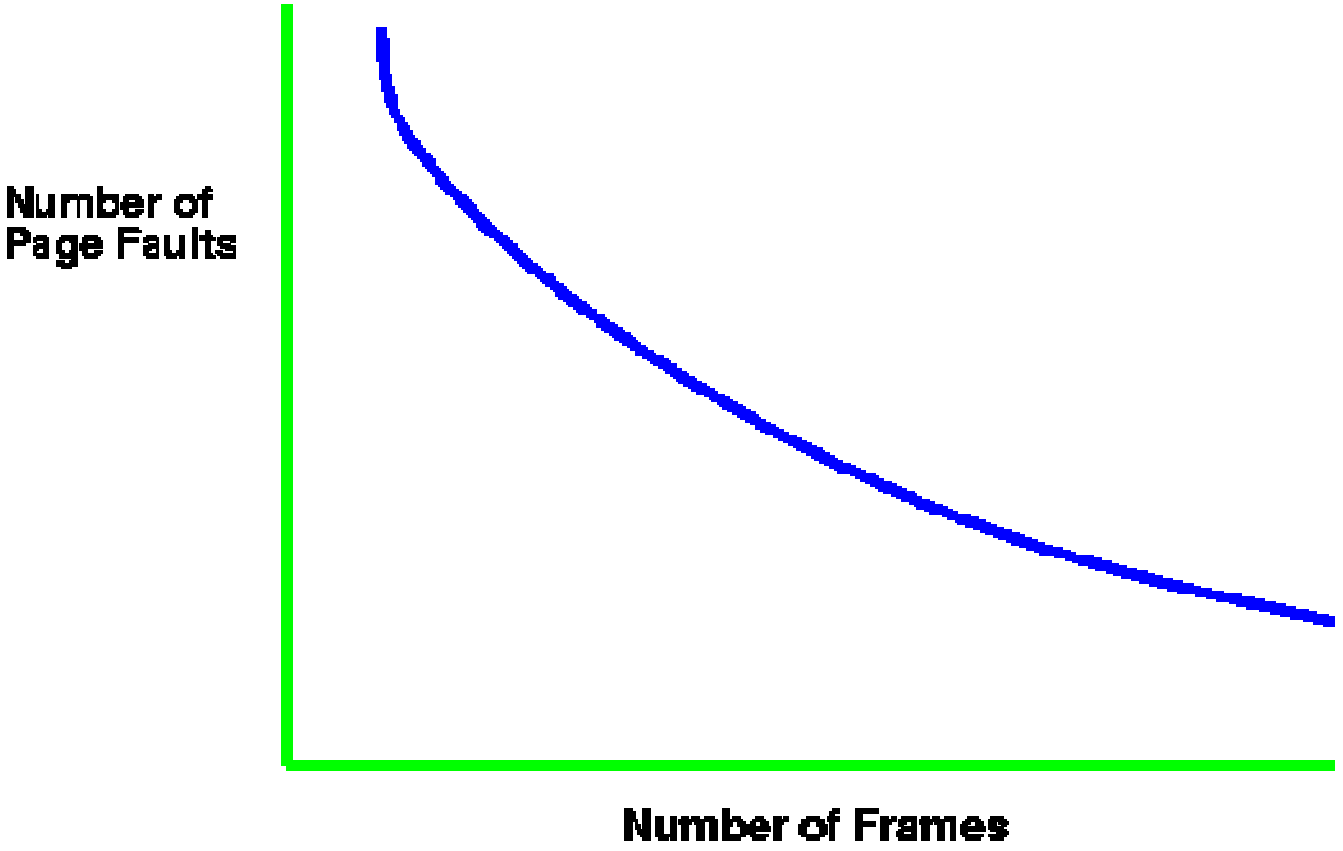


FIFO

12 references,
9 faults

Page Refs	3 Page Frames			
	Fault?	Page Contents		
A	yes	A		
B	yes	B	A	
C	yes	C	B	A
D	yes	D	C	B
A	yes	A	D	C
B	yes	B	A	D
E	yes	E	B	A
A	no	E	B	A
B	no	E	B	A
C	yes	C	E	B
D	yes	D	C	E
E	no	D	C	E

Average Paging Behavior with Increasing Page Frames



Belady's Anomaly (for FIFO)

FIFO with 4 physical pages

12 references, 10 faults

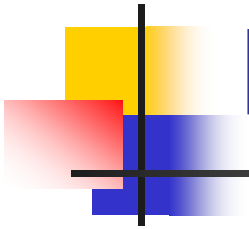
As the number of page frames increase, so does the fault rate.

Page Refs	4 Page Frames				
	Fault?	Page Contents			
A	yes	A			
B	yes	B	A		
C	yes	C	B	A	
D	yes	D	C	B	A
A	no	D	C	B	A
B	no	D	C	B	A
E	yes	E	D	C	B
A	yes	A	E	D	C
B	yes	B	A	E	D
C	yes	C	B	A	E
D	yes	D	C	B	A
E	yes	E	D	C	B

LRU

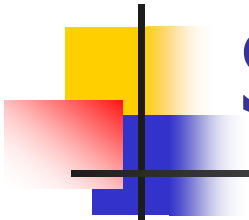
12 references,
10 faults

Page Refs	3 Page Frames			
	Fault?	Page Contents		
A	yes	A		
B	yes	B	A	
C	yes	C	B	A
D	yes	D	C	B
A	yes	A	D	C
B	yes	B	A	D
E	yes	E	B	A
A	no	A	E	B
B	no	B	A	E
C	yes	C	B	A
D	yes	D	C	B
E	yes	E	D	C



Least Recently Used Issues

- How to track "recency"?
  - use time
    - record time of reference with page table entry
    - use counter as clock
    - search for smallest time.
  - use stack
    - remove reference of page from stack (linked list)
    - push it on top of stack
- both approaches require large processing overhead, more space, and hardware support.



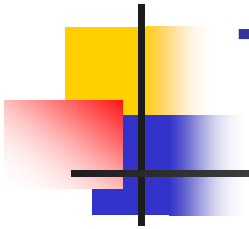
Second Chance

- Only one reference bit in the page table entry.
  - 0 initially
  - 1 When a page is referenced
- pages are kept in FIFO order using a circular list.
- Choose “victim” to evict
  - Select head of FIFO
  - If page has reference bit set, reset bit and select next page in FIFO list.
  - keep processing until you reach page with zero reference bit and page that one out.
- System V uses a variant of second chance

Second Chance Example

12 references,
9 faults

Page Refs	3 Page Frames			
	Fault?	Page Contents		
A	yes	A*		
B	yes	B*	A*	
C	yes	C*	B*	A*
D	yes	D*	C	B
A	yes	A*	D*	C
B	yes	B*	A*	D*
E	yes	E*	B	A
A	no	E*	B	A*
B	no	E*	B*	A*
C	yes	C*	E	B
D	yes	D*	C*	E
E	no	D*	C*	E*

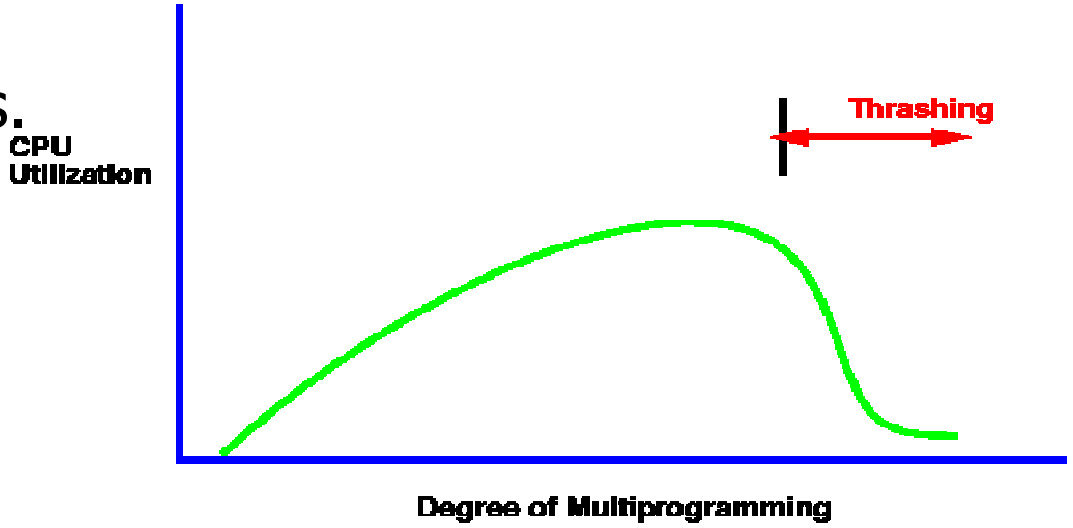


Thrashing

- Computations have locality.
- As page frames decrease, the page frames available are not large enough to contain the locality of the process.
- The processes start faulting heavily.
- Pages that are read in, are used and immediately paged out.

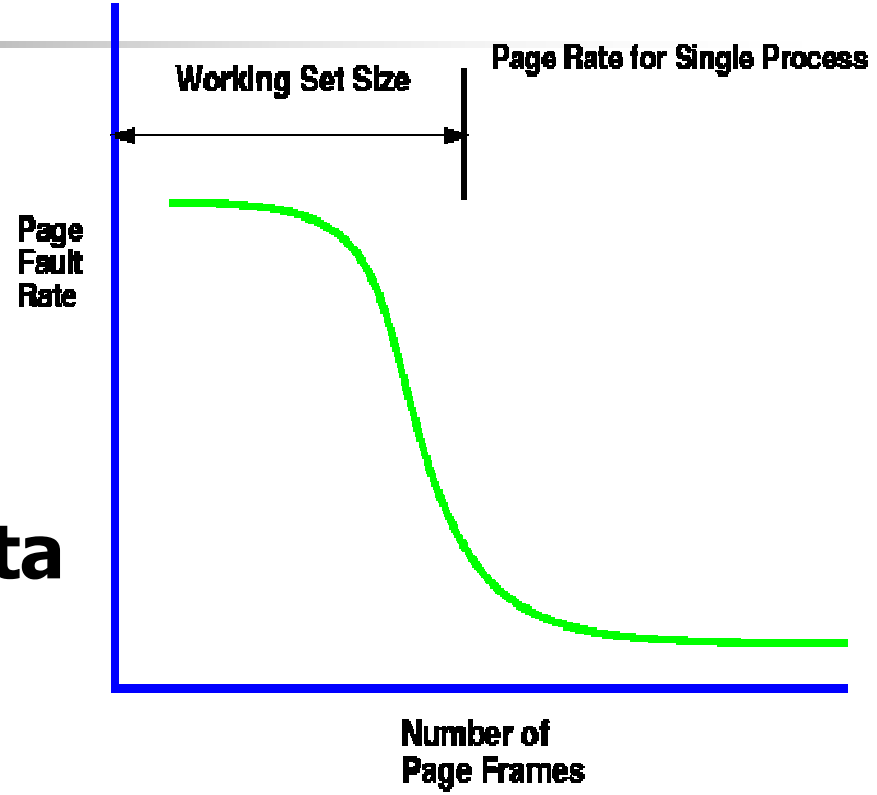
Thrashing and CPU Utilization

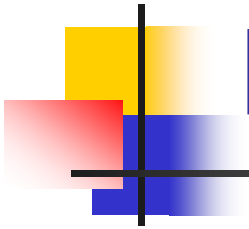
- As the page rate goes up, processes get suspended on page out queues for the disk.
- the system may try to optimize performance by starting new jobs.
- starting new jobs will reduce the number of page frames available to each process, increasing the page fault requests.
- system throughput plunges.



Working Set

- the working set model assumes locality.
- **the principle of locality states that a program clusters its access to data and text temporally**.
- *As the number of page frames increases above some threshold, the page fault rate will drop dramatically*.





Page Size Considerations

- Small pages
  - Reason:
    - Locality of reference tends to be small (256)
    - Less fragmentation
  - Problem: require large page tables
- Large pages
  - Reason
    - Small page table
    - I/O transfers have high seek time, so better to transfer more data per seek
  - Problem: Internal fragmentation, needless caching