


# Memory

---

Copyright ©: Nahrstedt, Angrave, Abdelzaher

1

Copyright ©: Nahrstedt, Angrave, Abdelzaher

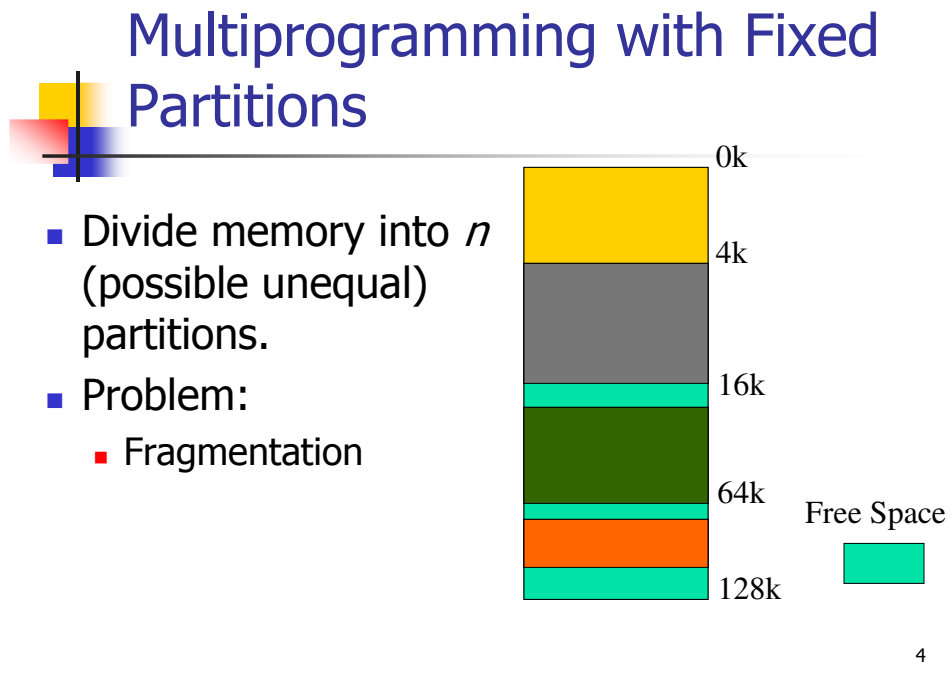
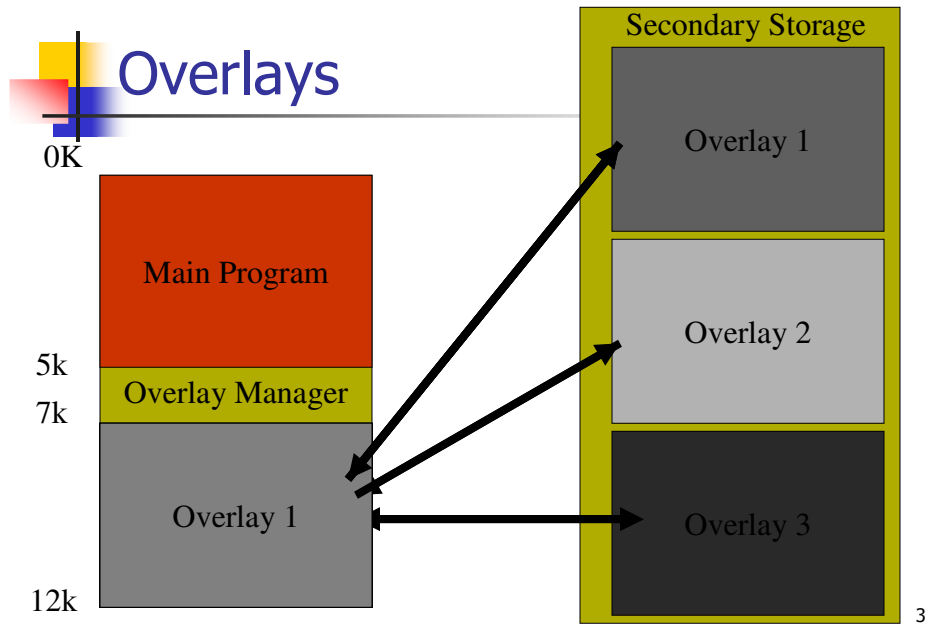


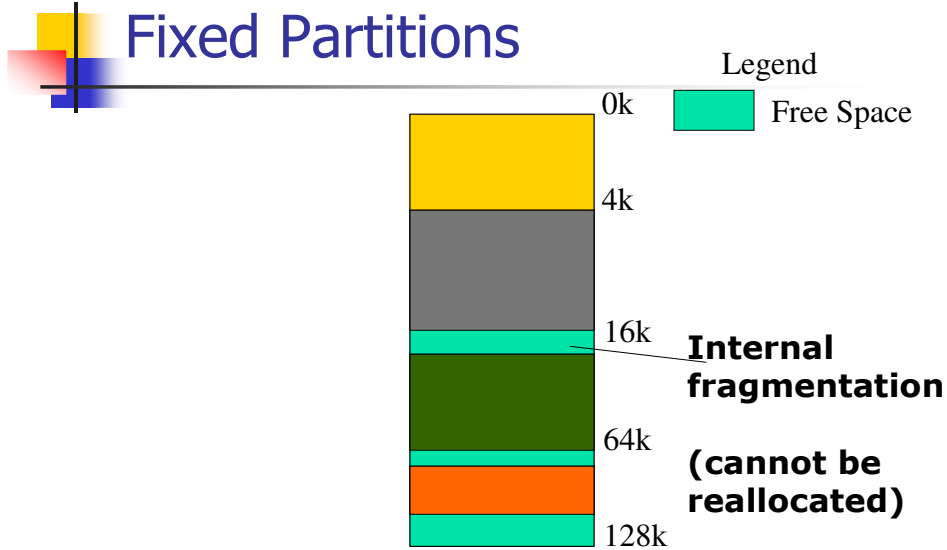
## Memory Allocation

---

- Compile for overlays
- Compile for fixed Partitions
  - Separate queue per partition
  - Single queue
- Relocation and variable partitions
  - Dynamic contiguous allocation (bit maps versus linked lists)
- Fragmentation issues
- Swapping
- Paging

2





5

## Fixed Partition Allocation Implementation Issues

- Separate input queue for each partition
  - Requires sorting the incoming jobs and putting them 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

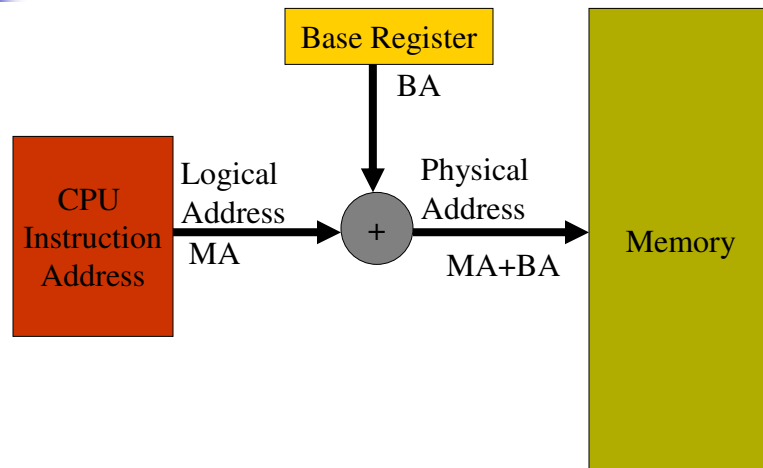
6

## Relocation

- Correct starting address when a program starts in 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, Virtual 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
- Memory-management unit (MMU)
  - map virtual to physical addresses.
- Relocation register
  - Mapping requires hardware (MMU) with the base register

7

## Relocation Register



8

## Question 1 - Protection

### ■ Problem:

- How to prevent a malicious process to write or jump into other user's or OS partitions

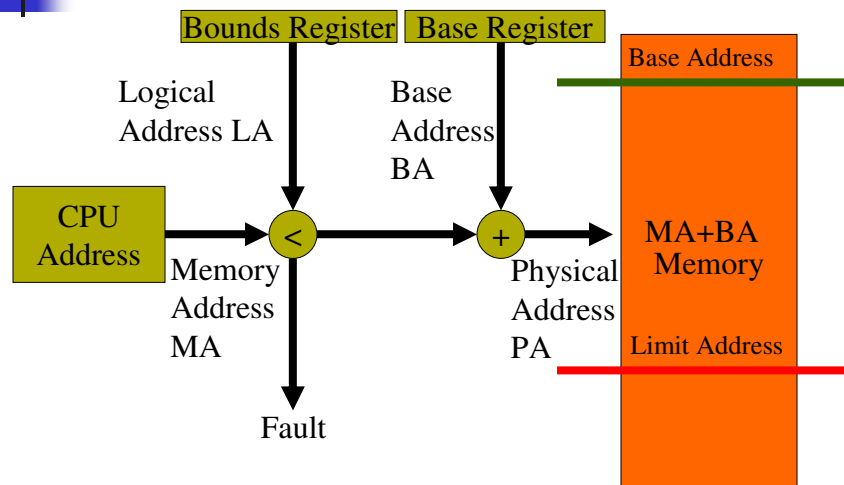
### ■ Solution:

- Base bounds registers



9

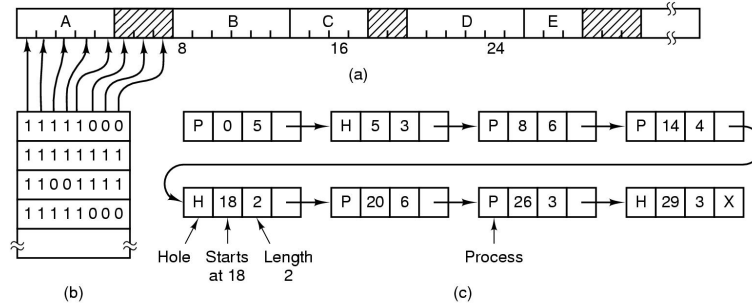
## Base Bounds Registers



10

# Contiguous Allocation and Variable Partitions:

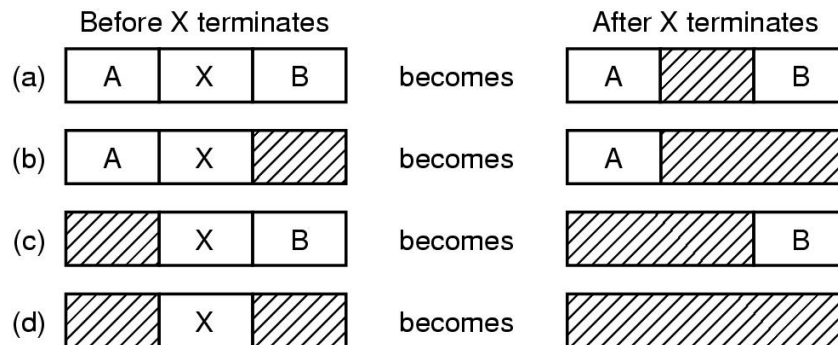
## Bit Maps versus Linked Lists



- Part of memory with 5 processes, 3 holes
  - tick marks show allocation units
  - shaded regions are free
- Corresponding bit map

11

## More on Memory Management with Linked Lists



- Four neighbor combinations for the terminating process X

12

## Contiguous Variable Partition Allocation schemes

- Bitmap and link list
  - Which one occupies more space?
    - Depending on the individual memory allocation scenario.  
In most cases, **bitmap usually occupies more space.**
  - Which one is faster to reclaim freed space?
    - On average, bitmap is faster because it just needs to set the corresponding bits
  - Which one is faster to find a free hole?
    - On average, a link list is faster because we can link all free holes together

13

## Storage Placement Strategies

- Best fit
  - Use the hole whose size is equal to the need, or if none is equal, the whole that is larger but closest in size.
  - Rationale?
- First fit
  - Use the first available hole whose size is sufficient to meet the need
  - Rationale?
- Worst fit
  - Use the largest available hole
  - Rationale?

14



## Storage Placement Strategies

- Every placement strategy has its own problem
  - Best fit
    - Creates small holes that cant be used
  - Worst Fit
    - Gets rid of large holes making it difficult to run large programs
  - First Fit
    - Creates average size holes

15



## How Bad Is Fragmentation?

- Statistical arguments - Random sizes
- First-fit
- Given N allocated blocks
- $0.5 \cdot N$  blocks will be lost because of fragmentation
- Known as 50% RULE

16



## Solve Fragmentation w. Compaction



17

## Storage Management Problems

- Fixed partitions suffer from
  - internal fragmentation
- Variable partitions suffer from
  - external fragmentation
- Compaction suffers from
  - overhead

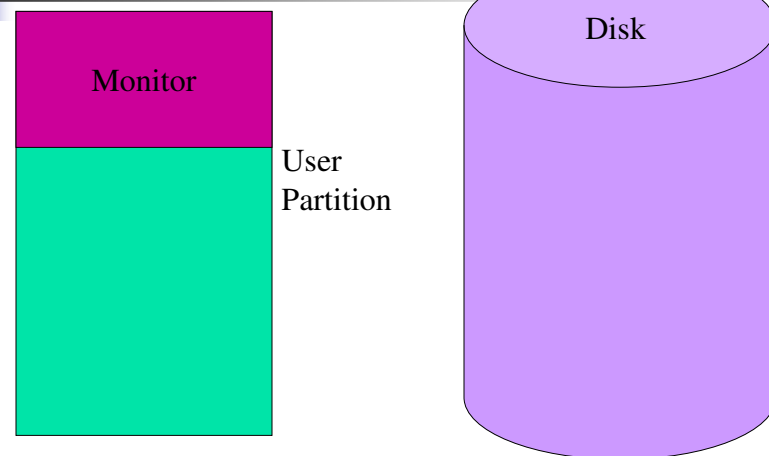
18

## Question

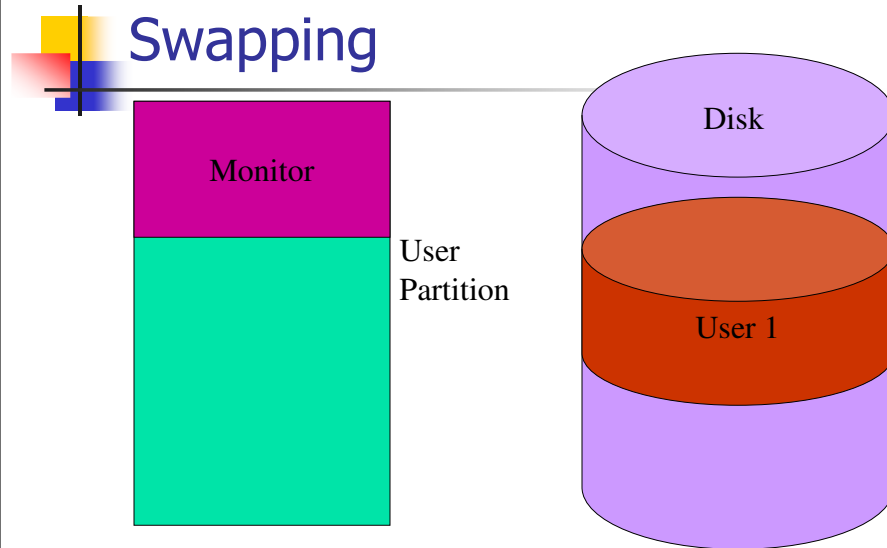
- What if there are more processes than what could fit into the memory?

19

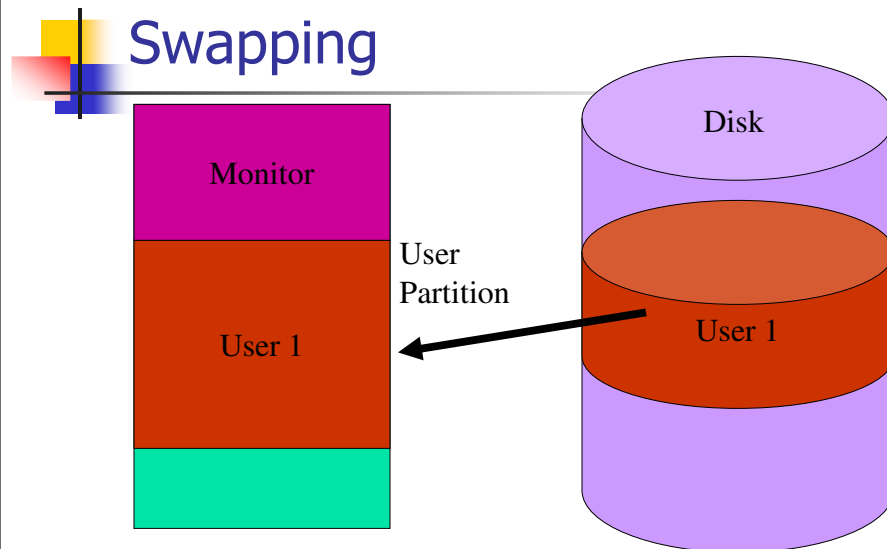
## Swapping



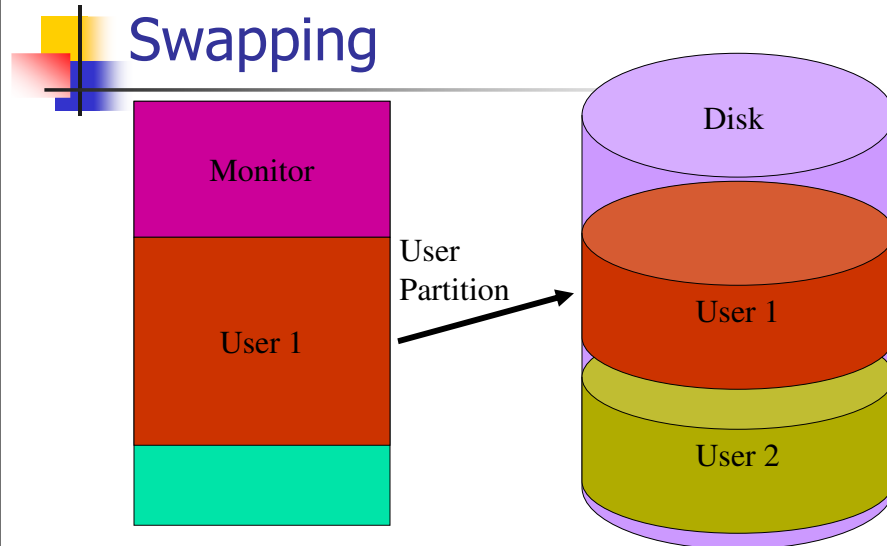
20



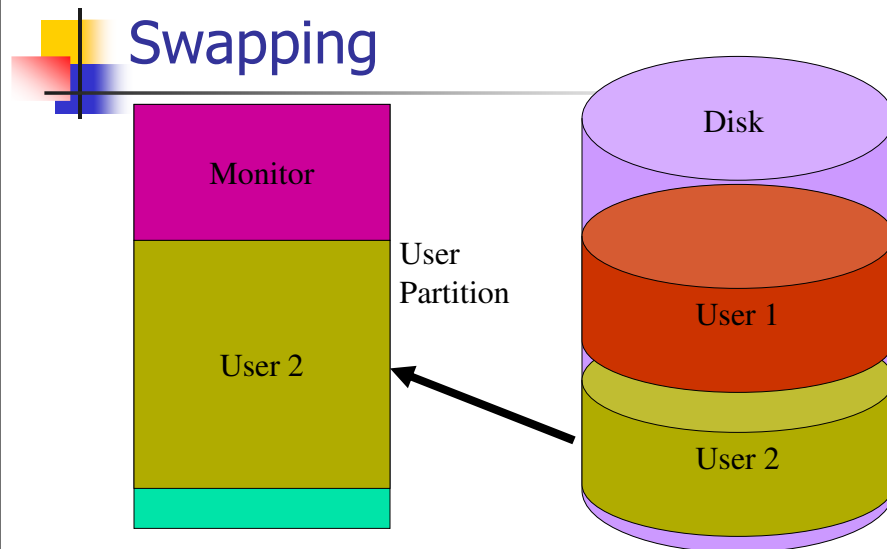
21



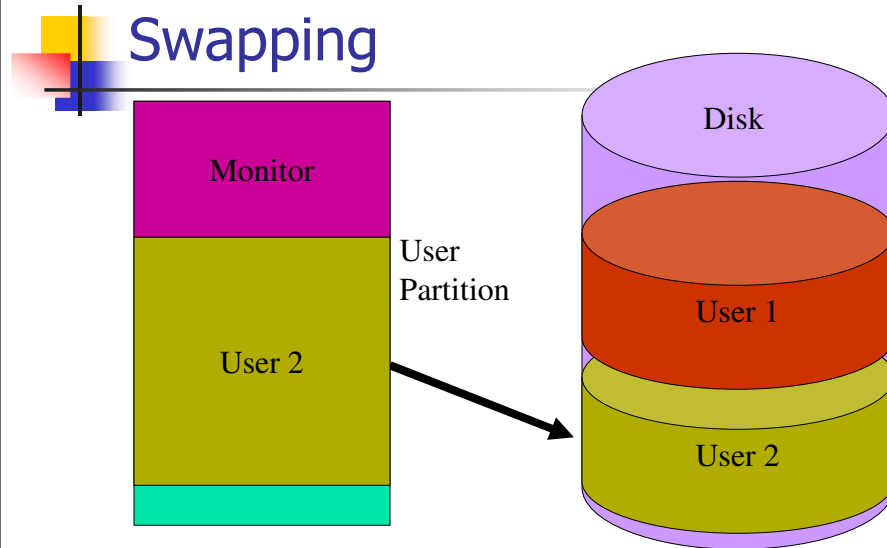
22



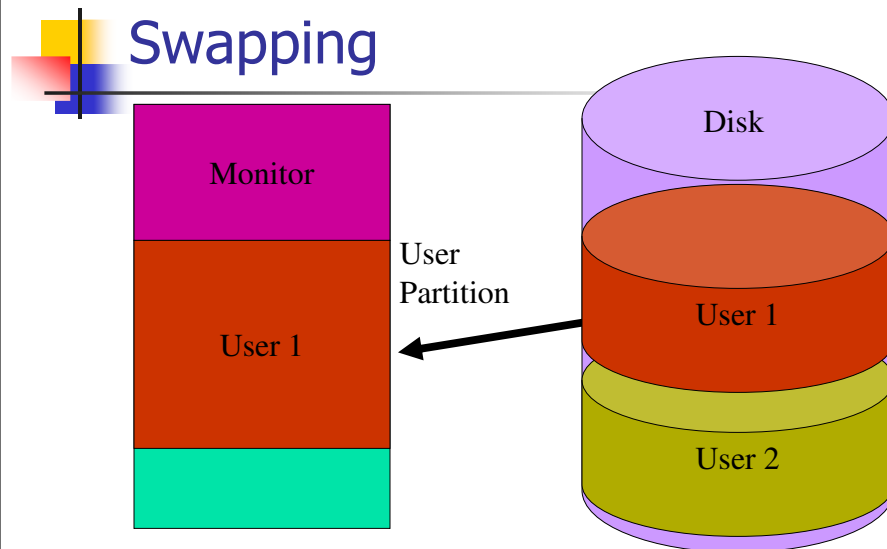
23



24



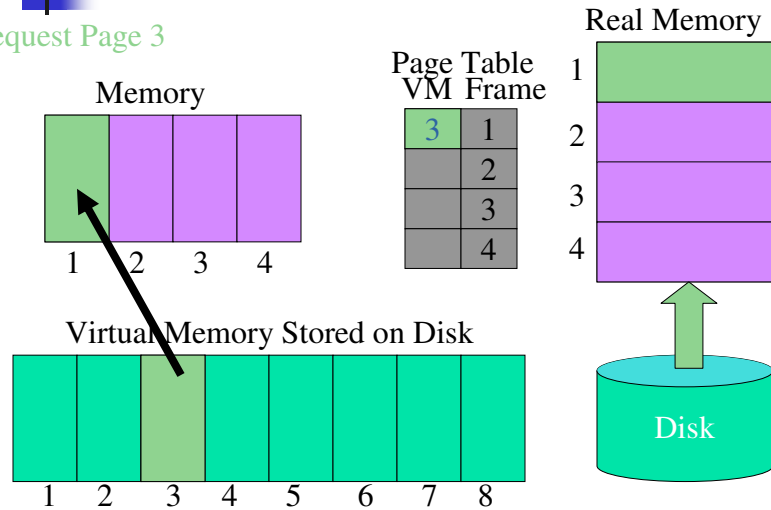
25



26

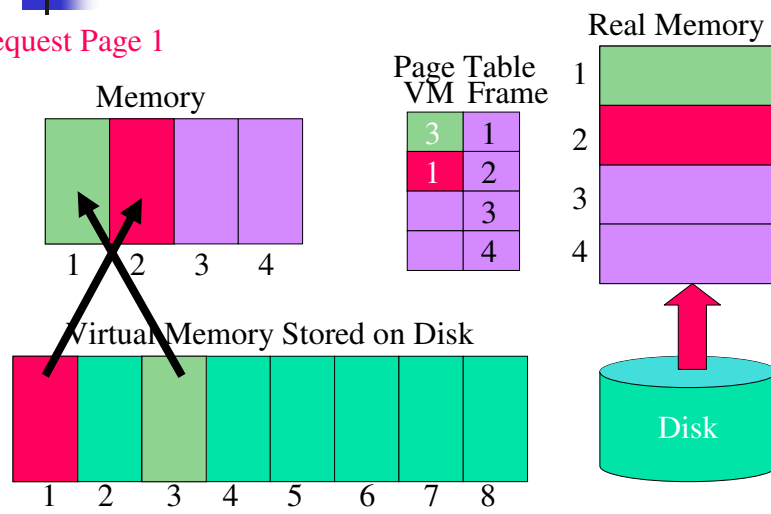
## Paging Request

Request Page 3



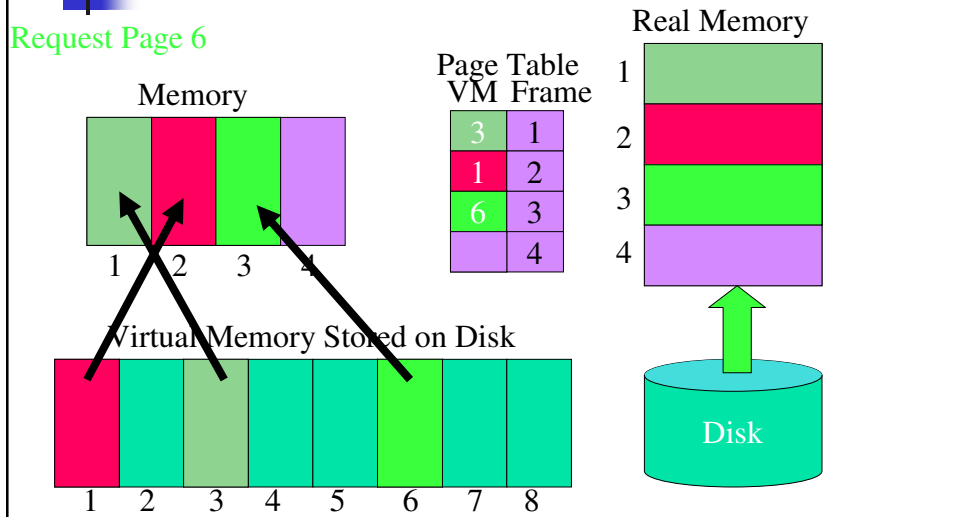
## Paging

Request Page 1



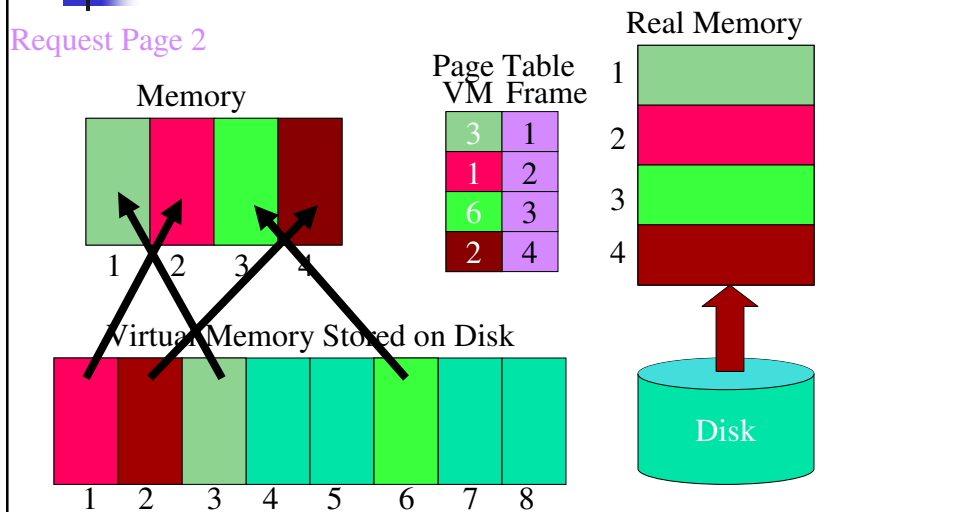
# Paging

Request Page 6

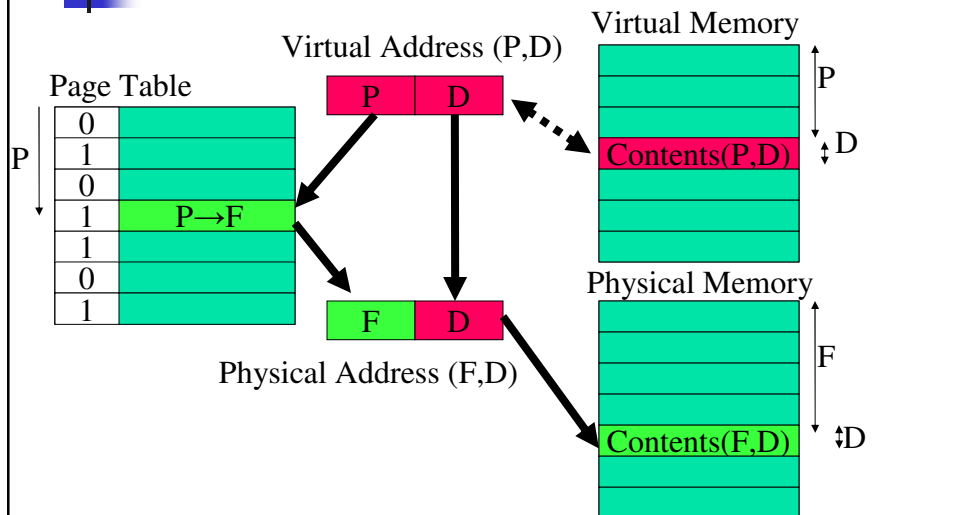


# Paging

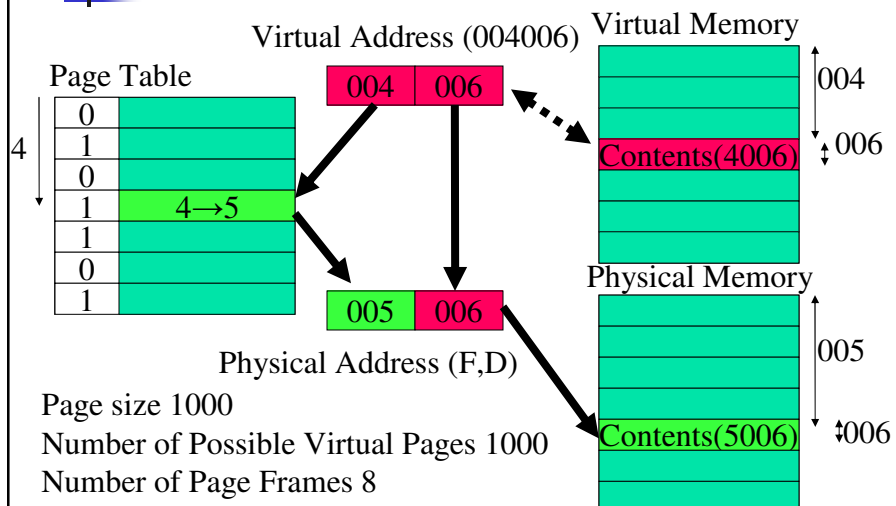
Request Page 2



## Page Mapping Hardware



## Page Mapping Hardware







## 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



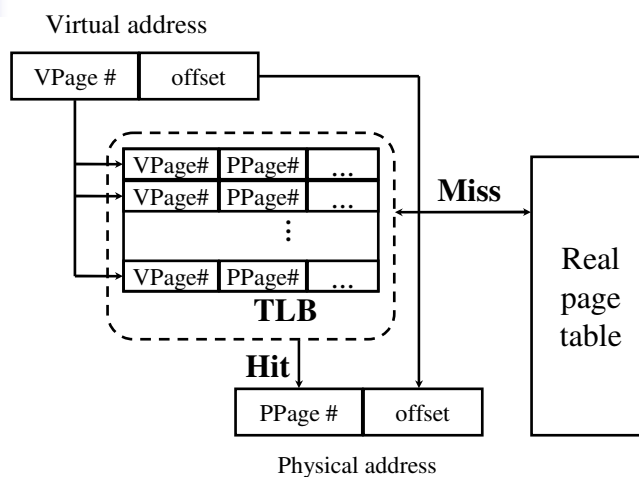
## Paging Issues

- 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)
  - page frames must map into real memory
  - Page Table base register must be changed for context switch
- *No* external fragmentation; internal fragmentation on last page *only*

## Virtual-to-Physical Lookups

- Programs only know virtual addresses
  - The page table can be extremely large
- Each virtual address must be translated
  - May involve walking hierarchical page table
  - Page table stored in memory
  - So, each program memory access requires several actual memory accesses
- Solution: cache “active” part of page table
  - TLB is an “associative memory”

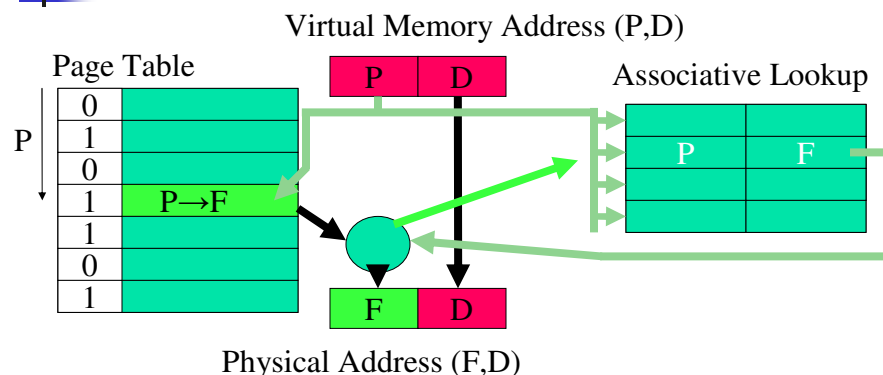
## Translation Lookaside Buffer (TLB)



## 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.

## Page Mapping Hardware





## Effective Access Time

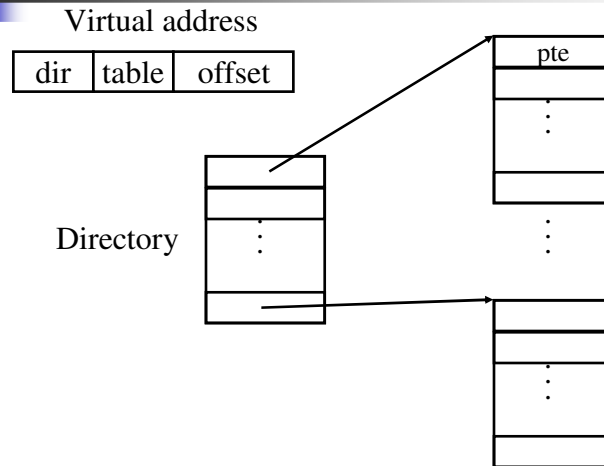
- TLB lookup time =  $\sigma$  time unit
- Memory cycle =  $m \mu s$
- TLB Hit ratio =  $\eta$
- Effective access time
  - $Eat = (m + \sigma) \eta + (2m + \sigma)(1 - \eta)$
  - $Eat = 2m + \sigma - m\eta$



## Multilevel Page Tables

- Since the page table can be very large, one solution is to page the page table
- Divide the page number into
  - An index into a table of second level page tables
  - A page within a second level page table
- Advantage
  - No need to keep all the page tables in memory all the time
  - Only recently accessed memory's mapping need to be kept in memory, rest can be fetched on demand

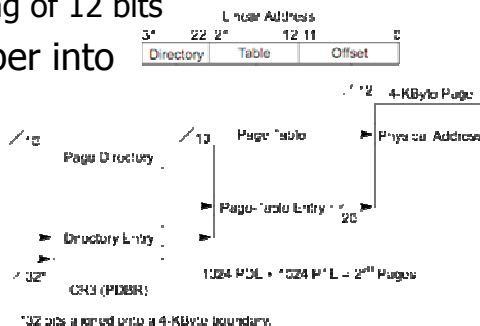
## Multilevel Page Tables



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

## Example Addressing on a Multilevel Page Table System

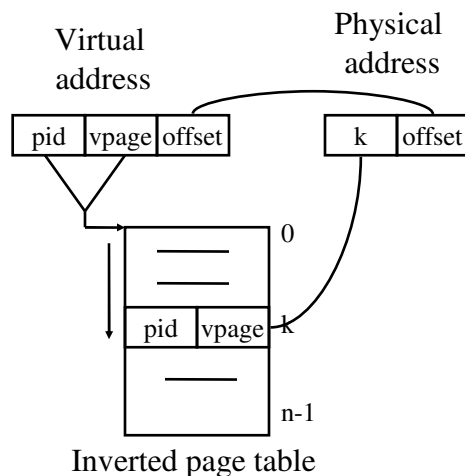
- A logical address (on 32-bit x86 with 4k page size) is divided into
  - A page number consisting of 20 bits
  - A page offset consisting of 12 bits
- Divide the page number into
  - A 10-bit page number
  - A 10-bit page offset



## 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



- Main idea
  - One PTE for each physical page frame
  - Hash (Vpage, pid) to Ppage#
  - Trade off space for time
- Pros
  - Small page table for large address space
- Cons
  - Lookup is difficult
  - Overhead of managing hash chains, etc



## Paging

---

- 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