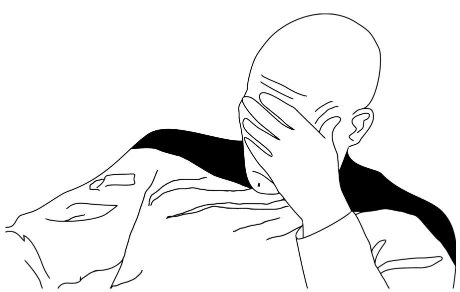
CS 423
Operating System Design: Virtual Memory Mgmt

Professor Adam Bates

Spring 2018

Goals for Today

- <u>Learning Objective</u>:
  - Understand properties of virtual memory systems
- <u>Announcements, etc</u>:
  - MP2 Out! **Due March 16th**



vSphere...

Reminder: Please put away devices at the start of class

History: Summary

Overlay → Fixed Partitions → Relocation

Overlay
- No multi-programming support

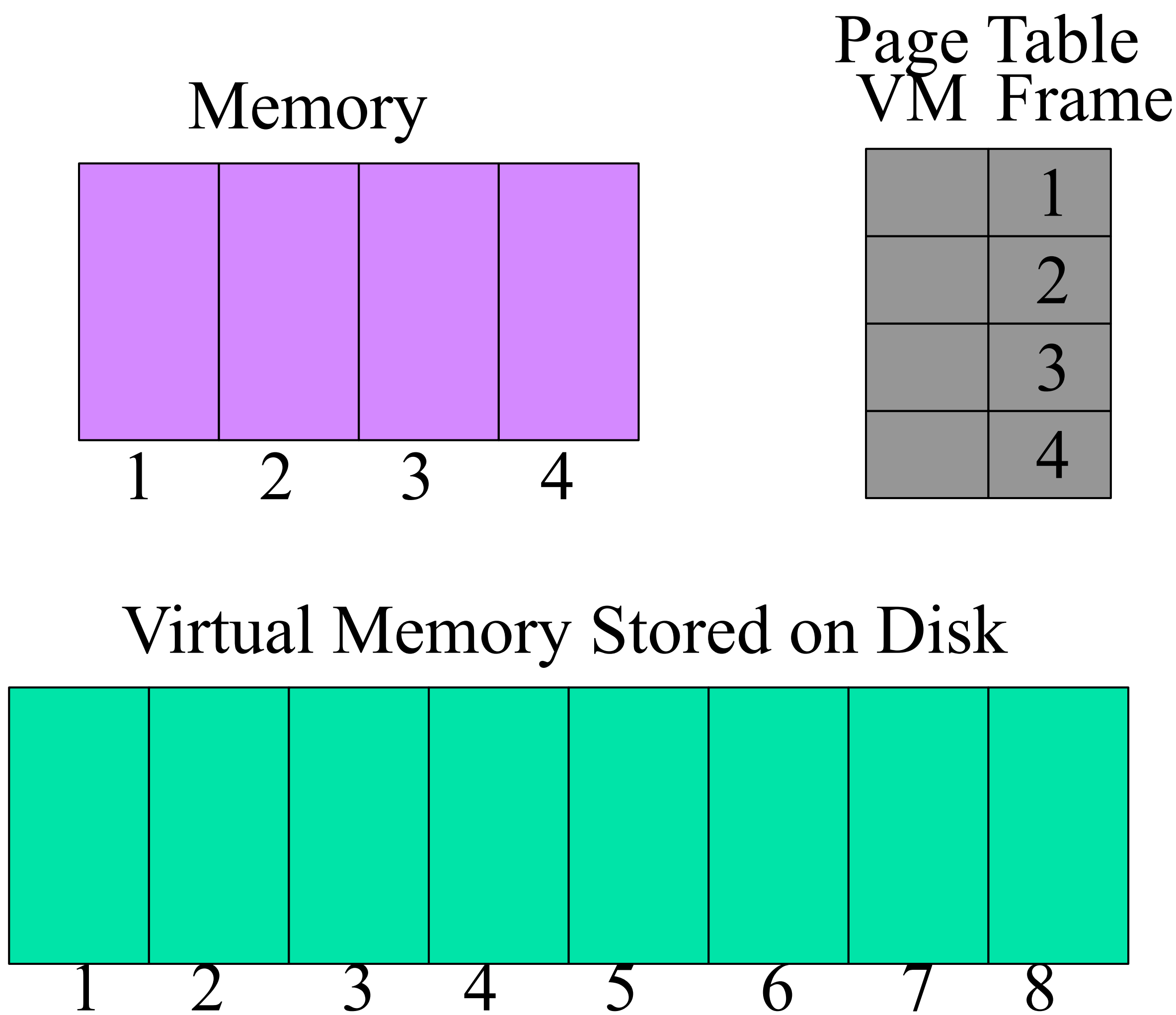
Fixed Partitions
- Supports multi-programming
- Internal fragmentation

Relocation
- No internal fragmentation
- Introduces external fragmentation

Virtual Memory

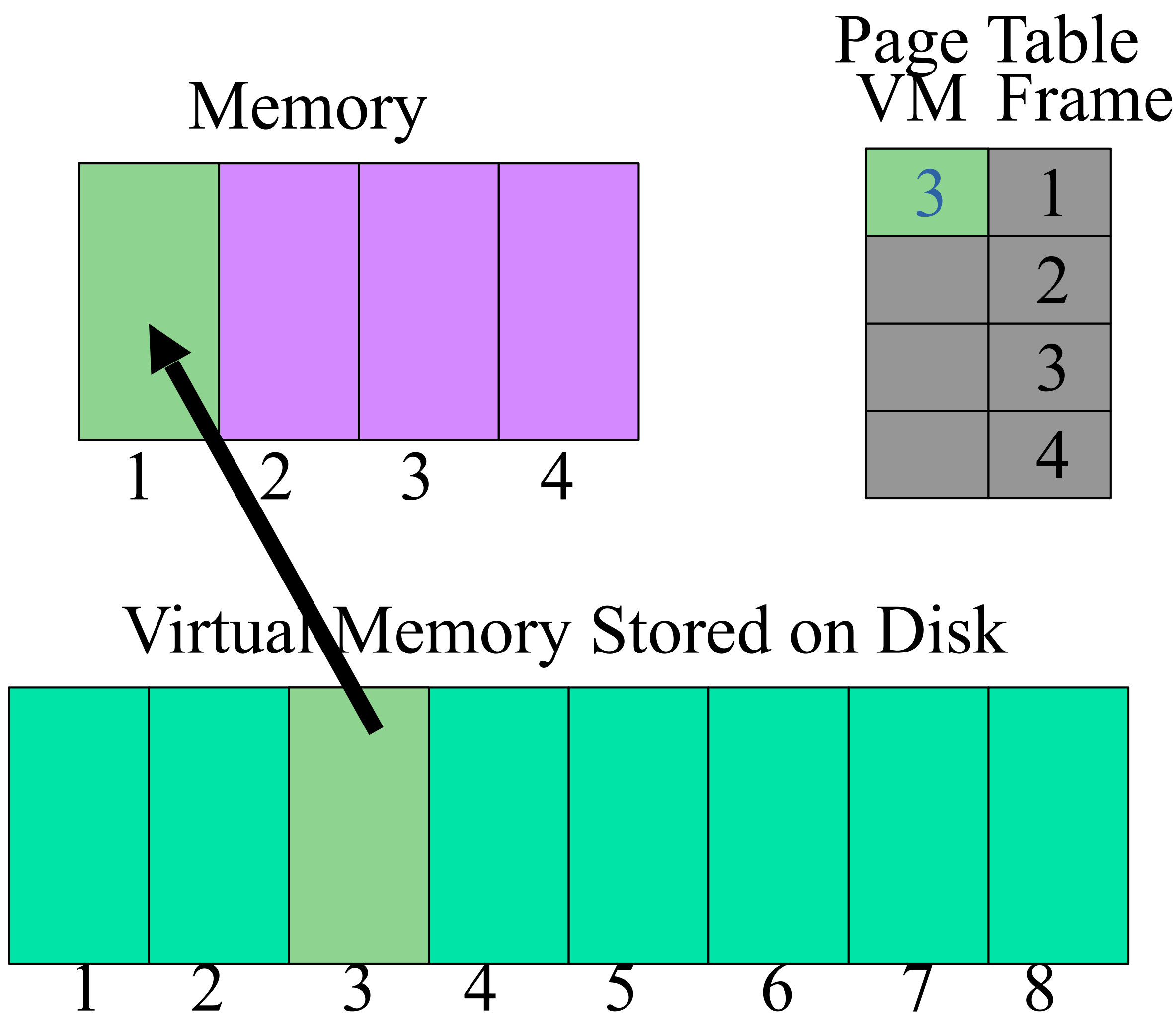
- Provide user with virtual memory that is as big as user needs
- Store virtual memory on disk
- Cache parts of virtual memory being used in real memory
- Load and store cached virtual memory without user program intervention

Paging


Page Table
VM Frame
1
2
3
4
Memory
1
2
3
4
Virtual Memory Stored on Disk
1
2
3
4
5
6
7
8

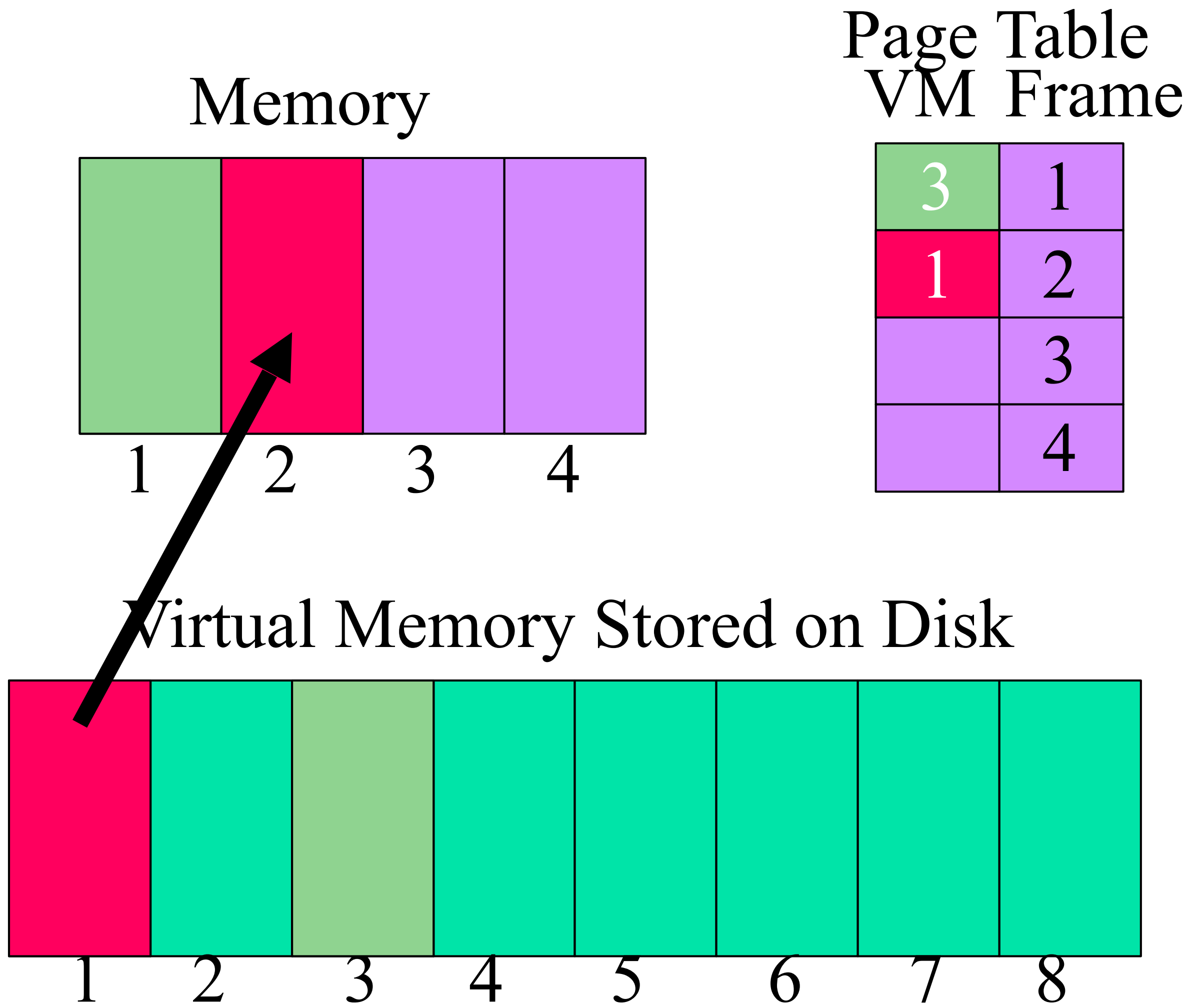
Paging

Request Page 3…



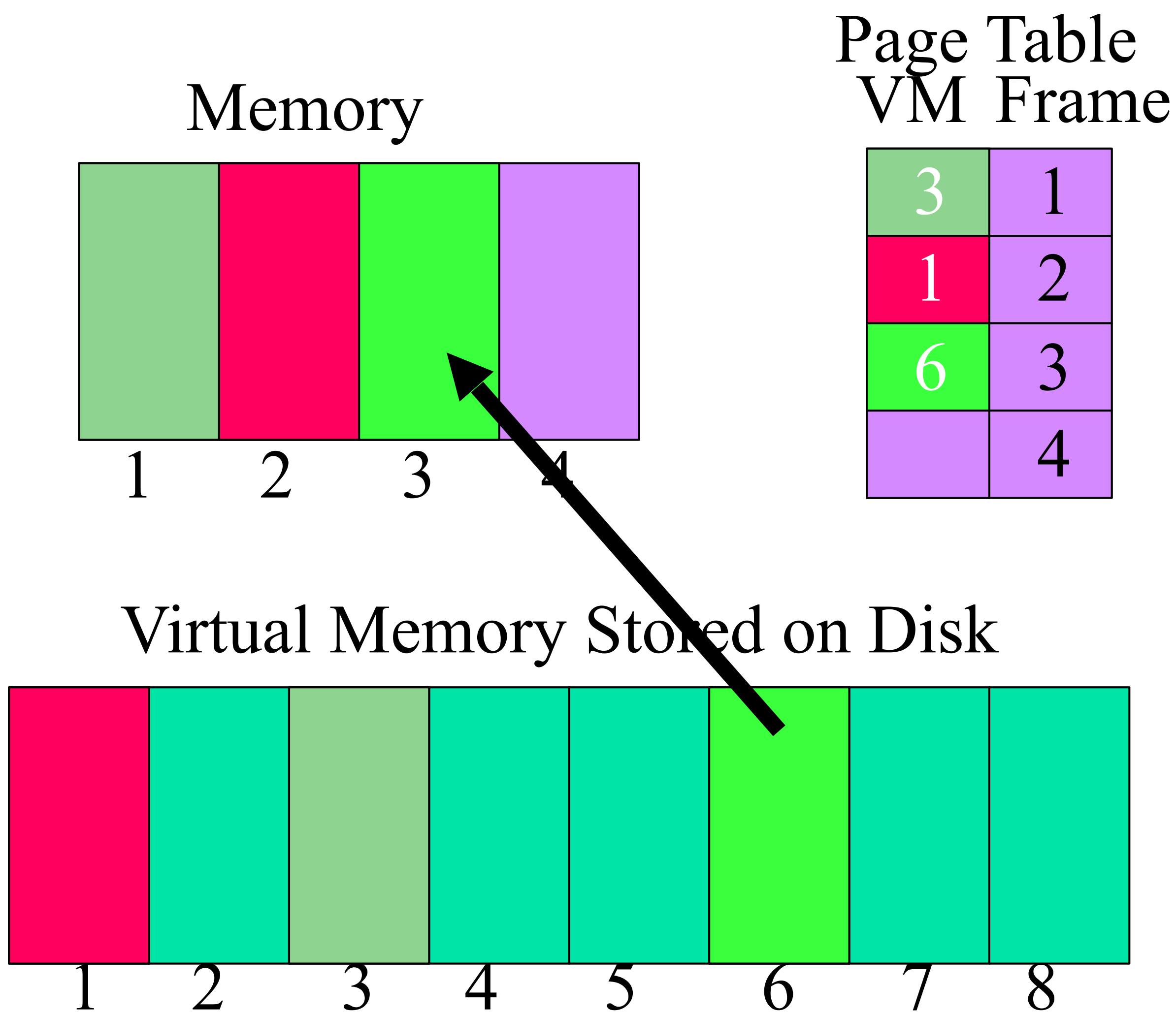
Paging

Request Page 1…



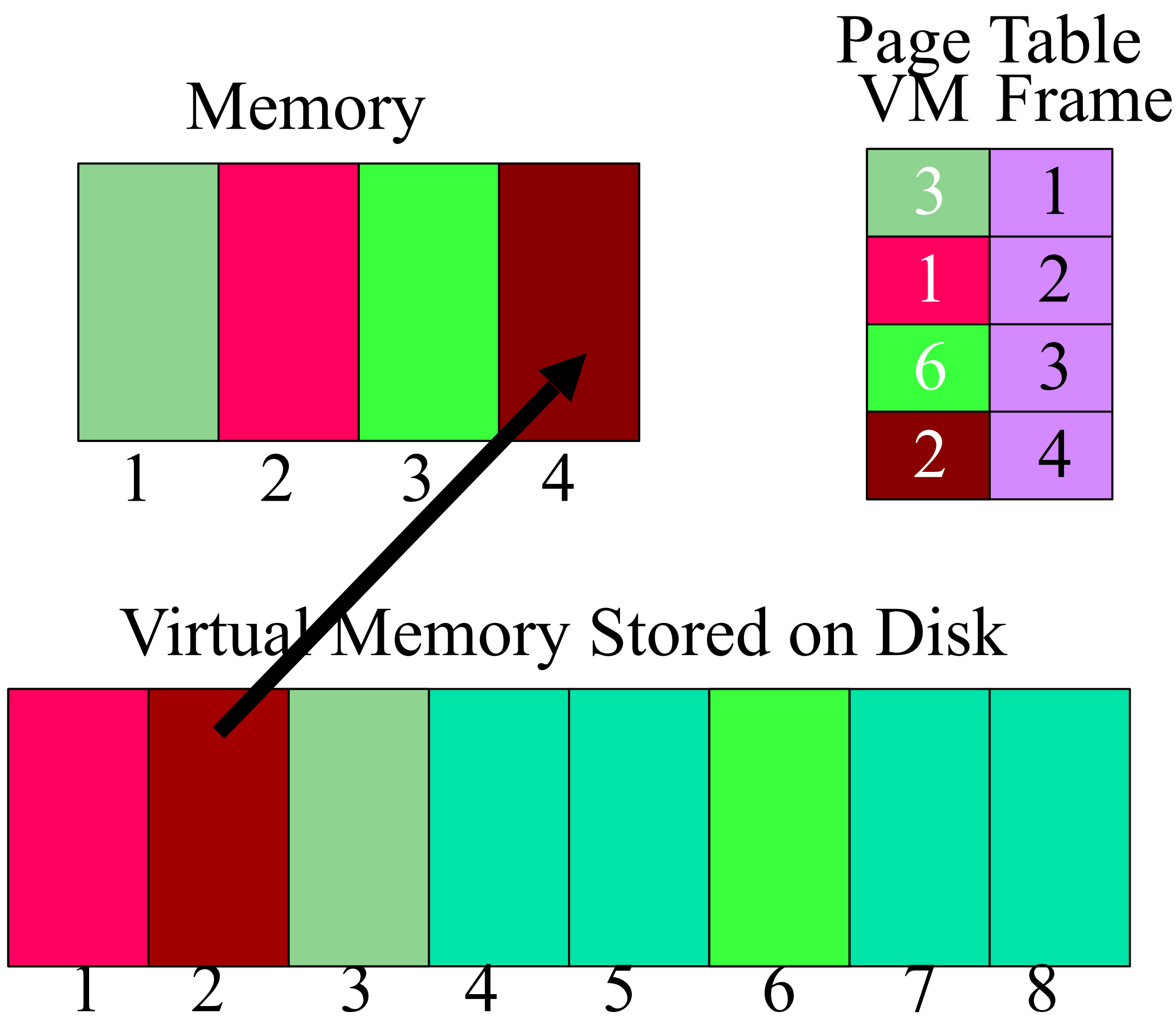
Paging

Request Page 6…

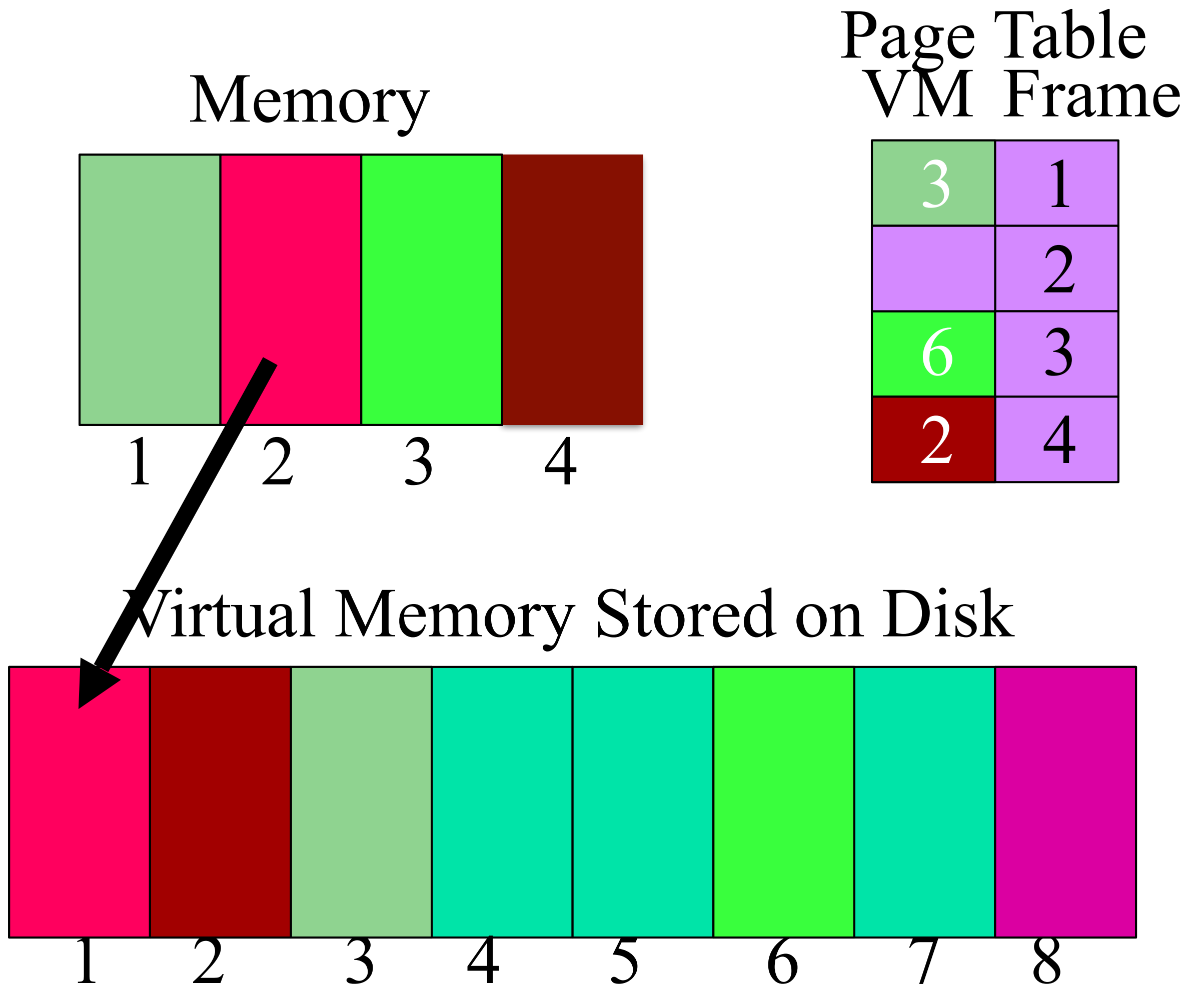


Paging

Request Page 2…

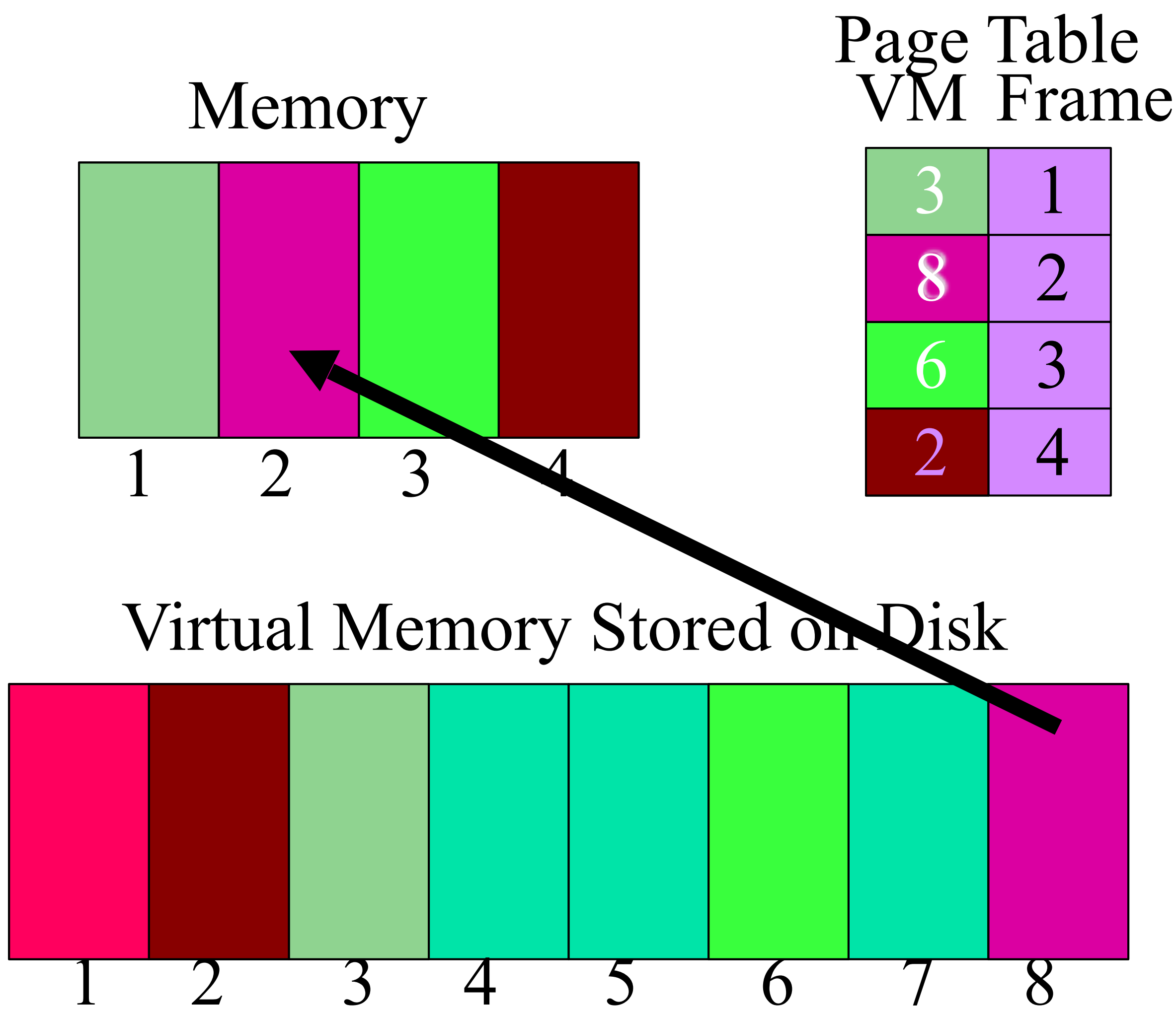


Request Page 8. Swap Page 1 to Disk First…

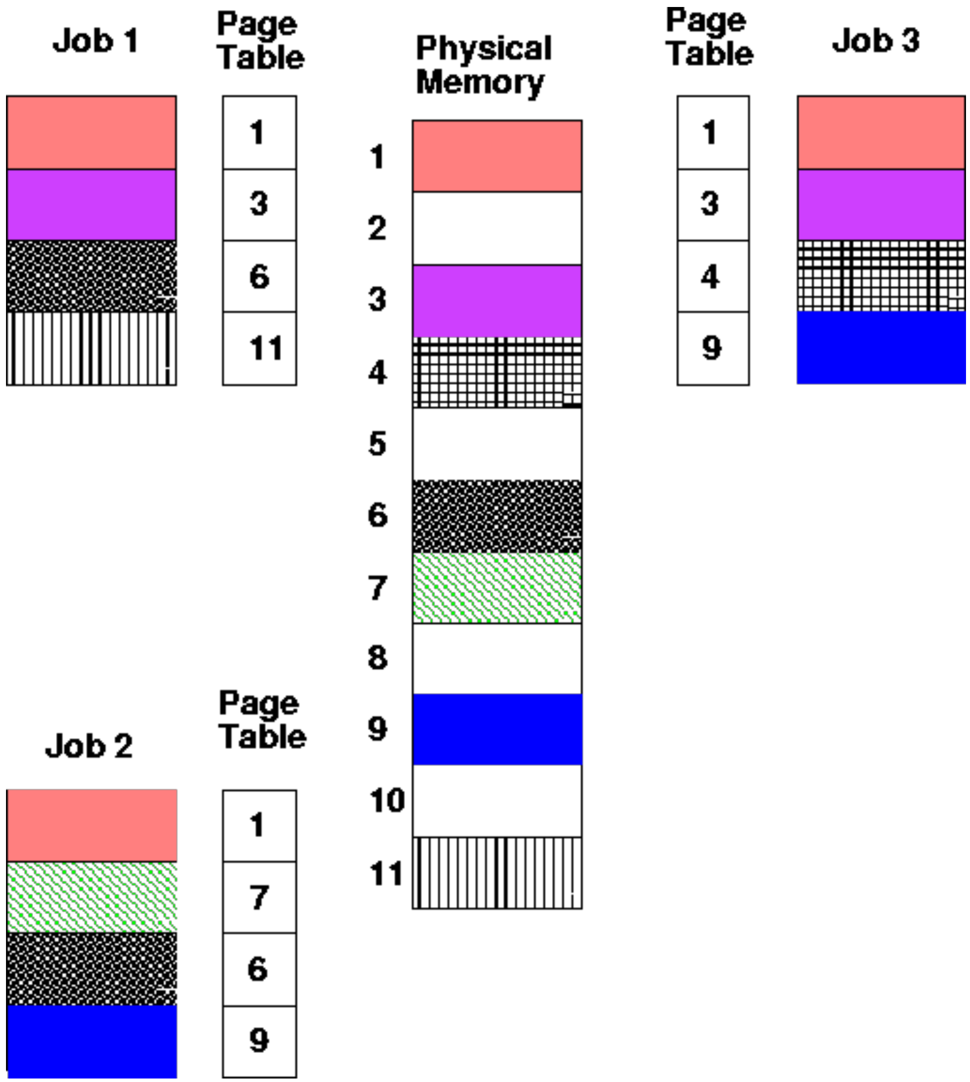


Paging

Request Page 8. … now load Page 8 into Memory.

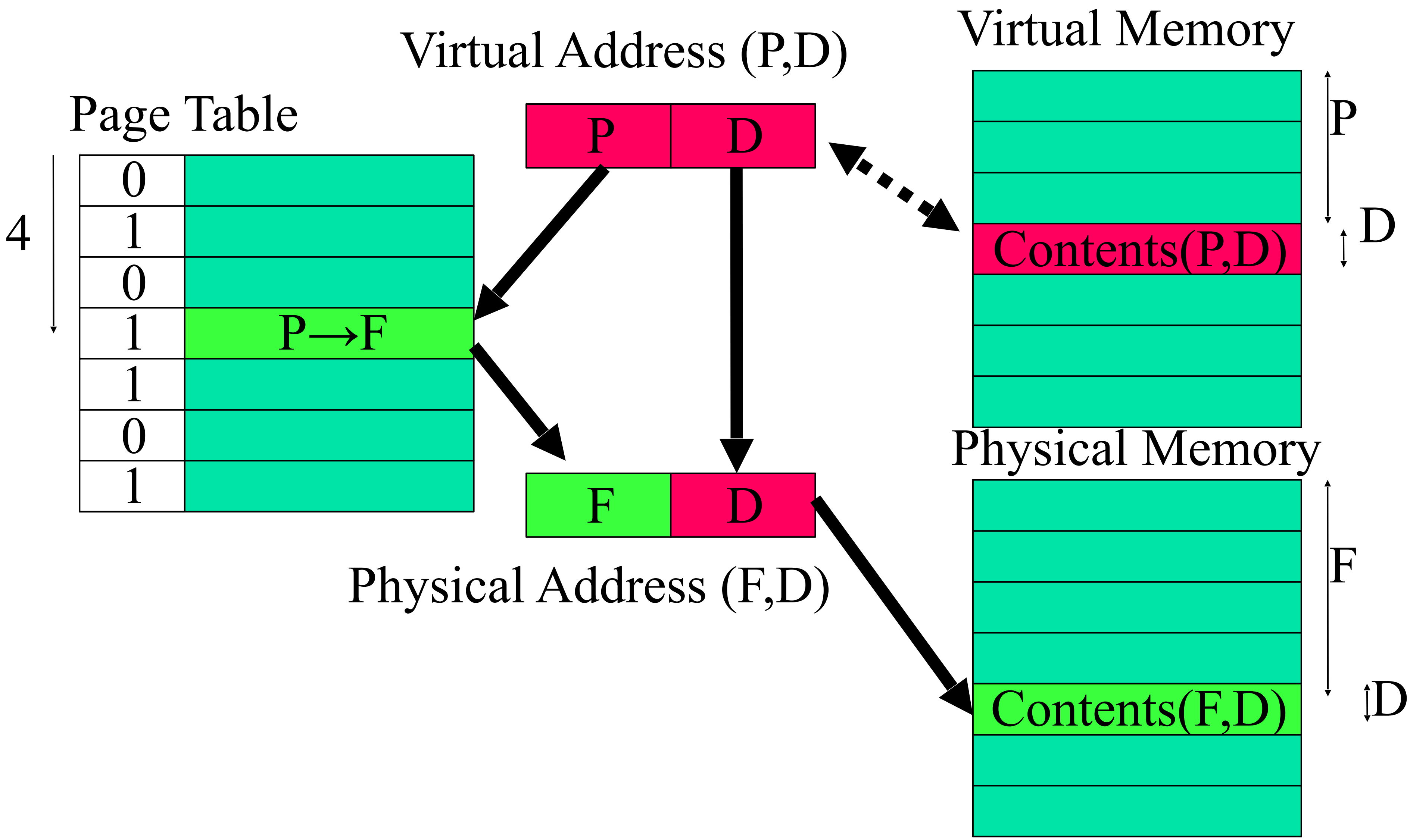


Shared Pages

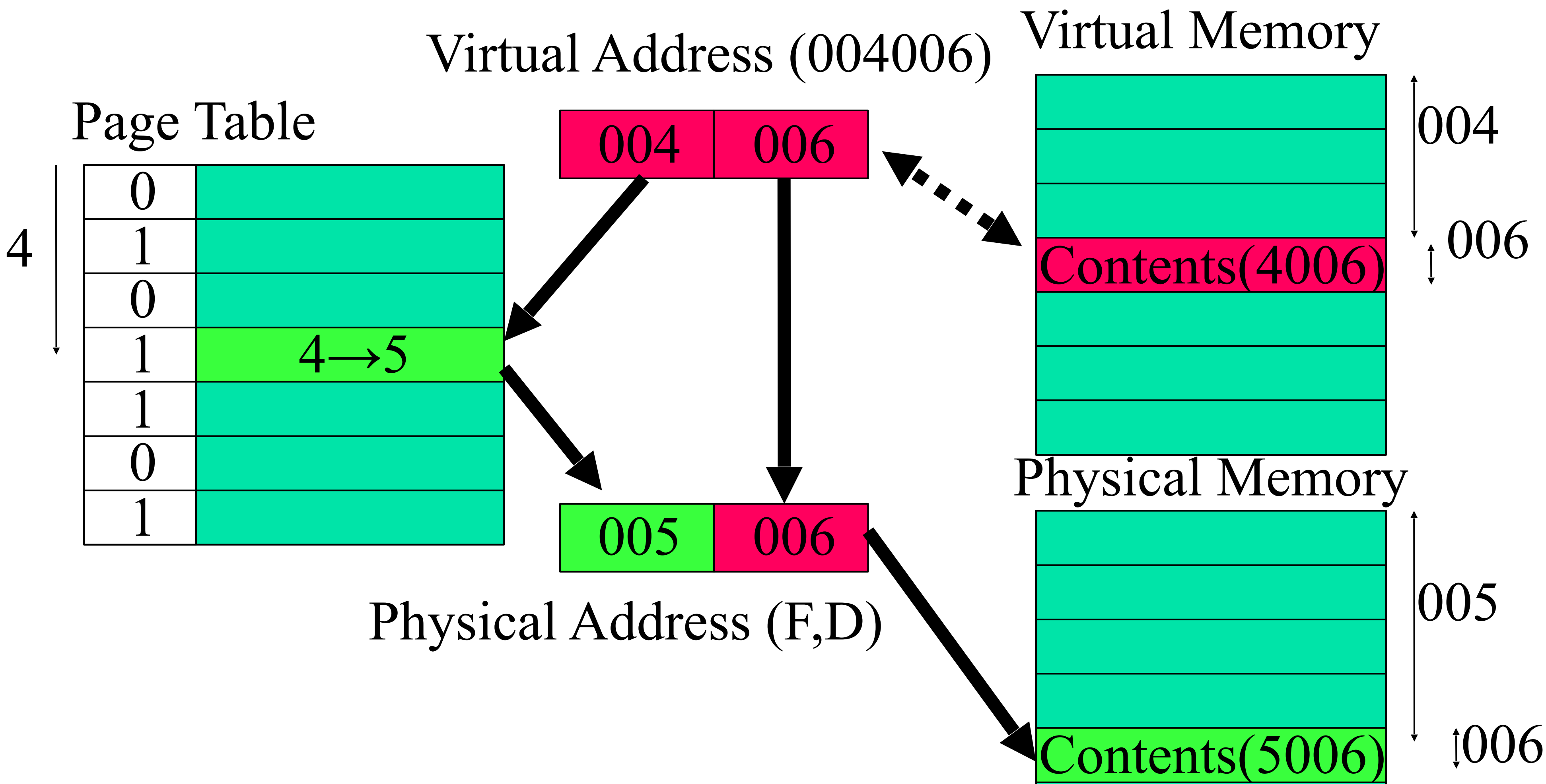


Note: Virtual Memory also supports shared pages.

Page Mapping Hardware



Page Mapping Hardware



Page size 1000
Number of Possible Virtual Pages 1000
Number of Page Frames 8

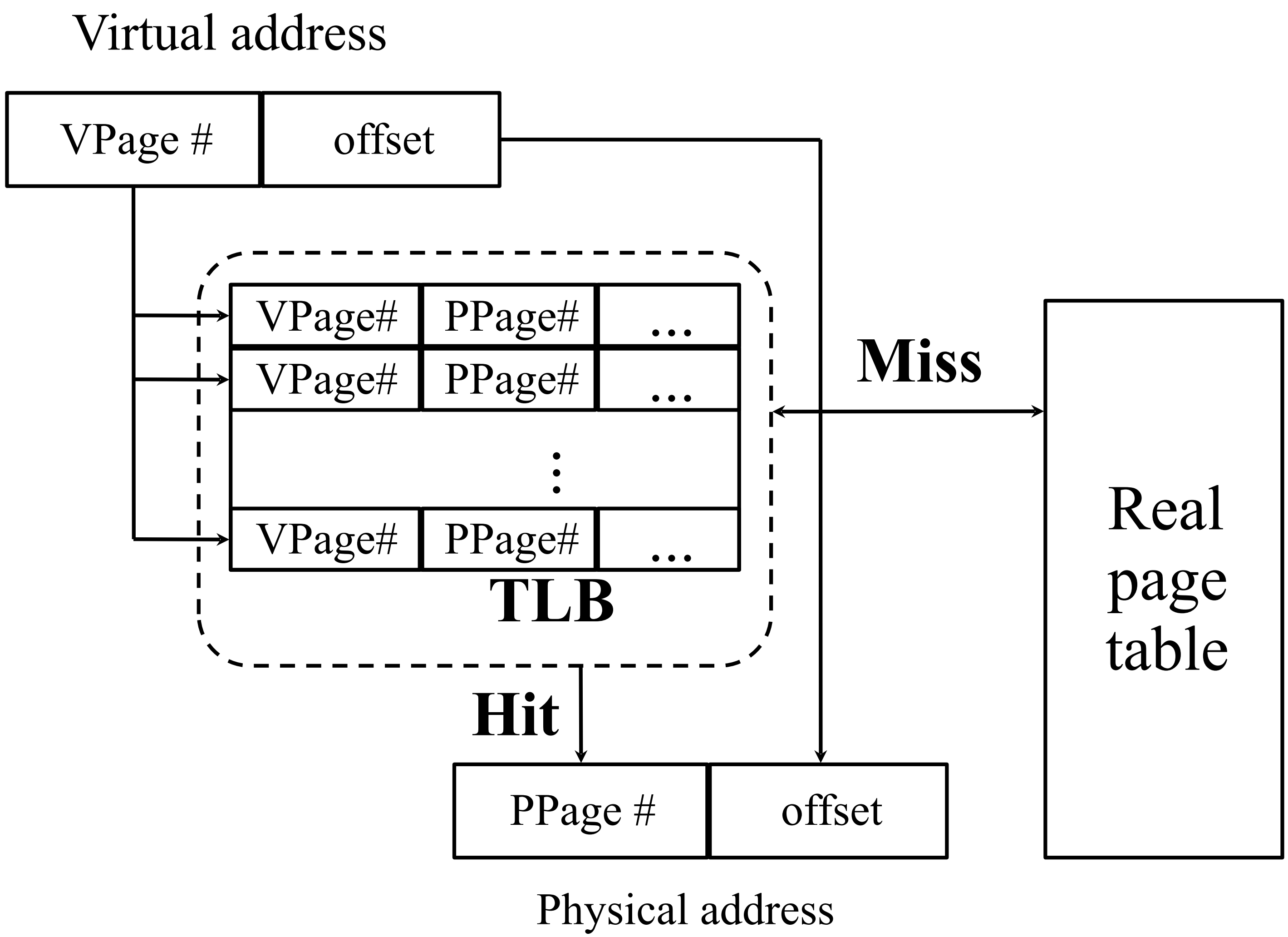
Page Faults

- 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
- **Other sources of overhead besides page faults??**

Optimization:



Translation Lookaside Buffers

- 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.

Translation Lookaside Buffers

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 = $2m + \sigma - m\,\eta$

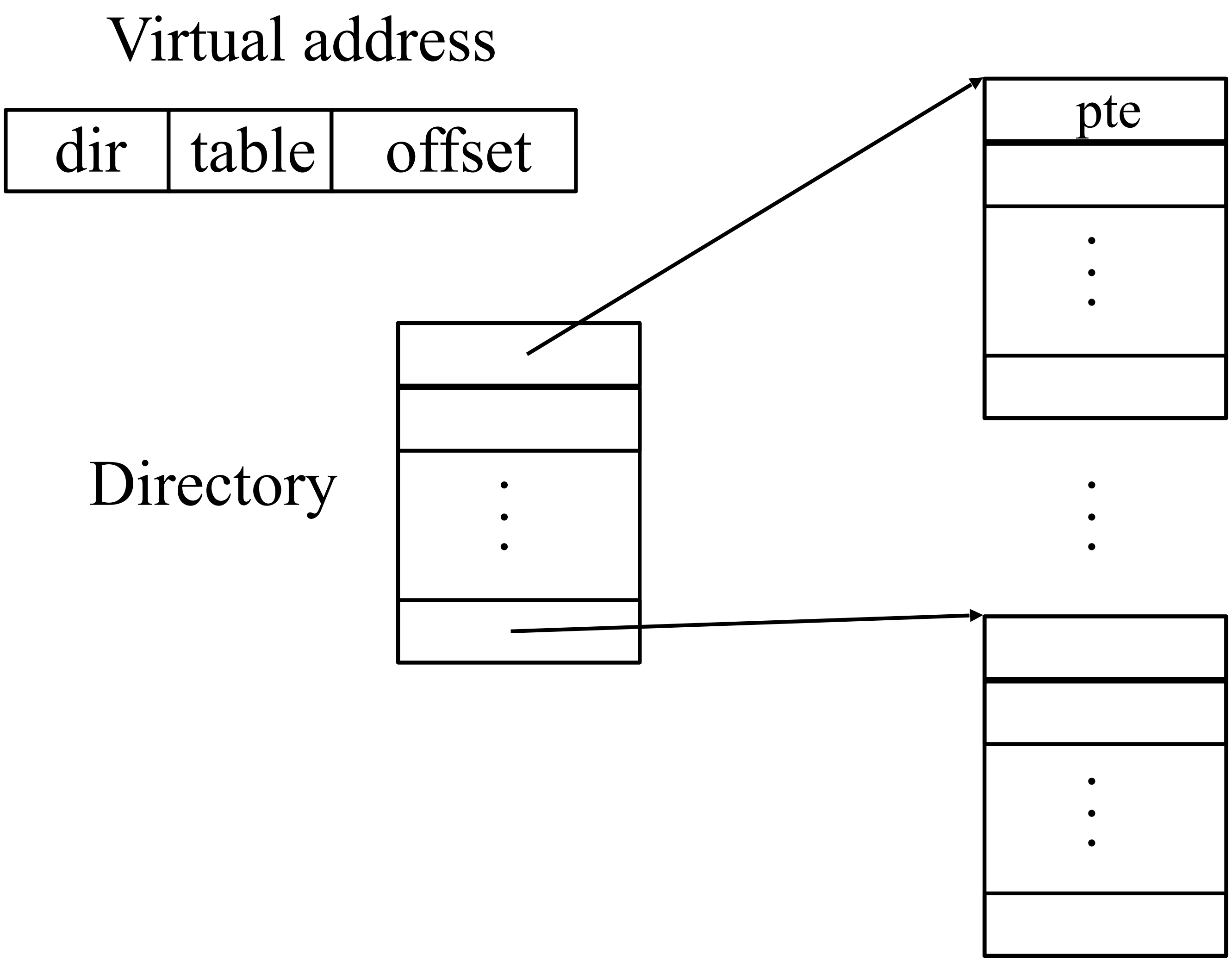
Note: Doesn't consider page faults. How would we extend?

Question

Applications might make sparse use of their virtual address space. How can we make our page tables more efficient?

Multi-level Page Tables

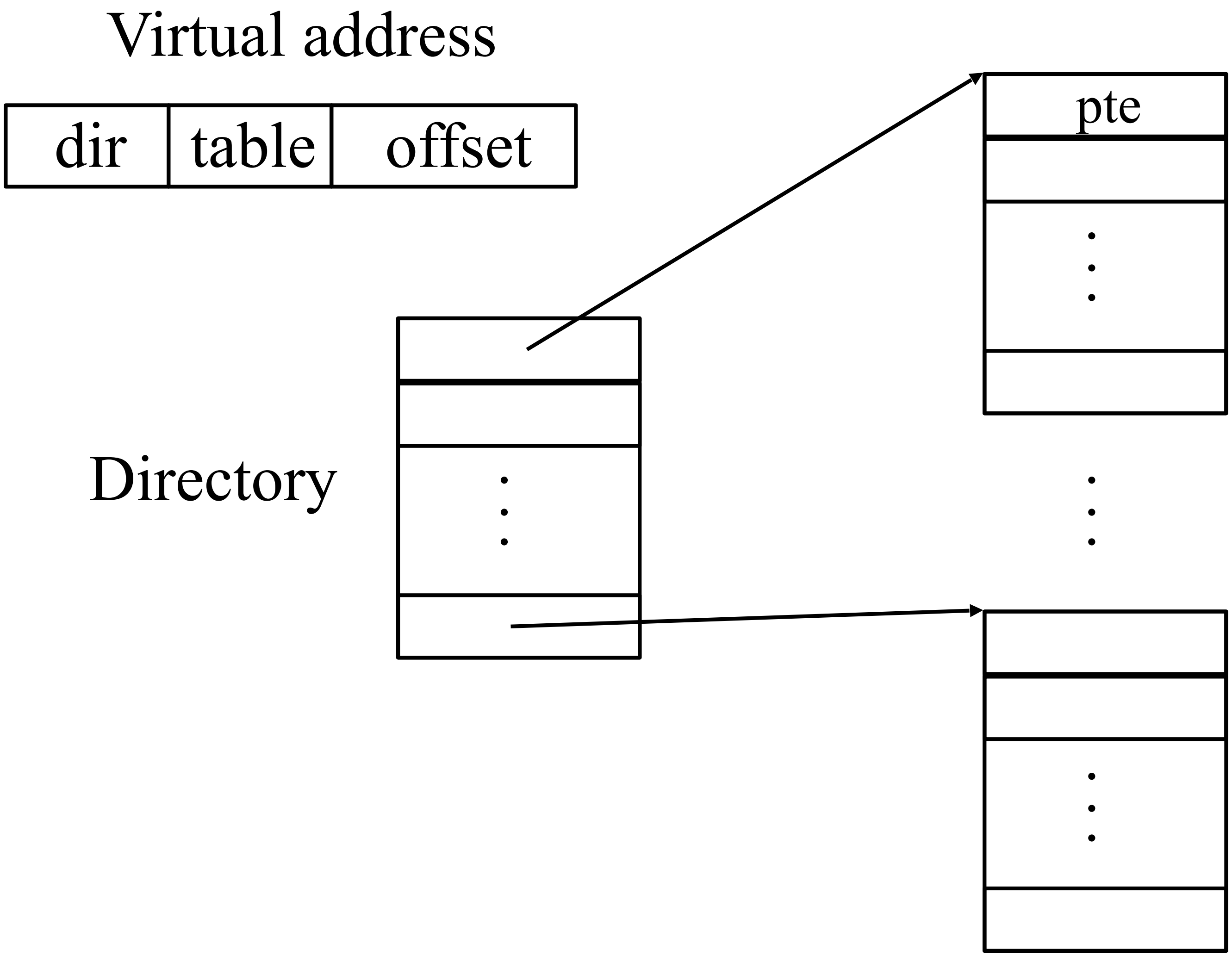
What does this buy us?



Multi-level Page Tables

What does this buy us?

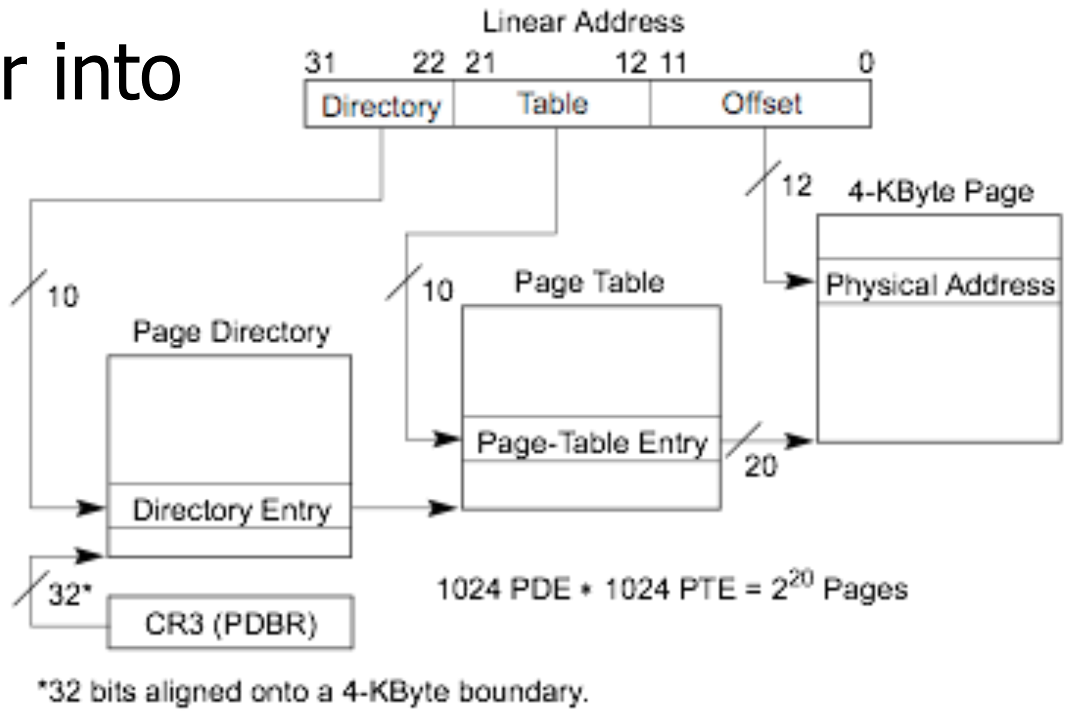
Answer: Sparse address spaces, and easier paging



Multi-level Page Tables

Example: Addressing in a Multi-level 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



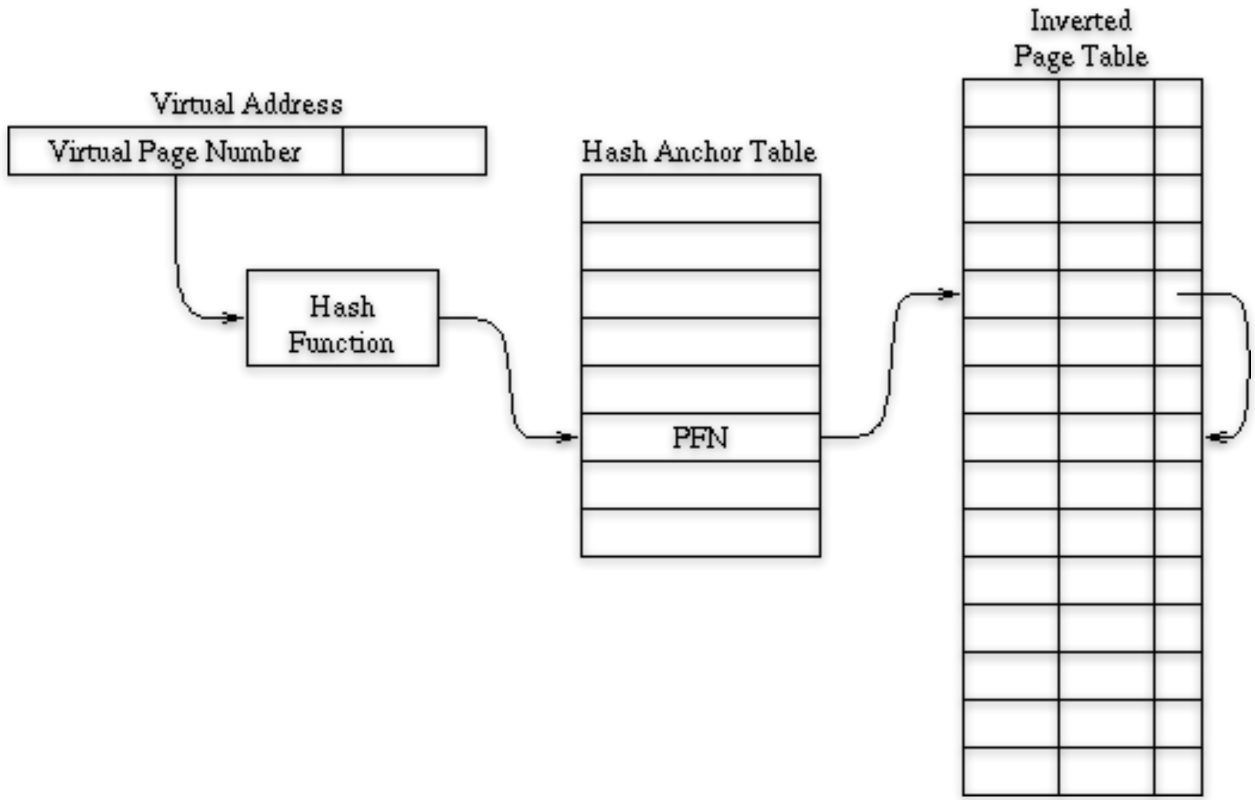
Multi-level Paging Performance

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

Question

In 64-bit system, up to 2^52 PT entries.
2^52 ~= 1,000,000,000,000,000
... bro, can I borrow some RAM?

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.

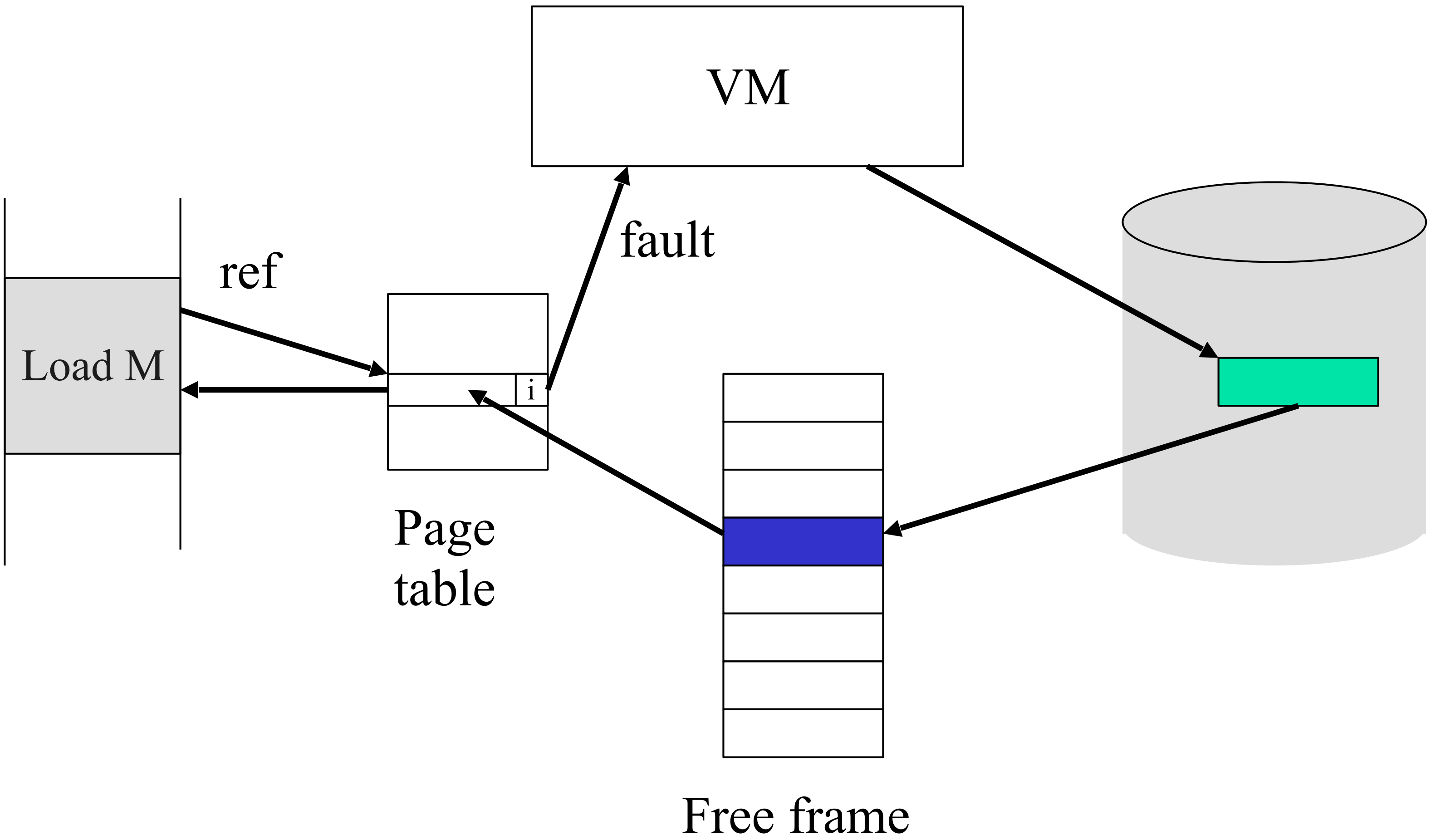
Paging Policies

- 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