INDIAN INSTITUTE OF INFORMATION TECHNOLOGY

Dharwad
ज्ञानेन विकासः

# Memory management and File systems


Dr. Animesh Chaturvedi

Assistant Professor: IIIT Dharwad

Young Researcher: Heidelberg Laureate Forum

Postdoc: King's College London & The Alan Turing Institute

PhD: IIT Indore MTech: IIITDM Jabalpur


HEIDELBERG LAUREATE FORUM

KING'S College LONDON

Indian Institute of Technology Indore
भारतीय प्रौद्योगिकी संस्थान इंदौर

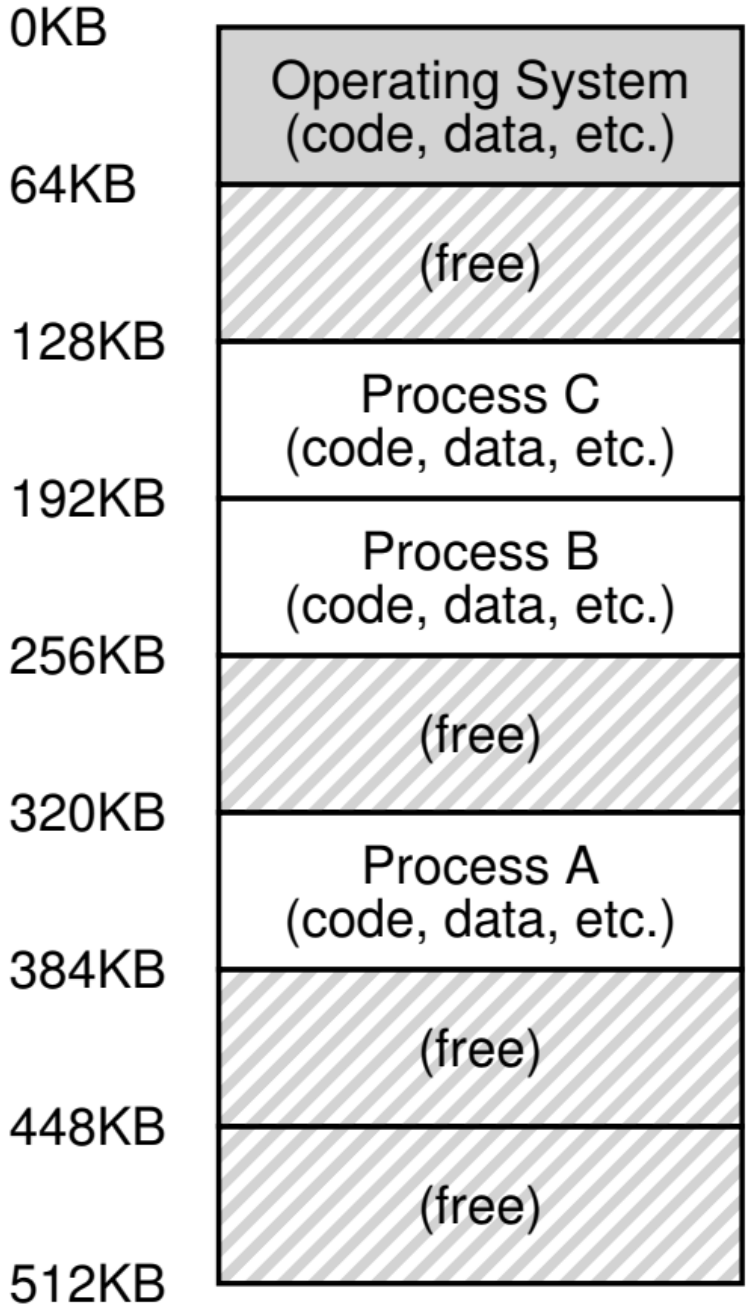
The Alan Turing Institute

PDPM
Indian Institute of Information Technology, Design and Manufacturing, Jabalpur

# Memory management (Virtual memory and Paging)

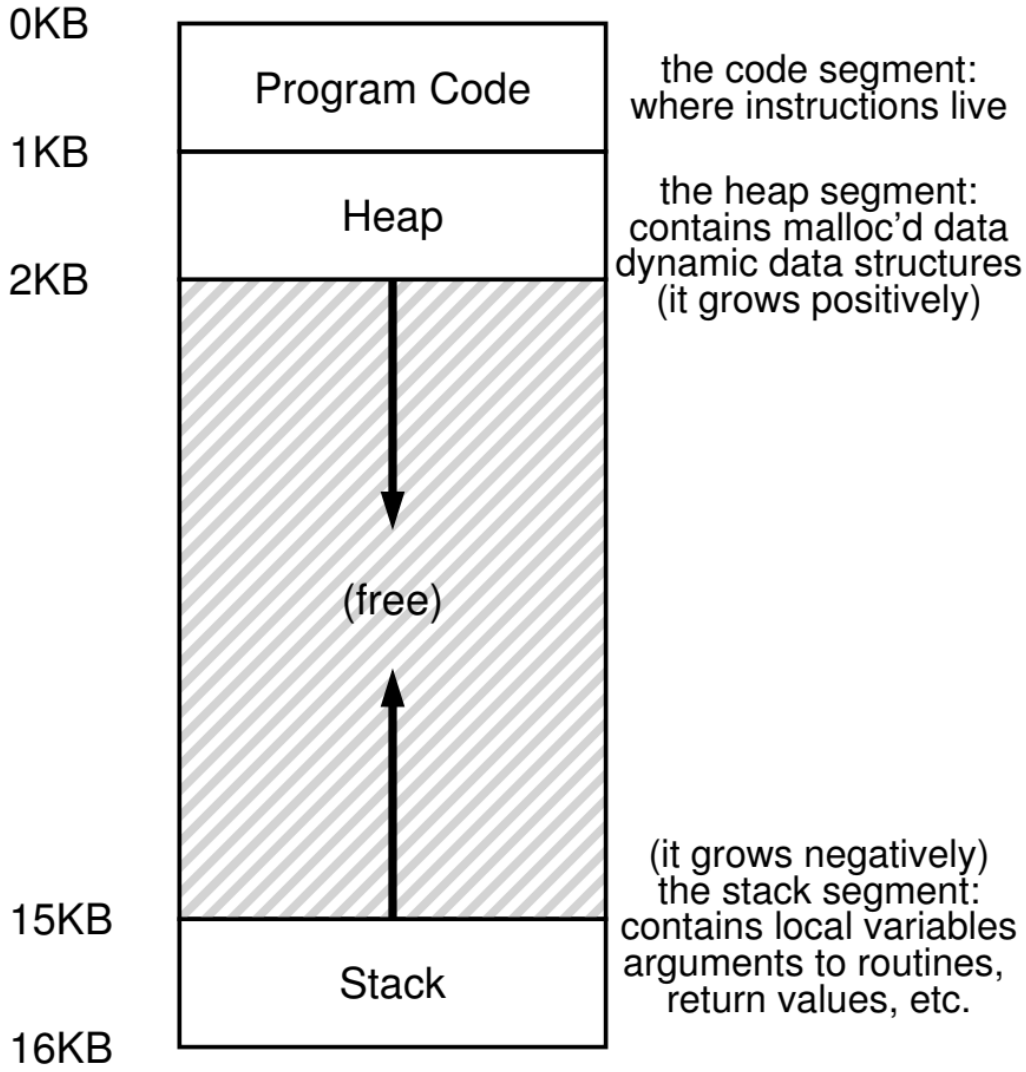
# Why virtualize memory?

- Because real view of memory is messy!
- Earlier, memory had only code of one running process (and OS code)
- Now, multiple active processes timeshare CPU
  - Memory of many processes must be in memory
  - Non-contiguous too
- Need to hide this complexity from user
- Image of Three Processes in Sharing Memory



# Abstraction: (Virtual) Address Space

- **Virtual address space**: every process assumes it has access to a large space of memory from address 0 to a MAX
- Contains program code (and static data), **heap (dynamic allocations)**, and **stack (used during function calls)**
- **Stack** and **Heap** grow during runtime
- CPU issues loads and stores to virtual addresses

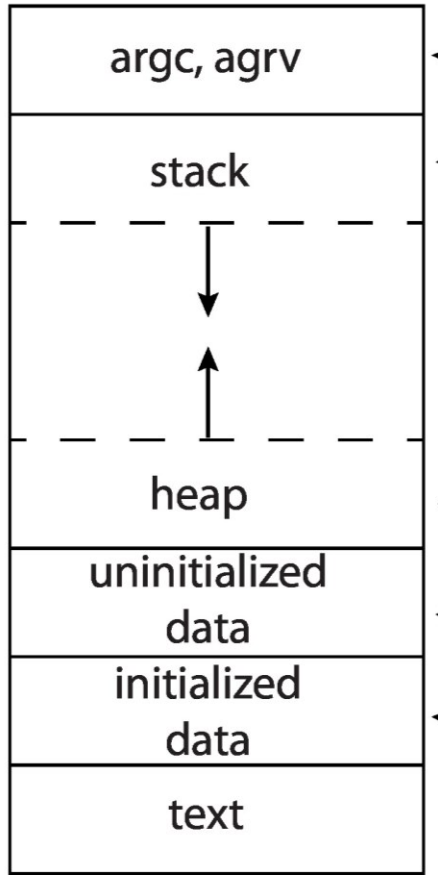


# Goals of memory virtualization

- **Transparency**: user programs should not be aware of the messy details
- **Efficiency**: minimize overhead and wastage in terms of memory space and access time
- **Isolation and protection**: a user process should not be able to access anything outside its address space
- Address translation from virtual addresses (VA) to physical addresses (PA)
  - CPU issues loads/stores to VA but memory hardware accesses PA
- OS allocates memory and tracks location of processes

# Motivation: Program Code to Memory

- **Abstraction** of complex usage Program as a Memory (RAM or Cache).
- Conversion of **High level language to Low level language**
- Static/global variables are allocated in the executable
  - Local variables of a function on Stack
  - Dynamic allocation with malloc on the heap




Process as a Memory

```
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;

int main(int argc, char *argv[])
{
    int *values;
    int i;

    values = (int *)malloc(sizeof(int)*5);

    for(i = 0; i < 5; i++)
        values[i] = i;

    return 0;
}
```

Program as a Code

# Address Translation

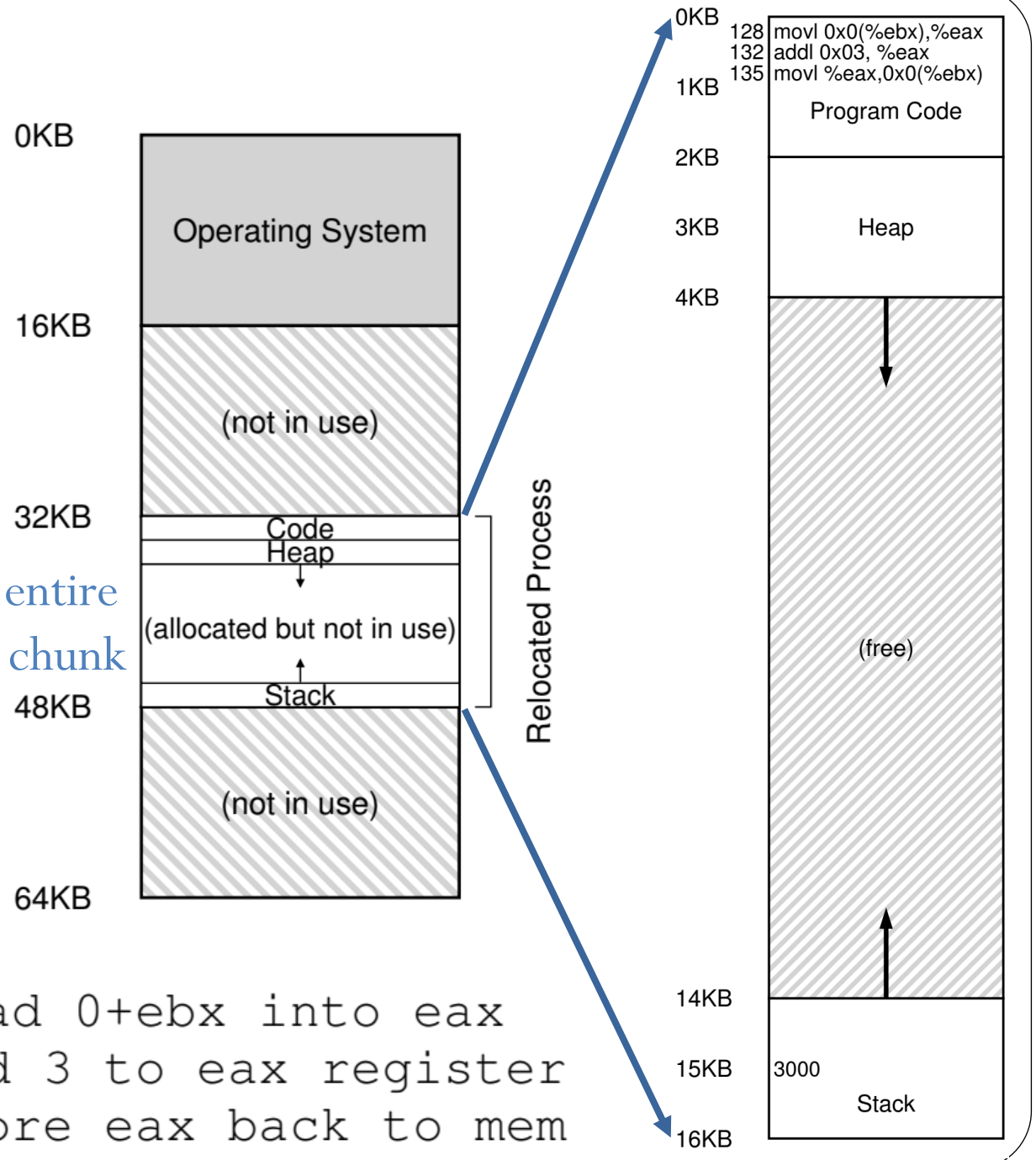
- Virtual address space is setup by OS during process creation
- Translation from **VA to PA**
  - 128 to 32896 (32KB + 128)
  - 1KB to 33 KB
  - 20KB? Error!

Simplified OS: places entire memory image in one chunk

```
void func() {
    int x = 3000;
    x = x + 3;
    ...
```

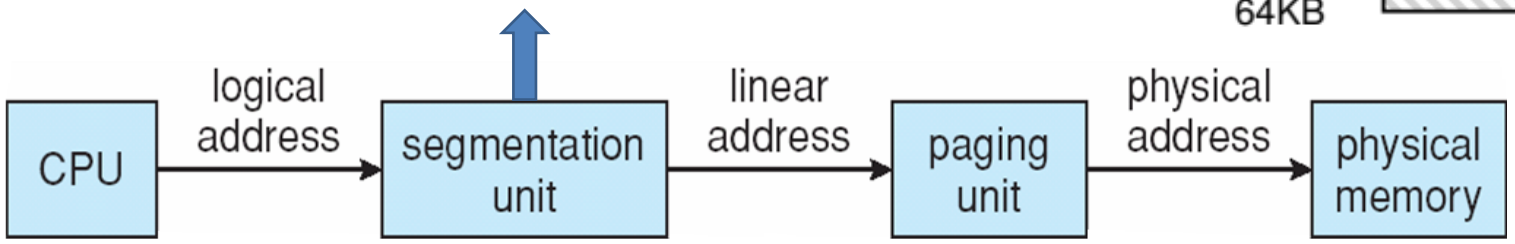
Compiler

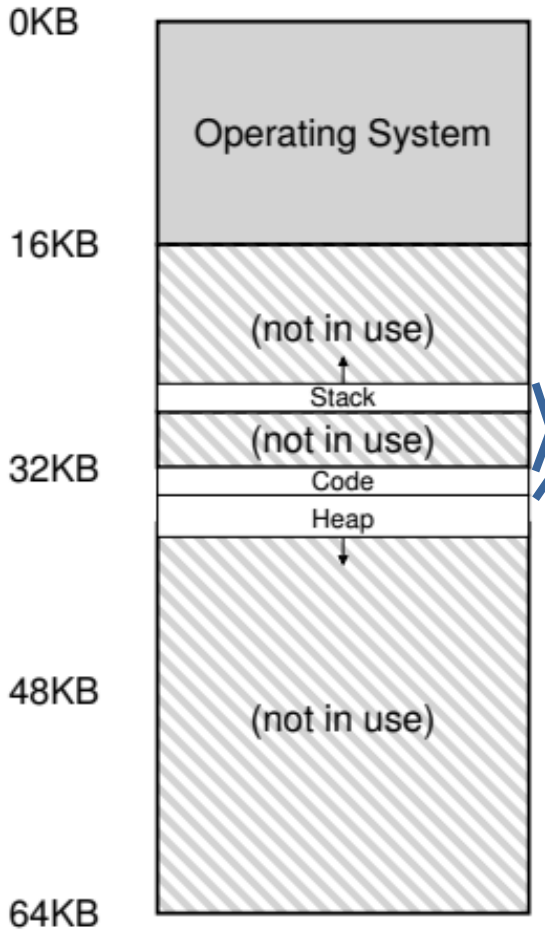
```
128: movl 0x0(%ebx), %eax     ;load 0+ebx into eax
132: addl $0x03, %eax         ;add 3 to eax register
135: movl %eax, 0x0(%ebx)     ;store eax back to mem
```

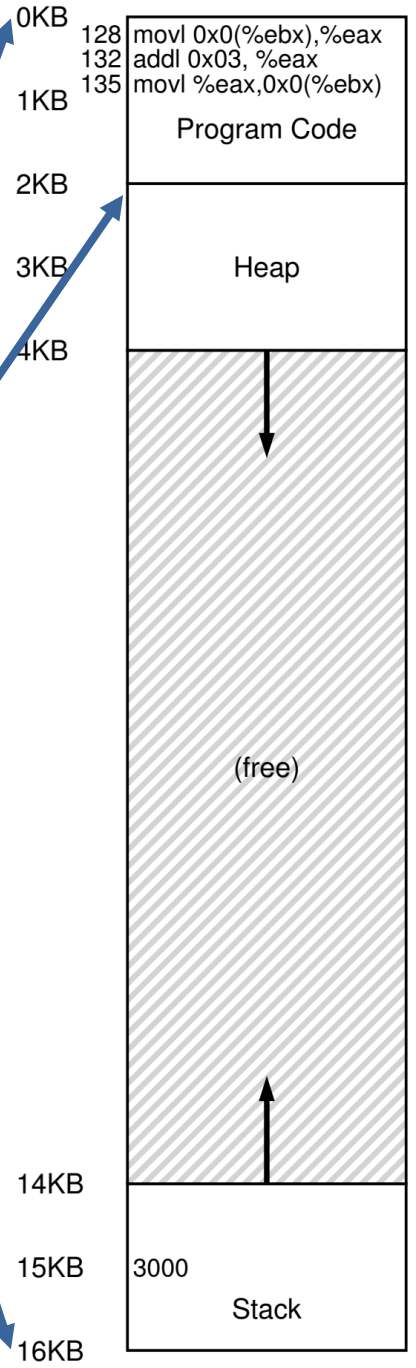


# Segmentation

- Generalized base and bounds
- Each segment of memory image placed separately
- Multiple (base, bound) values stored in MMU
- Good for sparse address spaces
- But variable sized allocation leads to external fragmentation – Small holes in memory left between segments



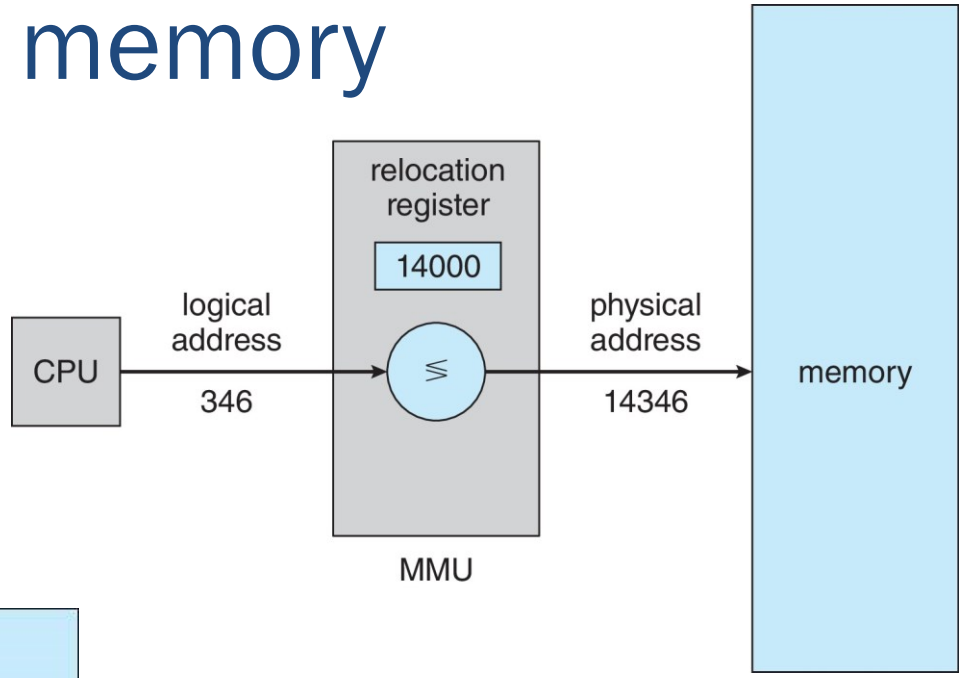



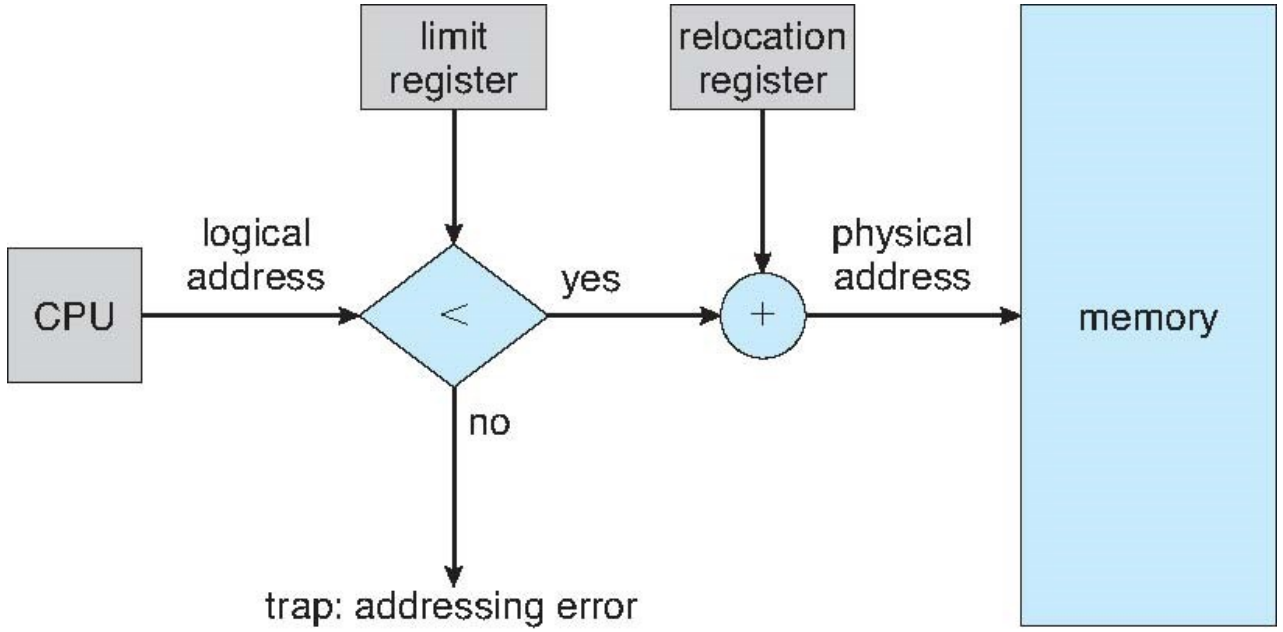



# Physical memory and Virtual memory

- OS provides the **base** (starting address) and **bound** (total size of process) values to
  - **Memory Management Unit (MMU)** (a hardware)
- Memory hardware MMU calculates PA from VA

  Physical Address (PA) = Virtual Address (VA) + Base



# MMU hardware in translation

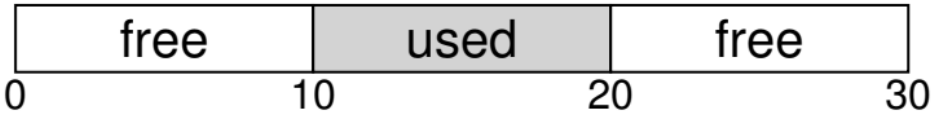
- Who performs address translation? Memory Management Unit (MMU)
- MMU also checks if address is beyond bound
- OS is not involved in every translation
- CPU provides privileged mode of execution
- Instruction set has privileged instructions to set translation information (e.g., base, bound)
- Hardware (MMU) uses this information to perform translation on every memory access
- MMU generates faults and traps to OS when access is illegal (e.g., VA is out of bound)

# OS software codes in translation

- OS maintains free list of memory
- Allocates space to process during creation (and when asked) and cleans up when done
- Maintains information of where space is allocated to each process (in PCB)
- Sets address translation information (e.g., base & bound) in hardware
- Updates this information upon context switch
- Handles traps due to illegal memory access

# Memory Allocation Strategies and Paging

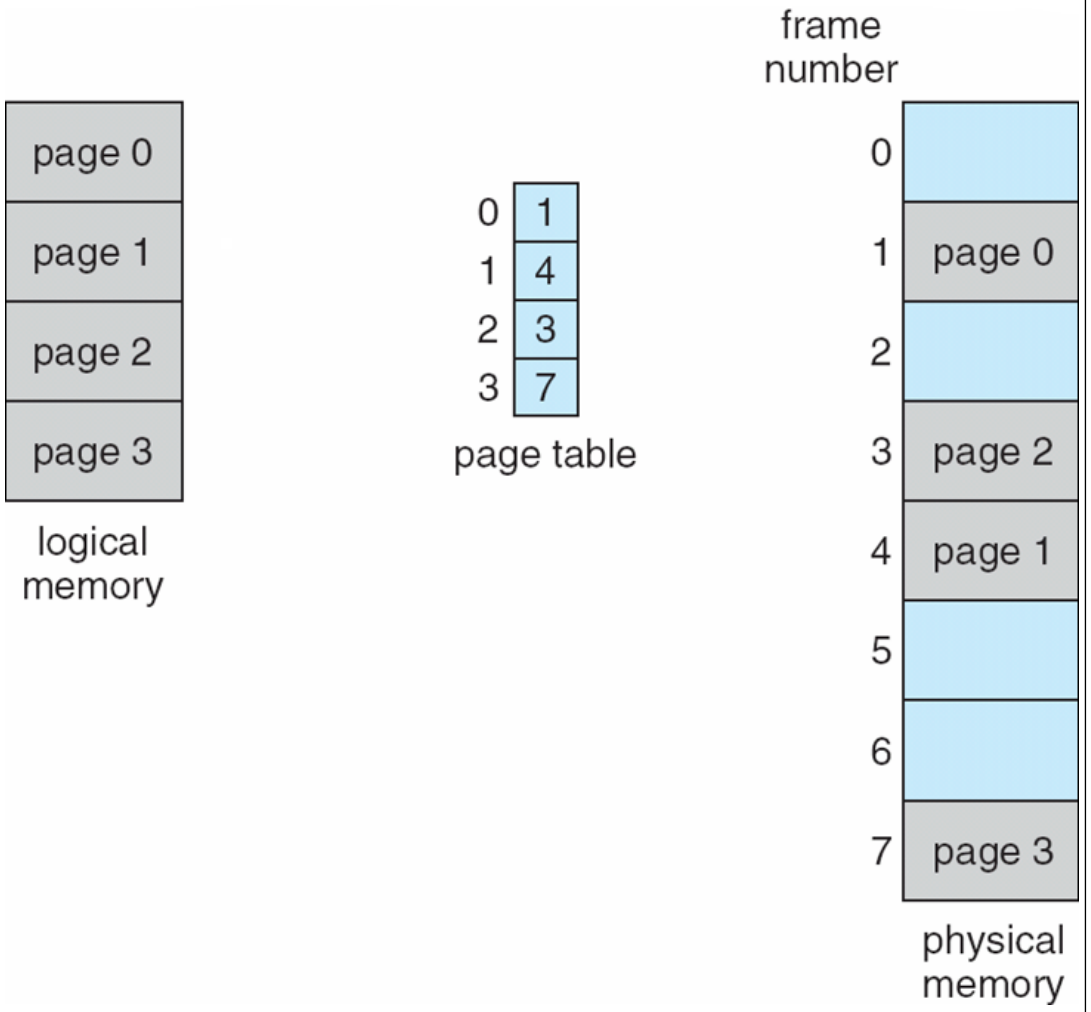
- Allocate memory in fixed size "**blocks**" or "**chunks**" ("**pages**")
- OS allocates a set of pages to the memory image of the process
- Variable Size Allocation Strategies
  - **First fit**: finds the first block and allocate the first free chunk that is sufficient
  - **Best fit**: search and allocate free chunk that is closest in size (smallest fit)
  - **Worst fit**: search and allocate free chunk that is farthest in size
- Paging avoids **External Fragmentation** (no small "holes")
  - the free space gets fragmented into little pieces of different sizes;
  - subsequent requests may fail because there is no single contiguous space that can satisfy the request, even though the total amount of free space exceeds the size of the request.




- Paging suffers from **Internal Fragmentation** (partially filled pages)

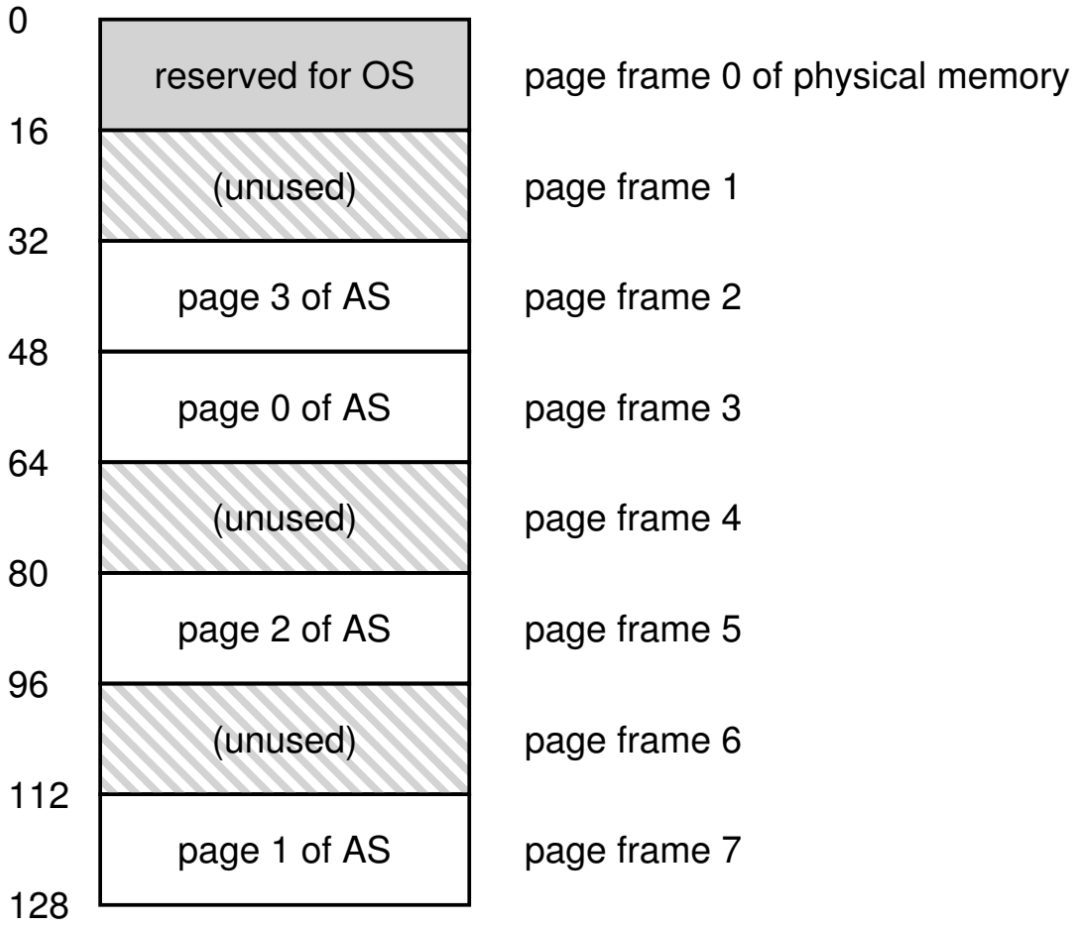
# Paging, Logical memory to Frame number

- OS divides virtual address space into **fixed size pages**, physical memory into **frames**
- To allocate memory, a page is mapped to a free physical frame
- **Page table** stores mappings from virtual page number to physical frame number for a process (e.g., page 0 to frame 3)
- MMU has access to page tables, and uses it to translate VA to PA



# Page Table

- Per process data structure to help VA-PA translation.
- Array stores mappings
  - from **Virtual Page Number (VPN)** to **Physical Frame Number (PFN)**
  - E.g., VPN 0 → PFN 3, VPN 1 → PFN 7
- Part of OS memory (in PCB)
- MMU has access to page table and uses it for address translation
- OS updates page table upon context switch



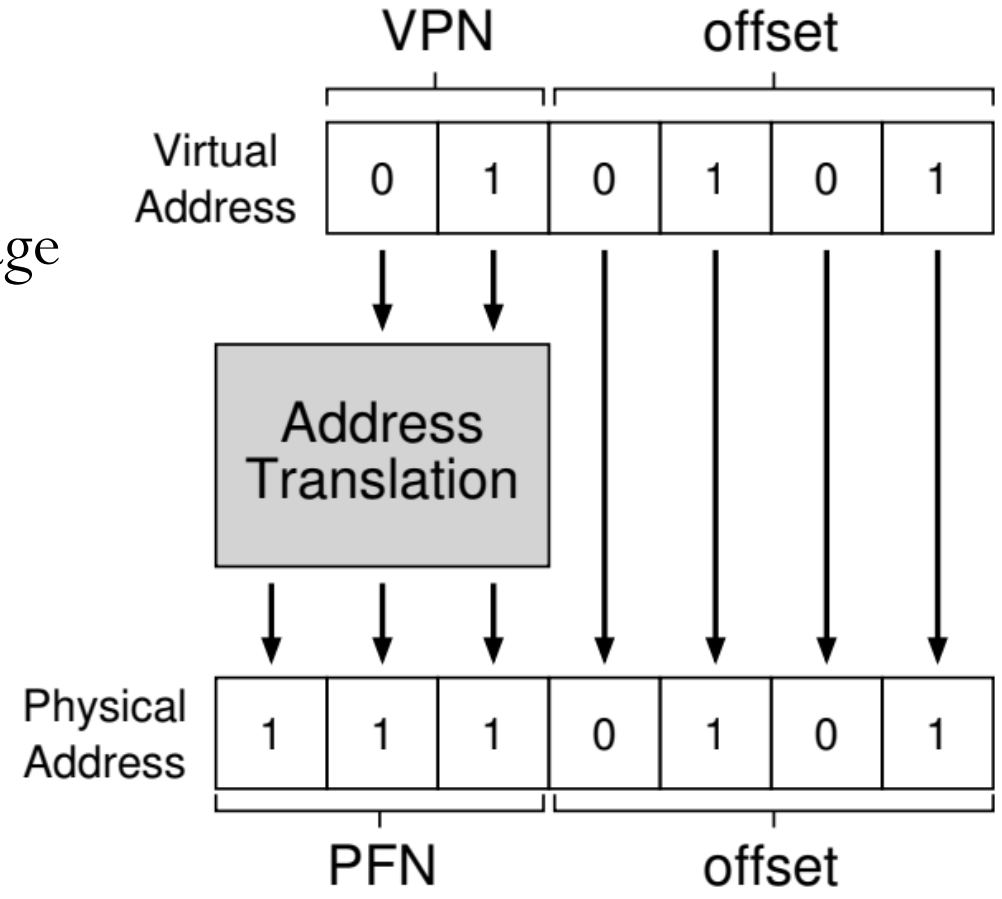

a 64-Byte address space in a 128-Byte physical memory

# Page Table Entry (PTE)

- Simplest page table: linear page table
- Page table is an array of page table entries, one per virtual page
- VPN (**virtual page number**) is index into this array
- Each PTE contains PFN (**physical frame number**) and few other bits
  - **Valid bit**: is this page used by process?
  - **Protection bits**: read/write permissions
  - **Present bit**: is this page in memory? (more later)
  - **Dirty bit**: has this page been modified?
  - **Accessed bit**: has this page been recently accessed?

# Address translation in hardware

- Most significant bits of VA give the VPN
- Page table maps VPN to PFN
- PA is obtained from PFN and offset within a page
- MMU stores (physical) address of start of page table, not all entries.
- **"Walks" the page table** to get relevant PTE

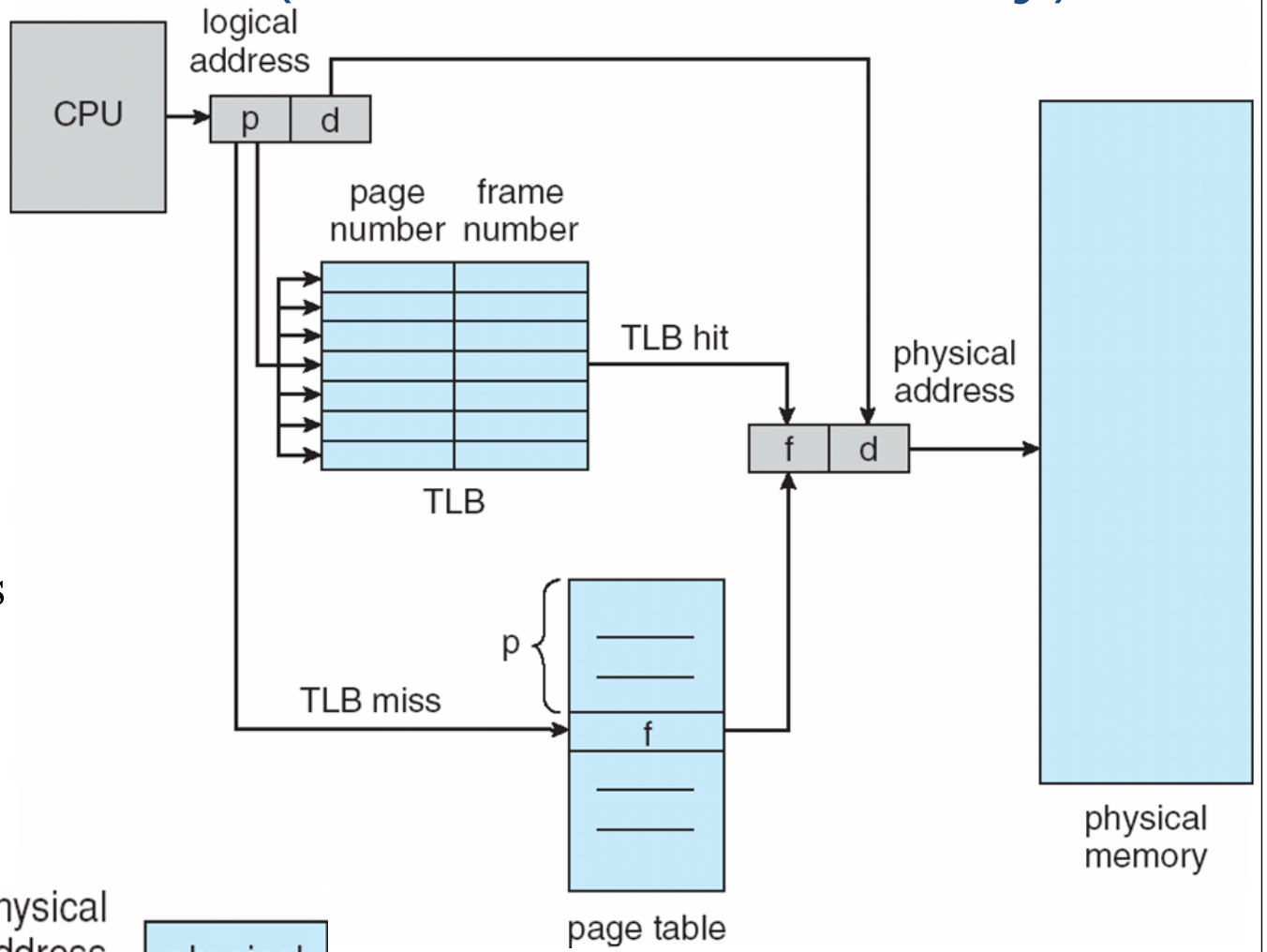


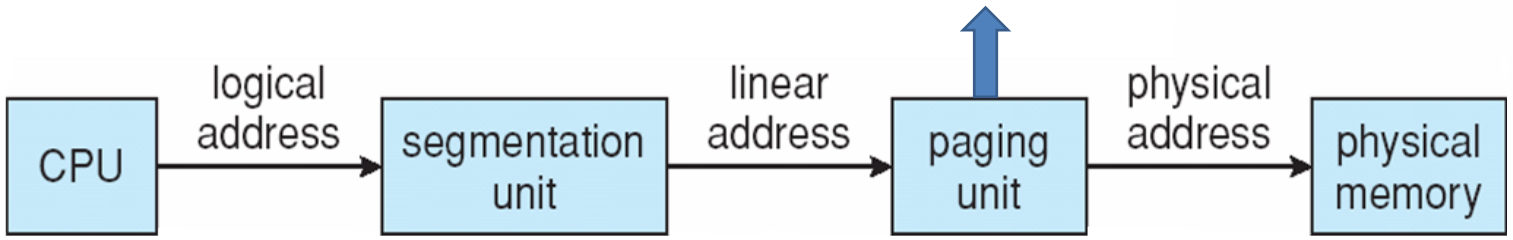
# What happens on memory access?

- CPU requests code or data at a virtual address
- MMU must translate VA to PA
  - First, access memory to read page table entry
  - Translate VA to PA
  - Then, access memory to fetch code/data
- Paging adds overhead to memory access
- Solution? A cache for **VA-PA mappings**.

# Translation Look-Aside Buffer (Associative memory)

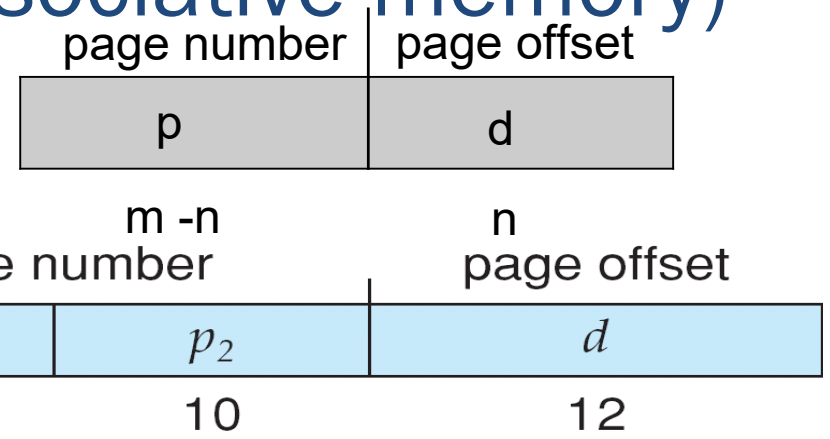
- A cache of recent VA-PA mappings
- To translate VA to PA, MMU first looks up TLB
- If **TLB hit**, obtaining PA, fetches memory location and returns to CPU (via CPU caches)
- If **TLB miss**, then MMU performs additional memory accesses to "walk" page table



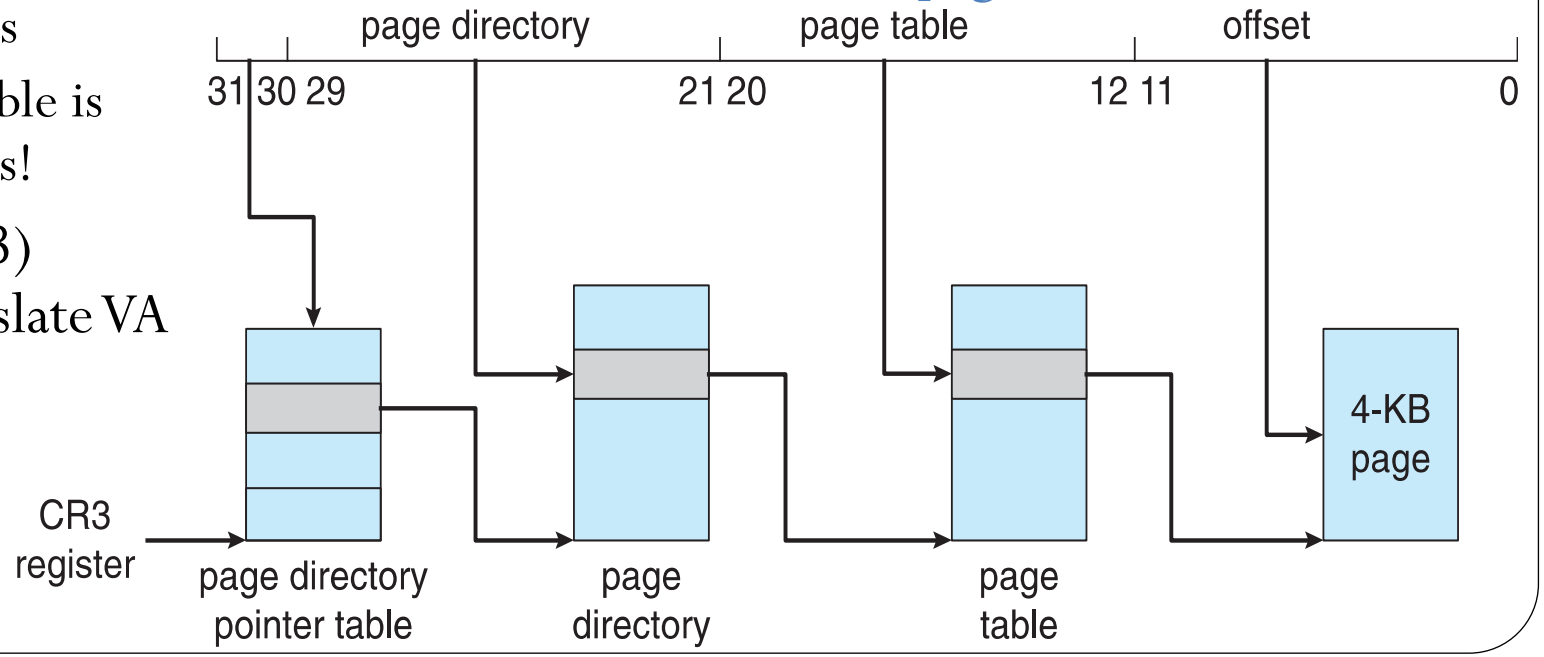



# Translation Look-Aside Buffer (Associative memory)

- 32 bit VA, 4 KB pages,
  - 2^32 / 2^12 = 2^20 entries
  - If each PTE is 4 bytes, then page table is **4MB**
  - One such page table per process!
- To reduce the size of page tables
  - Larger pages, so fewer entries
  - For such large tables, Page table is itself split into smaller chunks!
- **Control register** (e. g. CR3) enables the processor to translate VA into PA by locating the page directory and page tables



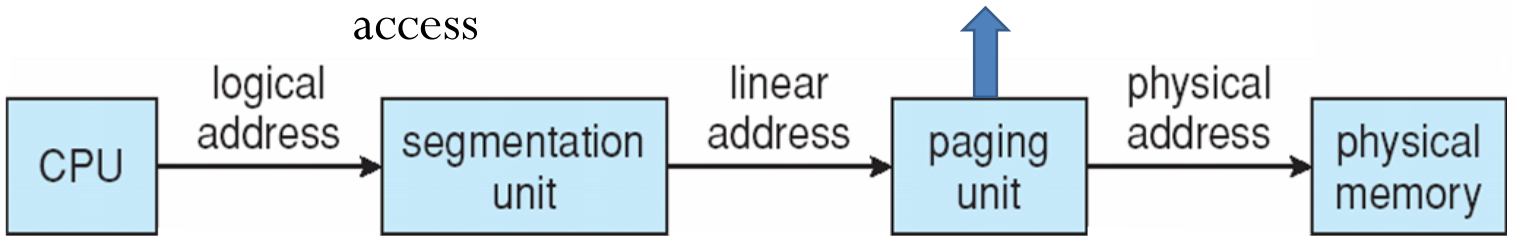

Multilevel page tables

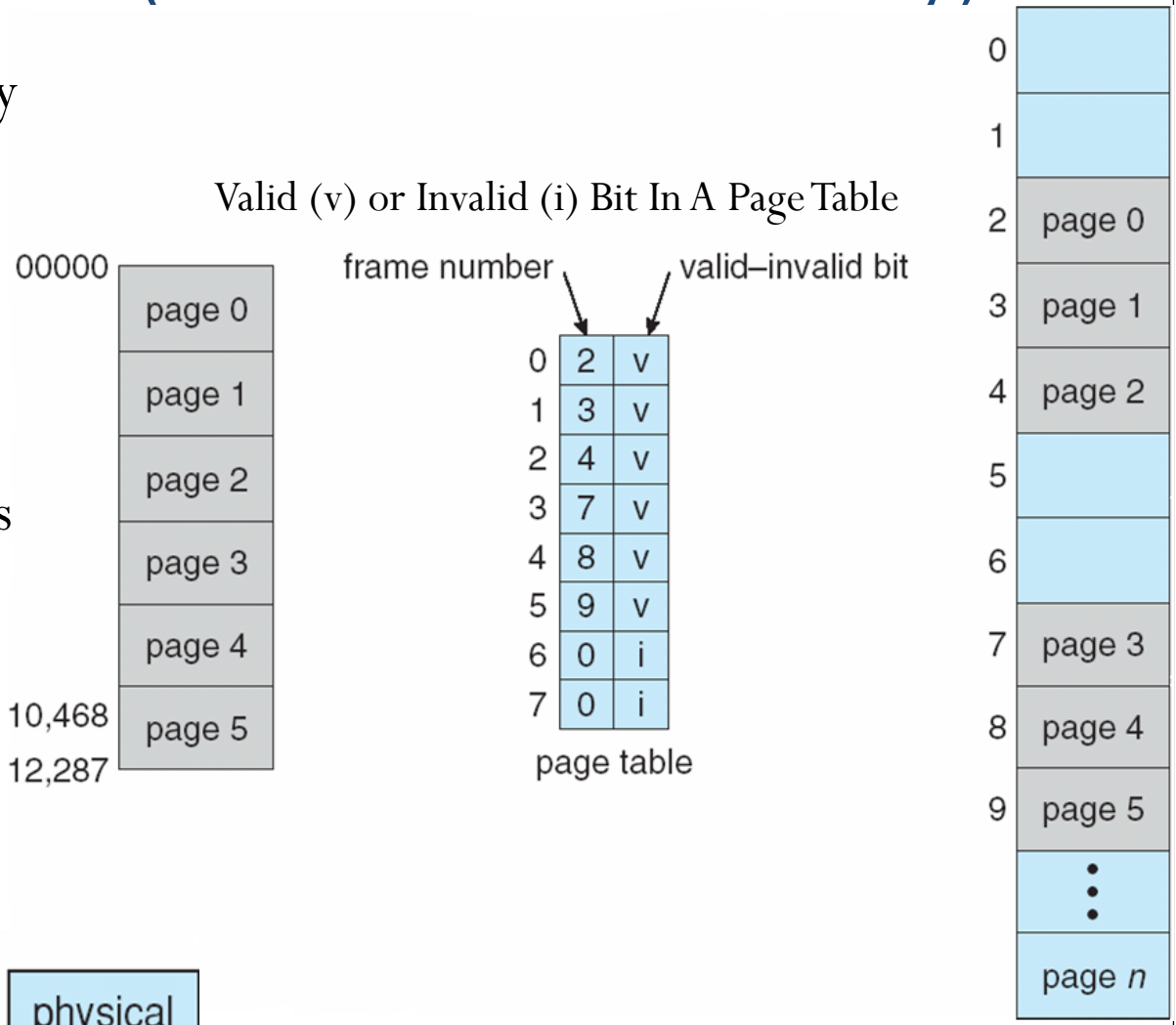



Intel Architecture IA-32 bit

# Translation Look-Aside Buffer (Associative memory)

- TLB misses are expensive (multiple memory accesses)
- **Locality of reference** helps to have high **hit rate**
- If **TLB hit**: PA can be directly used
- If **TLB miss**: MMU accesses memory, walks page table, and obtains page table entry
  - If present bit set in PTE, accesses memory
  - If not present but valid, raises **page fault**.
  - If invalid page access, trap to OS for illegal access



# Page fault

- Are all pages of all active processes always in main memory?
  - Not necessary, with large address spaces
- All active processes are not necessary always in main memory
- When translating VA to PA, MMU reads present bit
  - If page present in memory, directly accessed
  - If page not in memory, MMU raises a trap to the OS
    - page fault
- **Page fault:** Present bit in page table entry
  - indicates if a page of a process resides in memory or not

# Demand Paging (Page fault handling)

- Page fault traps OS and moves CPU to kernel mode
- OS fetches disk address of page and issues read to disk
  - OS keeps track of disk address (say, in page table)
- OS context switches to another process
  - Current process is blocked and cannot run
- When disk read completes, OS updates page table of process, and marks it as ready
- When process scheduled again, OS restarts the instruction that caused page fault

# Swapping

- When servicing page fault, if there is no free page to **swap-in** the faulting page.
- OS uses a part of disk (**swap space**) to store pages that are not in active use
- OS must **swap-out** an existing page and then swap in the faulting page.
- OS may proactively **swap-out** pages to keep list of free pages handy.
- Page replacement policy decides which pages to **swap-out**
- Page replacement policies
  - **Optimal**: replace page not needed for longest time in future (not practical!)
  - **FIFO**: replace page that was brought into memory earliest (may be a popular page!)
  - **LRU/LFU**: replace the page that was least recently (or frequently) used in the past

# Optimal Page replacement Policy

- Example: 3 frames for 4 pages (0,1,2,3)
- First few accesses are cold (compulsory) misses

- **Hit rate** $= \dfrac{\text{Hits}}{\text{Hits} + \text{Misses}}$

- 6 hits and 5 misses,

$$\text{Hit rate} = \frac{6}{6+5} \times 100 = 54.5\%$$

| Access | Hit/Miss? | Evict | Resulting Cache State |
|---|---|---|---|
| 0 | Miss | | 0 |
| 1 | Miss | | 0, 1 |
| 2 | Miss | | 0, 1, 2 |
| 0 | Hit | | 0, 1, 2 |
| 1 | Hit | | 0, 1, 2 |
| 3 | Miss | 2 | 0, 1, 3 |
| 0 | Hit | | 0, 1, 3 |
| 3 | Hit | | 0, 1, 3 |
| 1 | Hit | | 0, 1, 3 |
| 2 | Miss | 3 | 0, 1, 2 |
| 1 | Hit | | 0, 1, 2 |

# FIFO Page replacement Policy

- Usually worse than optimal
- **Belady's anomaly**: performance may get worse when memory size increases.
- 4 hits and 7 misses,

$$\text{Hit rate} = \frac{4}{4+7} \times 100$$
$$= 36.4\% \text{ hit rate}$$

| Access | Hit/Miss? | Evict | Resulting Cache State |
|---|---|---|---|
| 0 | Miss | | First-in→ 0 |
| 1 | Miss | | First-in→ 0, 1 |
| 2 | Miss | | First-in→ 0, 1, 2 |
| 0 | Hit | | First-in→ 0, 1, 2 |
| 1 | Hit | | First-in→ 0, 1, 2 |
| 3 | Miss | 0 | First-in→ 1, 2, 3 |
| 0 | Miss | 1 | First-in→ 2, 3, 0 |
| 3 | Hit | | First-in→ 2, 3, 0 |
| 1 | Miss | 2 | First-in→ 3, 0, 1 |
| 2 | Miss | 3 | First-in→ 0, 1, 2 |
| 1 | Hit | | First-in→ 0, 1, 2 |

# LRU Page replacement Policy

- Works well due to locality of references
- Spatial locality: if a page P is accessed, it is likely the pages around it
  - (say P − 1 or P + 1) will also likely be accessed.
- Temporal locality: if a page P have been accessed in the near past is likely to be accessed again in the near future.
- Equivalent to optimal in this example
  - 6 hits and 5 misses; Hit rate = 54.5%
- Opposites of these algorithms exist:
  - Most Frequently-Used (MFU) and
  - Most-Recently-Used (MRU).
  - these policies do not work well,
  - as they ignore the locality of reference

| Access | Hit/Miss? | Evict | Resulting Cache State |
|---|---|---|---|
| 0 | Miss | | LRU→ 0 |
| 1 | Miss | | LRU→ 0, 1 |
| 2 | Miss | | LRU→ 0, 1, 2 |
| 0 | Hit | | LRU→ 1, 2, 0 |
| 1 | Hit | | LRU→ 2, 0, 1 |
| 3 | Miss | 2 | LRU→ 0, 1, 3 |
| 0 | Hit | | LRU→ 1, 3, 0 |
| 3 | Hit | | LRU→ 1, 0, 3 |
| 1 | Hit | | LRU→ 0, 3, 1 |
| 2 | Miss | 0 | LRU→ 3, 1, 2 |
| 1 | Hit | | LRU→ 3, 2, 1 |

# LRU implementation

- OS is not involved in every memory access
  - how does it know which page is LRU?
- Hardware help and some approximations
- MMU sets a bit in **PTE ("accessed" bit)** when a page is accessed
- OS periodically looks at this bit to estimate pages that are active and inactive
- To replace, OS tries to find a page that does not have access bit set
  - May also look for page with dirty bit not set (to avoid **swapping-out** to disk)

# Average Memory Access Time (AMAT)

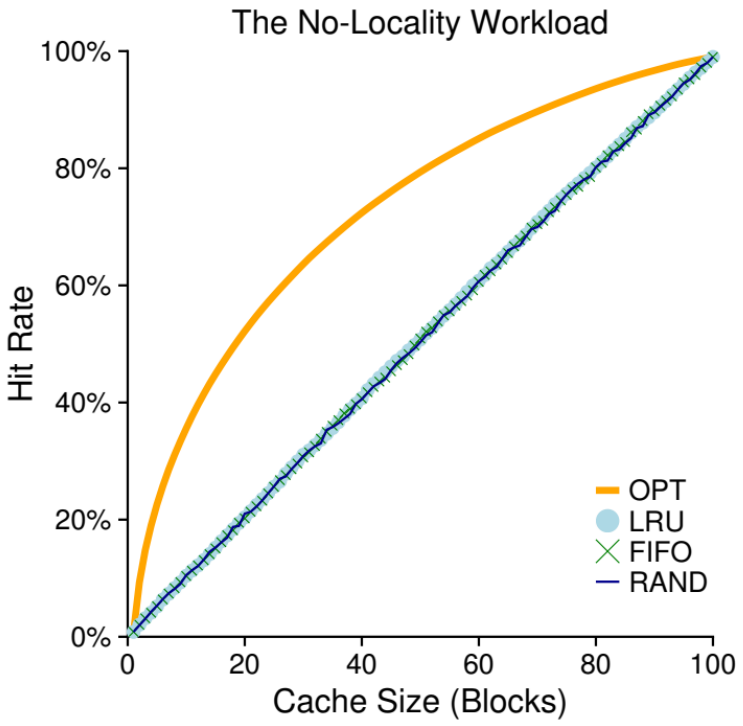
- **AMAT = TM + ($P_{miss}$ × TD)**
  - where, **TM** represents the cost of accessing memory;
  - **TD** the cost of accessing disk,
  - **$P_{Miss}$** the probability of not finding the data in the cache (a miss);
  - **$P_{Miss}$** varies from 0.0 to 1.0, and sometimes we refer to a percent miss rate instead of a probability (e.g., a 10% miss rate means **$P_{Miss}$** = 0.10).
- Assuming TM is around 100 nanoseconds, and TD is about 10 milliseconds,
- AMAT = 100ns + 0.1 × 10ms

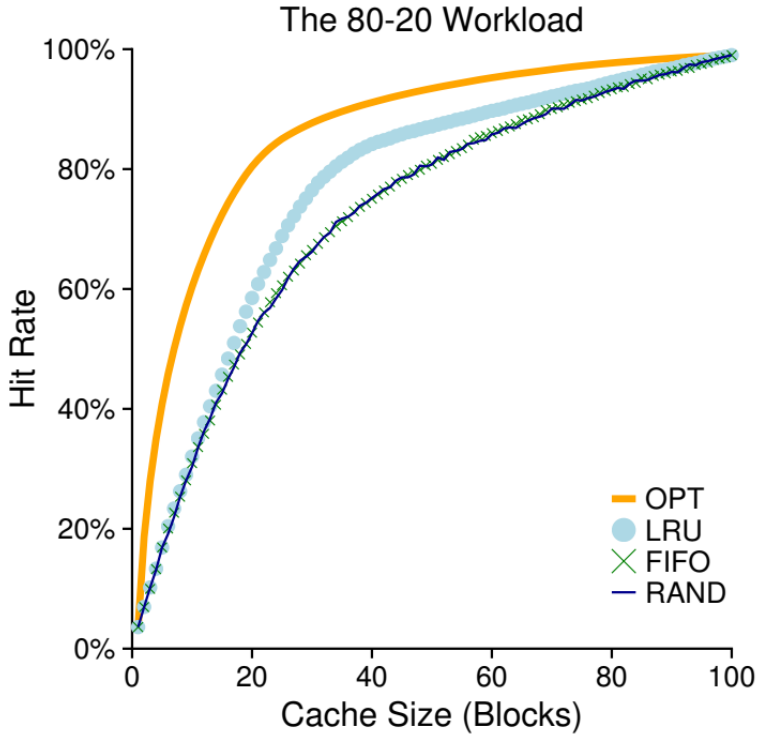
  = 100ns + 1ms = 1.0001ms

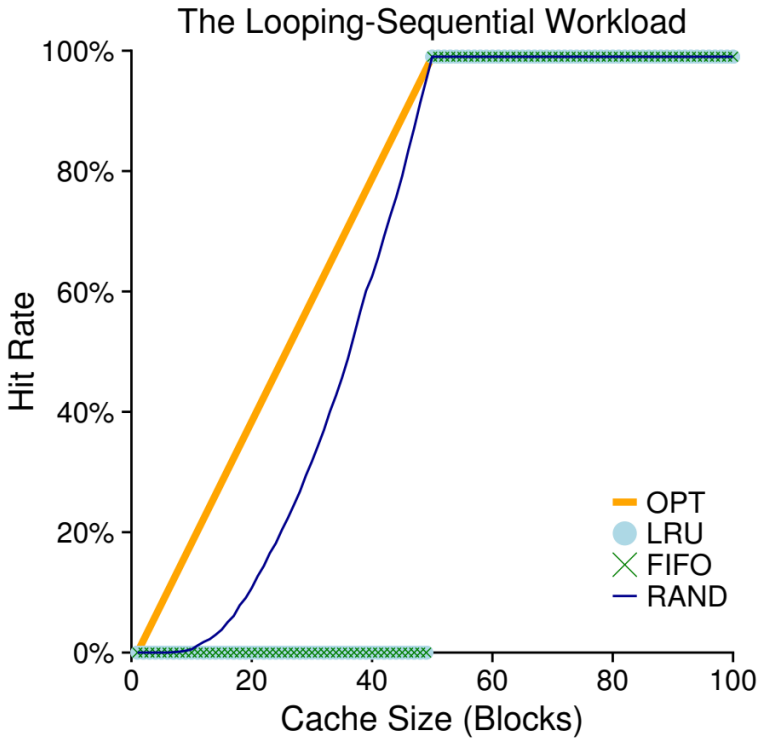
  = 1 millisecond
- If our hit rate had instead been 99.9% ($P_{Miss}$ = 0.001), the result is quite different: AMAT is 10.1 microseconds, or roughly 100 times faster.

# Workload Examples

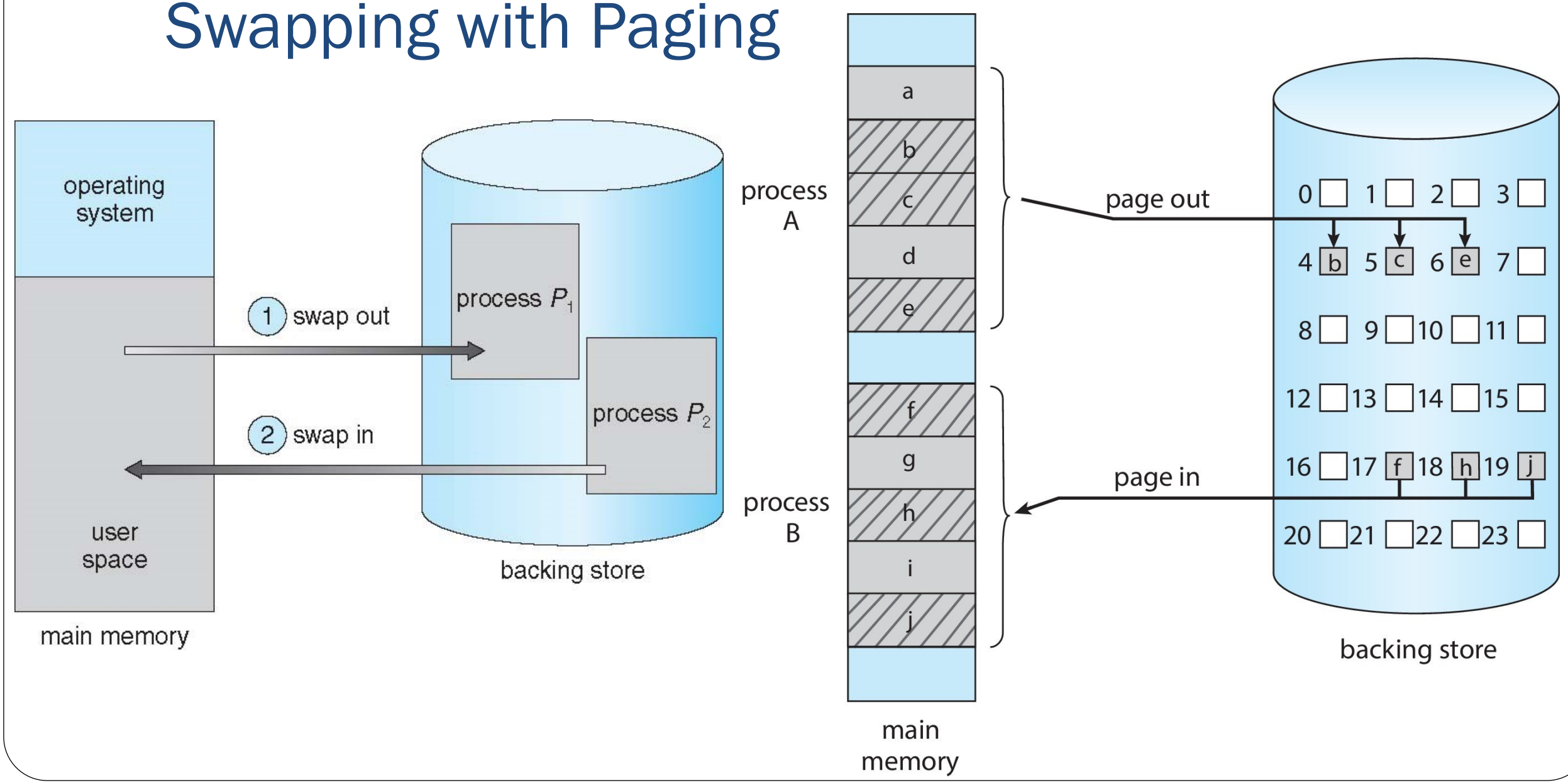
- 80% of the references are made to 20% of the **"hot" pages**; the remaining 20% of the references are made to the remaining 80% of the **"cold" pages**.
- accesses 50 unique pages in sequence, starting at 0, 1, …, 49, then loop, repeating those accesses, for a total of 10,000



# Swapping with Paging



# Summary: OS memory management

- CPU issues load to a **Virtual Address** for **Segments** of code or data:
  - Checks **CPU Cache first**
  - Goes to **Main memory** in case of cache miss
- **MMU** looks up **TLB** for VA
  - TLB hit: then read from PA
  - TLB miss: MMU accesses memory, walks page table, and obtains page table entry
    - If present bit set in PTE, accesses memory
    - If not present but valid, raises page fault.
    - If invalid page access, trap to OS for illegal access
- In case of **Page fault**, OS uses **Swapping** from Disk to Main memory
- In case of no free page in memory, OS uses **Page replacement policy**

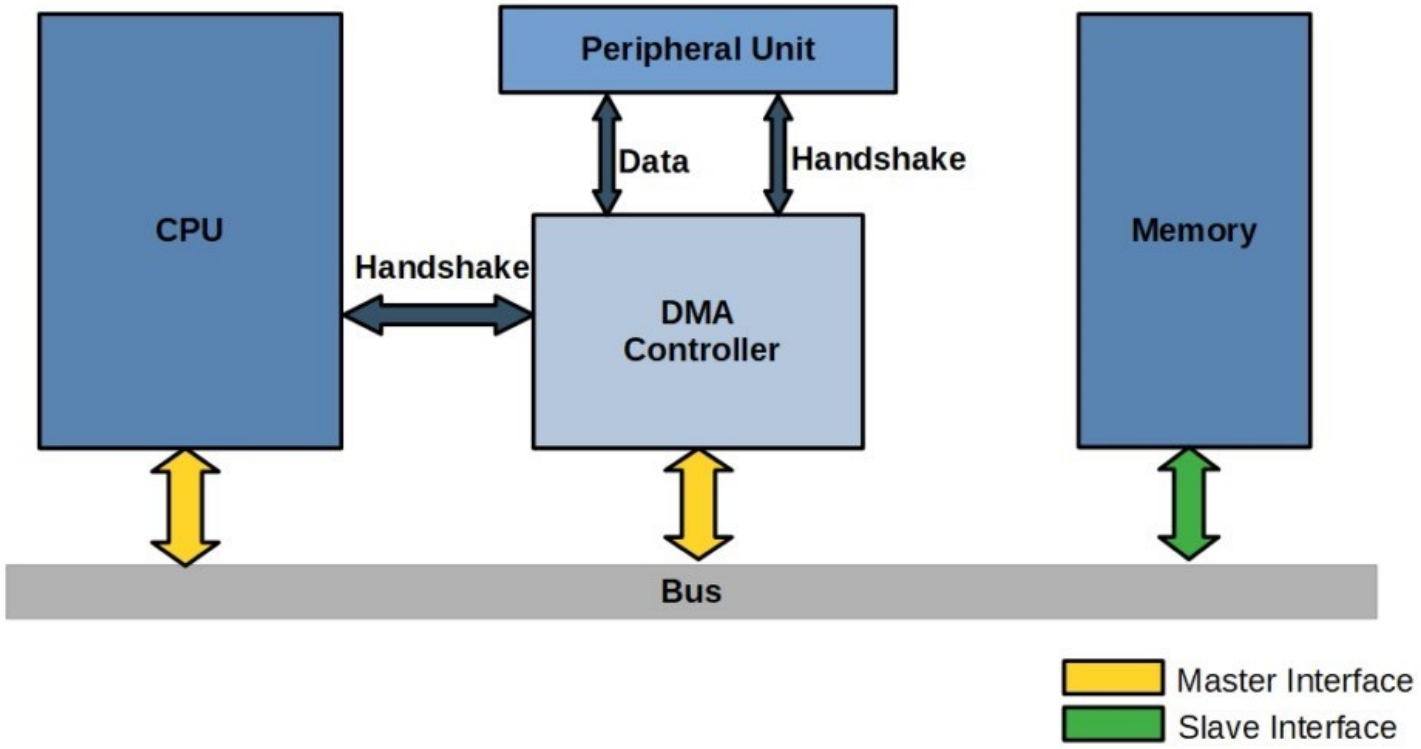
# File systems (I/O and Mass-Storage)

# File and Directory

- File – linear array of bytes, stored persistently
  - Identified with file name (human readable) and a OS-level identifier ("inode number")
  - Inode number unique within a file system
- Directory contains other subdirectories and files, along with their inode numbers
  - Stored like a file, whose contents are filename-to inode mappings
- Files and directories arranged in a tree, starting with **root** ("/")
- **System calls:** open(), read()/write(), lseek() (seek to random offset), close(), rename(), delete ()

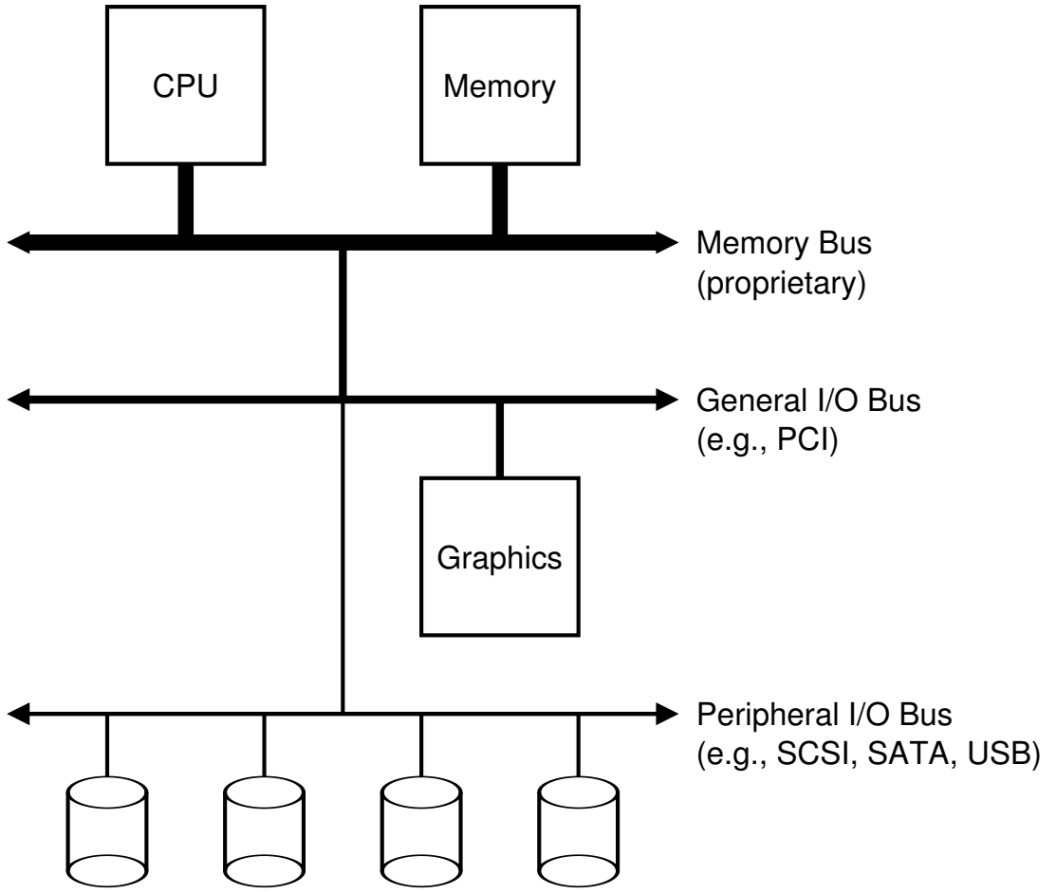
# Direct Memory Access (DMA)

- CPU cycles wasted in copying data to/from device
  - A special hardware (**DMA controller**) copies from main memory to device
  - CPU gives DMA engine the memory location of data

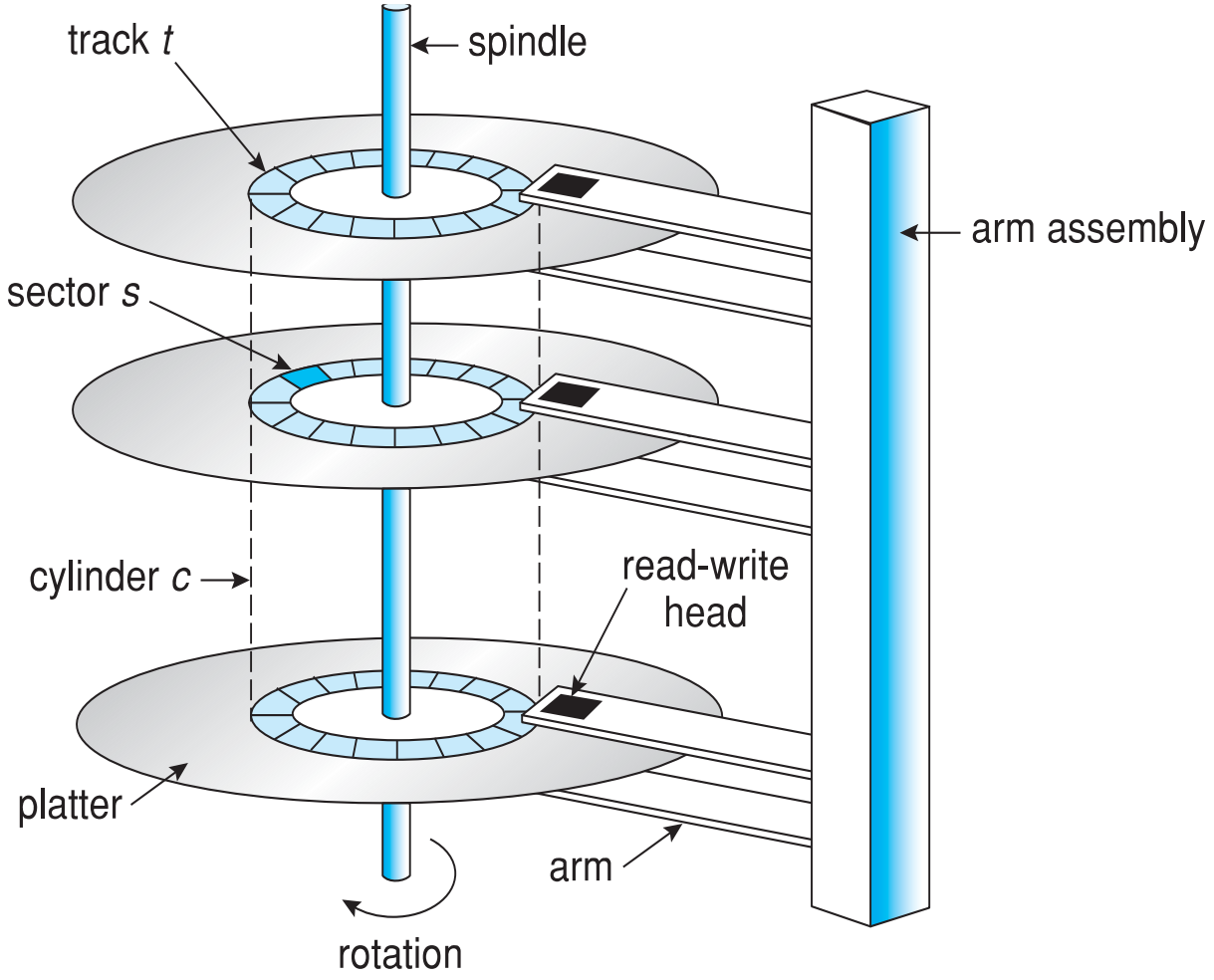


# Device Driver

- **Device driver**: part of OS code that talks to specific device, gives commands, handles interrupts etc.
- I/O devices connect to the CPU and memory via a Bus
  - High speed bus, e.g., PCI
  - Other: SCSI, USB, SATA
- Point of connection to the system: port

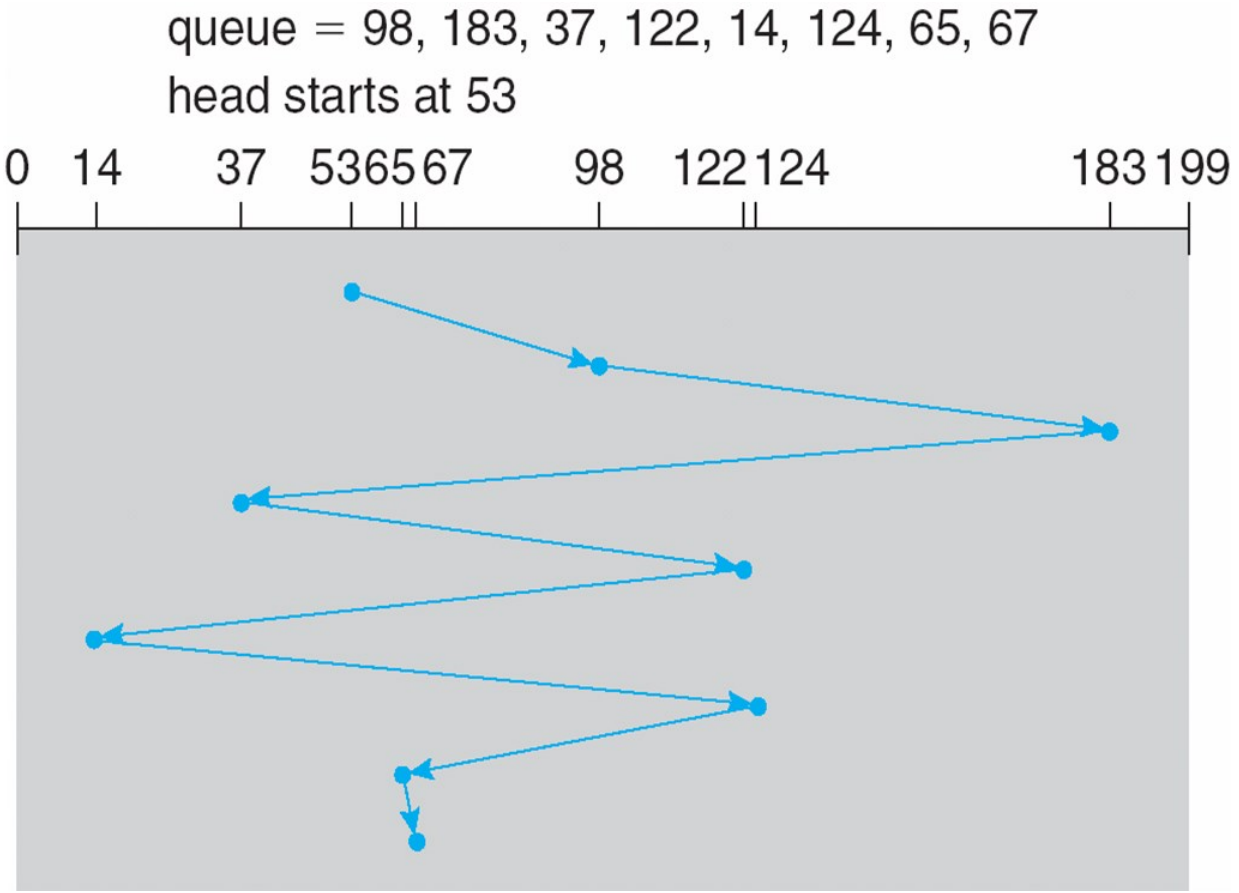


# Moving-head Disk Mechanism



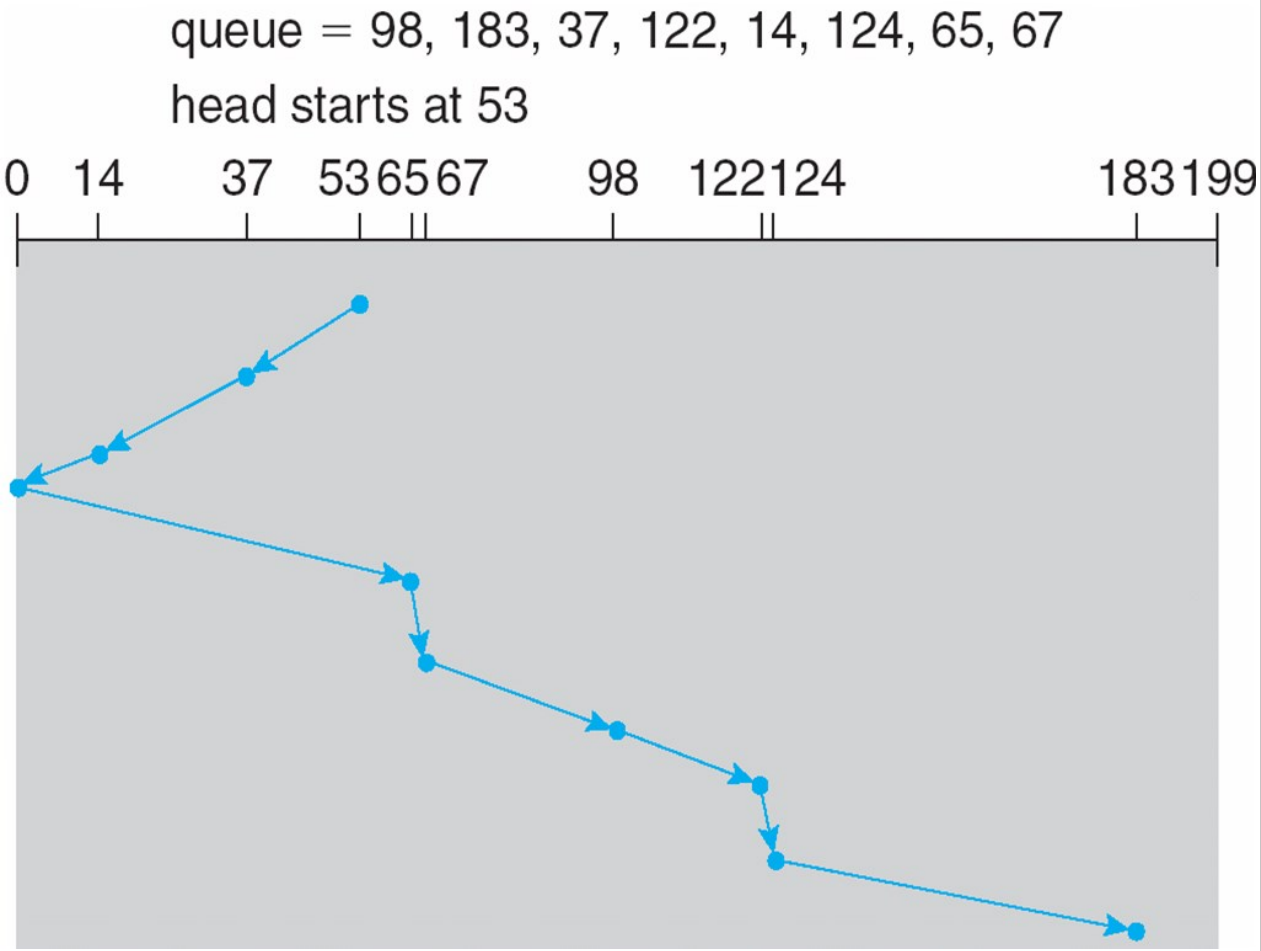
# FCFS

- Illustration shows total head movement of 640 cylinders



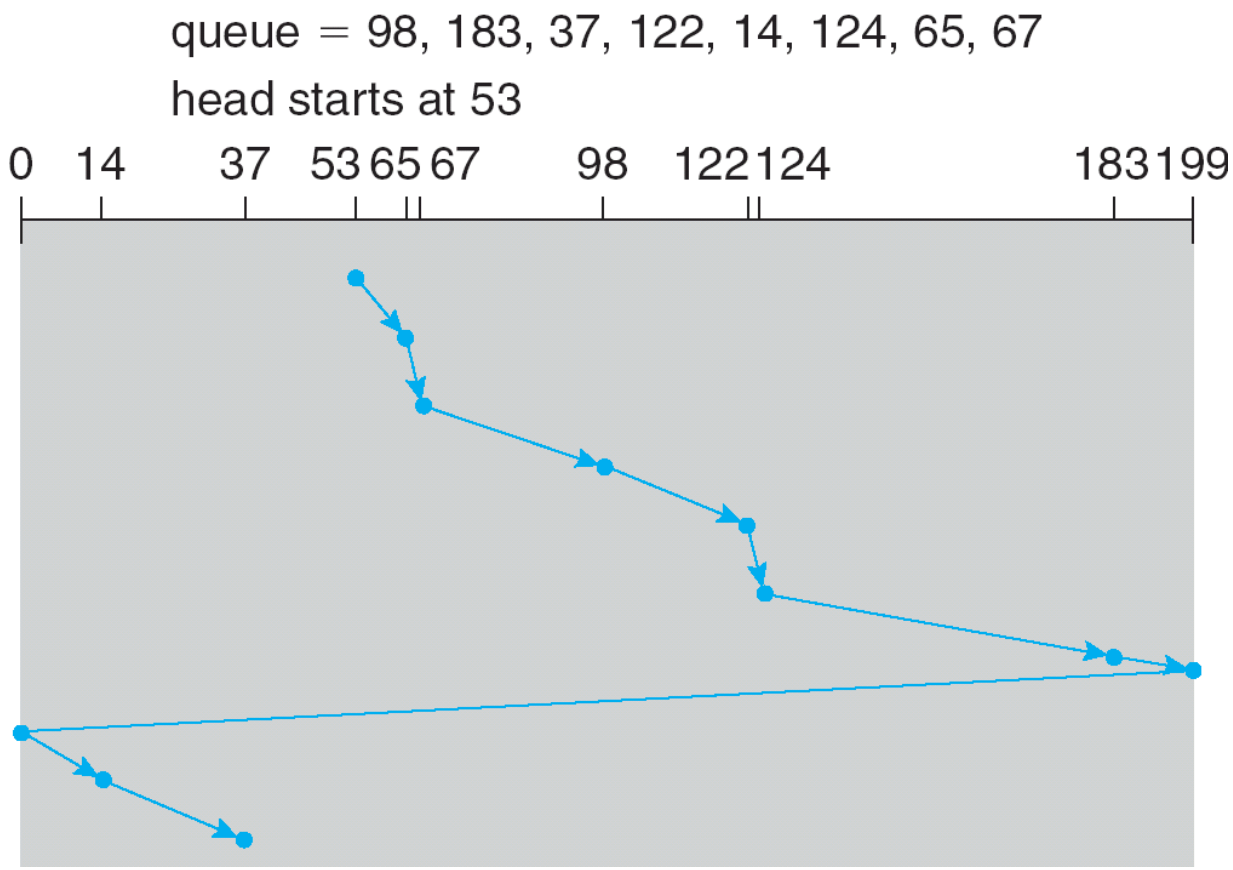
# SCAN

- The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.
- Total head movement of 208 cylinders



# Circular-SCAN (C-SCAN)

- The head moves from one end of the disk to the other, servicing requests as it goes
  - When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip
- Treats the cylinders as a circular list that wraps around from the last cylinder to the first one

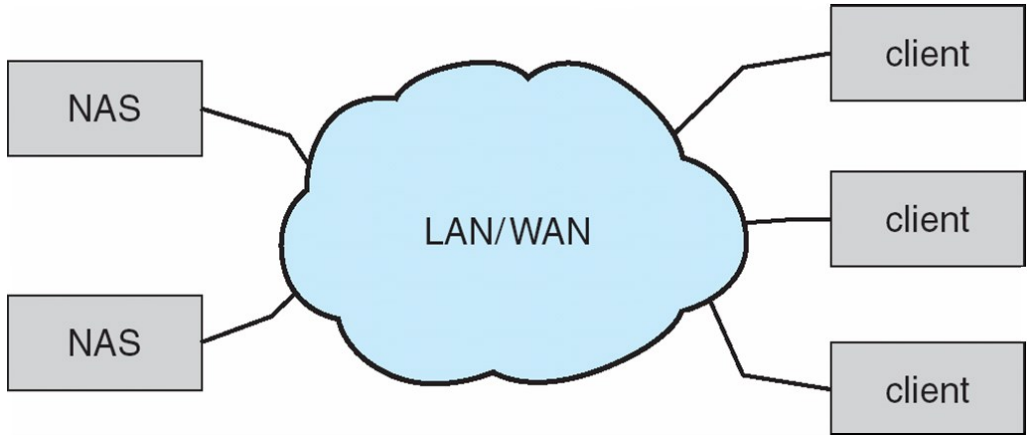


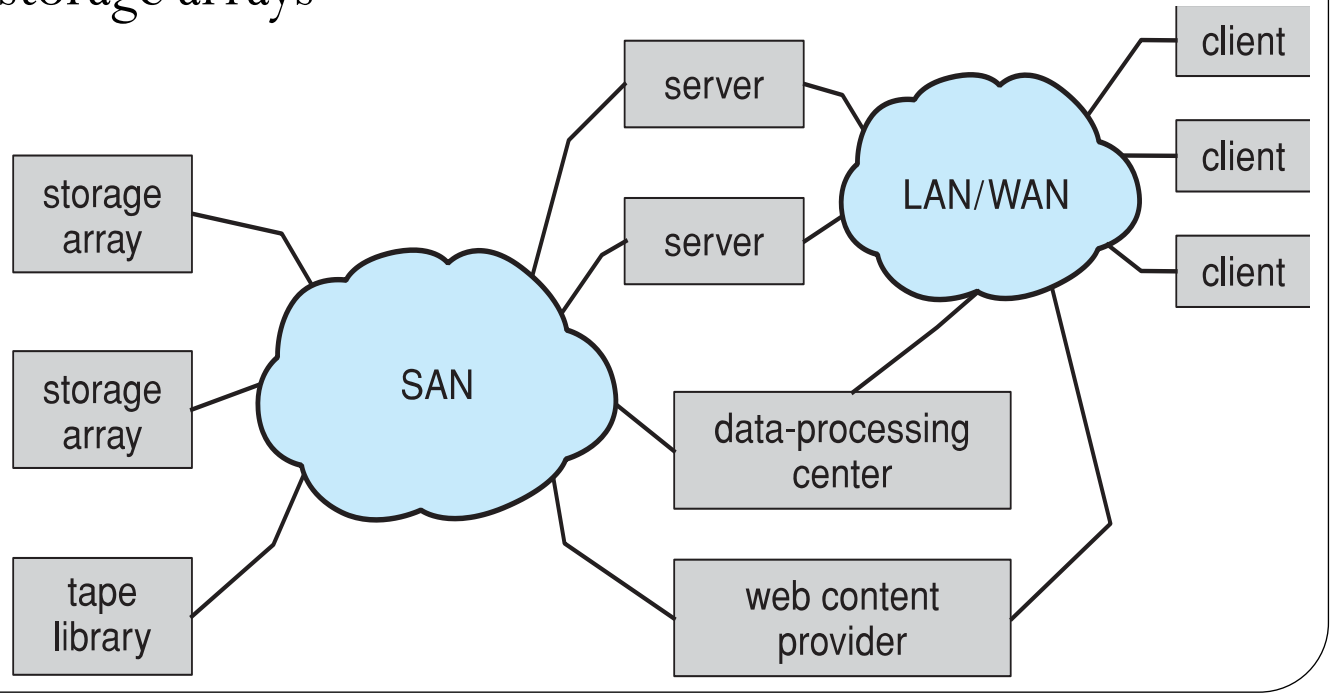
# Error Detection and Correction

- Fundamental aspect of many parts of computing (memory, networking, storage)
- **Error detection** determines if there a problem has occurred (for example a bit flipping)
  - If detected, can halt the operation
  - Detection frequently done via parity bit
- Parity one form of **checksum** – uses modular arithmetic to compute, store, compare values of fixed-length words
- Another error-detection method common in networking is **cyclic redundancy check** (**CRC**) which uses hash function to detect multiple-bit errors
- **Error-correction code** (**ECC**) not only detects, but can correct some errors

# Network-Attached Storage (NAS) & Storage Area Network (SAN)

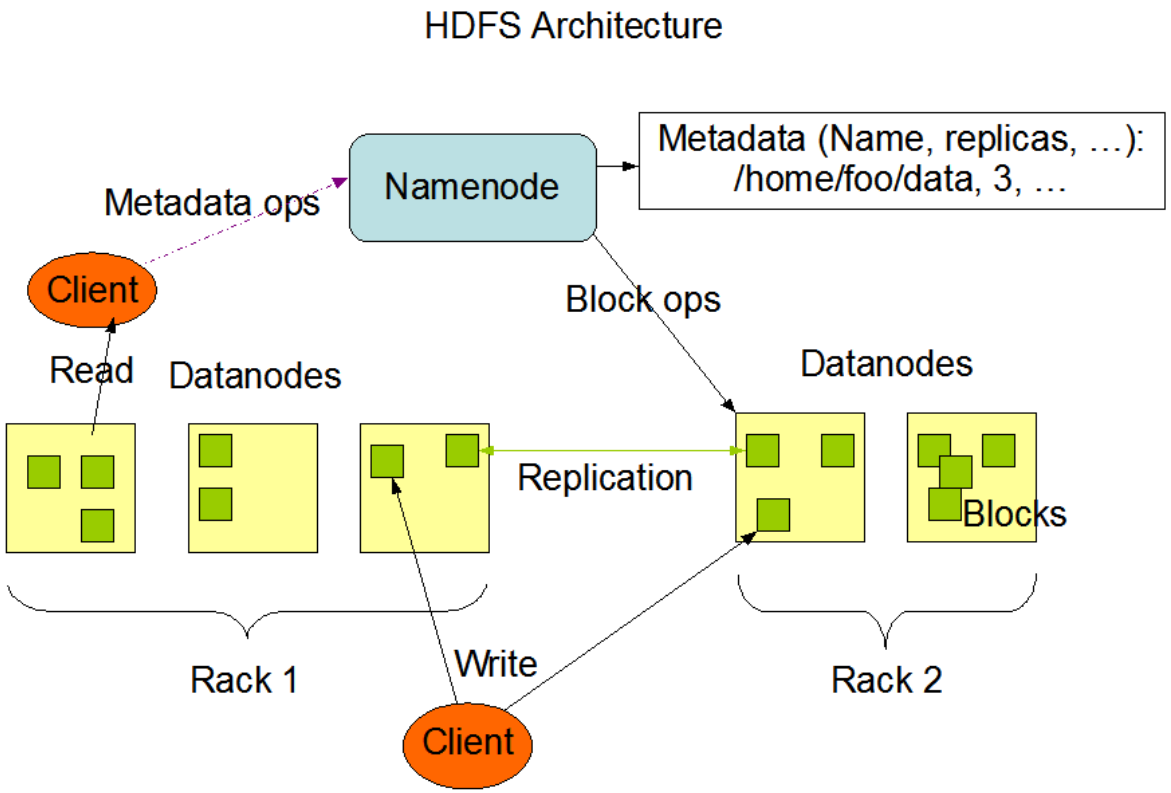
- **NAS** is storage made available over a network rather than over a local connection (such as a bus)
  - Remotely attaching to file systems
- **SAN** Common in large storage environments
  - Multiple hosts attached to multiple storage arrays – flexible

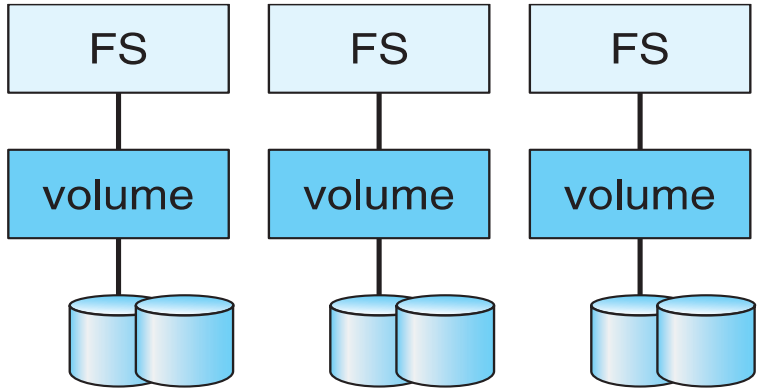


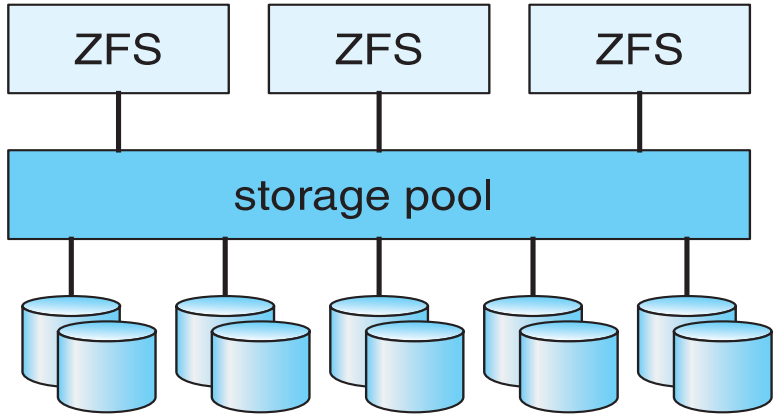
# Traditional and Pooled Storage

- File System (FS)
- Example: Hadoop Distributed File System (HDFS)
- **Horizontally scalable**




(a) Traditional volumes and file systems.




(b) ZFS and pooled storage.

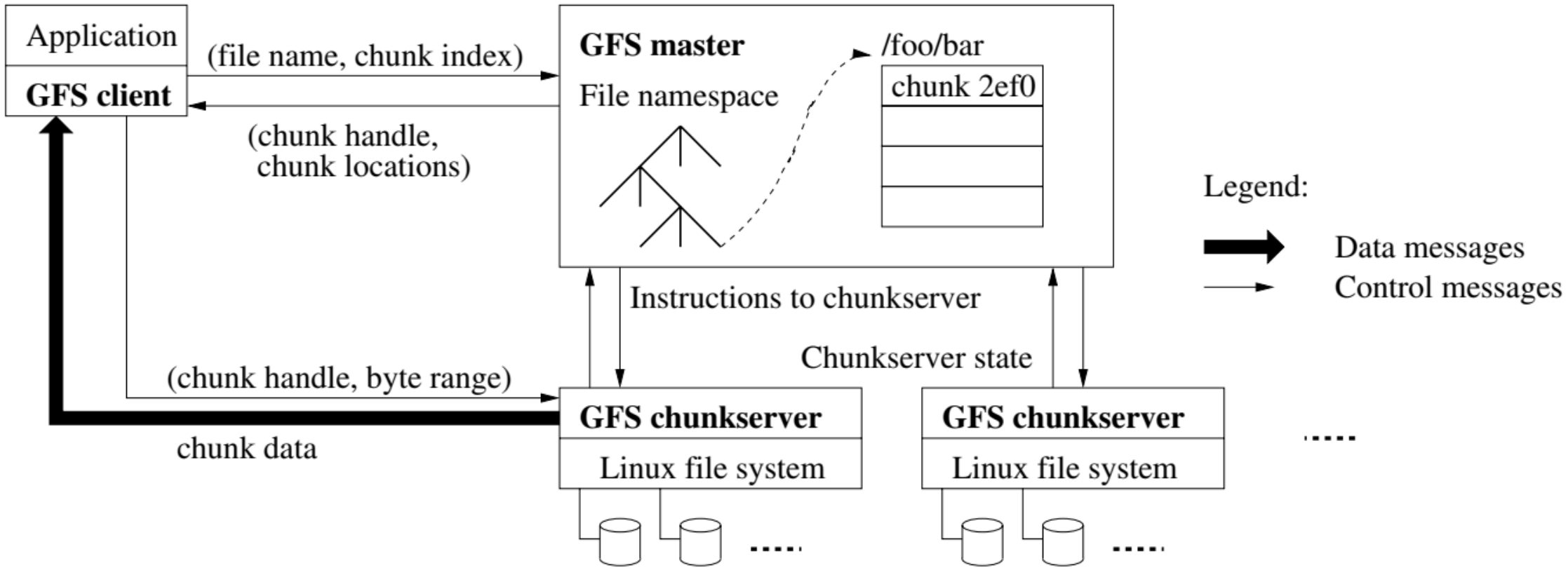
# Big Files to Google File System (GFS)

- Earlier Google effort, "**BigFiles**", developed by Larry Page and Sergey Brin.
  - Supervisers: Hector Garcia-Molina, Rajeev Motwani, Jeff Ullman, and Terry Winograd
- "Big File" was regenerated as "**Google File System**" by Sanjay Ghemawat, et al.
- Google File System (GFS)
  - "It is widely deployed within Google as the storage platform for the generation and processing of data used by Google service as well as research and development efforts that require large data sets." 2003
  - "The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds of clients." 2003

http://infolab.stanford.edu/~backrub/google.html

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google file system." Proceedings of the nineteenth ACM symposium on Operating systems principles. 2003.

# Google File System (GFS)




https://en.wikipedia.org/wiki/Google_File_System

https://sites.google.com/site/gfsassignmentwiki/home

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google file system." Proceedings of the nineteenth ACM symposium on Operating systems principles. 2003.

# GFS to HDFS

- Google File System (GFS) has similar open-source
  - "Hadoop Distributed File System (HDFS)"
- GFS and HDFS are distributed computing environment to process "Big Data".
- GFS and HDFS are not implemented in the kernel of an operating system, but they are instead provided as a user space library.
- GFS and HDFS properties
  - Files are divided into fixed-size chunks of 64 megabytes.
  - Scalable distributed file system for large distributed data intensive applications.
  - Provides fault tolerance.
  - High aggregate performance to a large number of clients.

https://en.wikipedia.org/wiki/Google_File_System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google file system." Proceedings of the nineteenth ACM symposium on Operating systems principles. 2003.

# Big Data

- **Big data** can be described by the following characteristics:
  - Volume: size large than terabytes and petabytes
  - Variety: type and nature, structured, semi-structured or unstructured
  - Velocity: speed of generation and processing to meet the demands
  - Veracity: the data quality and the data value
  - Value: Useful or not useful
- The main components and ecosystem of Big Data
  - Data Analytics: data mining, machine learning and natural language processing etc.
  - Technologies: Business Intelligence, Cloud computing & Databases etc.
  - Visualizations: Charts, Graphs etc.

# References


- Mythili Vutukur. Lectures on Operating Systems, Department of Computer Science and Engineering, IIT Bombay, https://www.cse.iitb.ac.in/~mythili/os/
- Avi Silberschatz, Peter Baer Galvin, and Greg Gagne. Operating System Concepts (Tenth Edition). https://www.os-book.com/OS10/slide-dir/index.html
- Online textbook Operating Systems: Three Easy Pieces (OSTEP)

ขอบคุณ
Thai

Grazie
Italian

תודה רבה
Hebrew

धन्यवादः
Sanskrit

ಧನ್ಯವಾದಗಳು
Kannada

Ευχαριστώ
Greek

Thank You
English

Gracias
Spanish

Спасибо
Russian

Obrigado
Portuguese

شكراً
Arabic

https://sites.google.com/site/animeshchaturvedi07

Merci
French

多謝
Traditional
Chinese

धन्यवाद
Hindi

Danke
German

多谢
Simplified
Chinese

நன்றி
Tamil
Tamil

ありがとうございました
Japanese

감사합니다
Korean