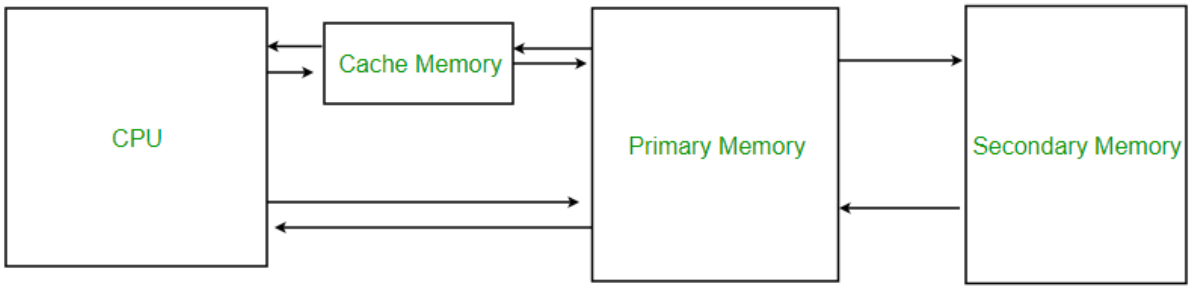
3.2.2 CACHING

Caching refers to **storing frequently accessed data** in a **fast-access memory** (like RAM or CPU cache) so that future requests for that data can be served more quickly. Caching in an operating system is a powerful technique used to **speed up data access** and improve overall system performance.

Concept:

- When a program or OS component needs data, it first checks the **cache**.
  - If the data is found (**cache hit**), it's retrieved instantly.
  - If not (**cache miss**), the data is fetched from slower storage (like disk), and then stored in the cache for future use.
- Caches use **replacement policies** like Least Recently Used (LRU) to manage limited space.



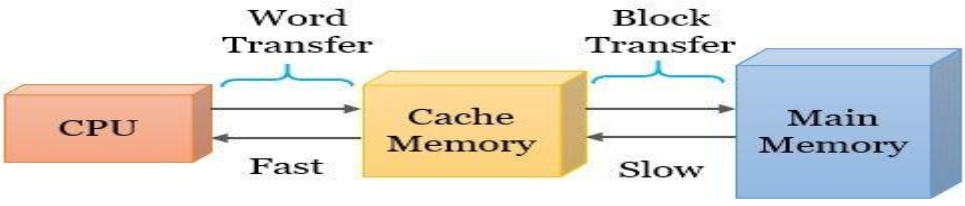
Components in the Diagram:

1. **CPU (Central Processing Unit):**
   - The brain of the computer where processing happens.
   - Needs fast access to data and instructions for execution.
2. **Cache Memory:**
   - Small, very fast memory located close to or within the CPU.
   - Stores frequently used data/instructions to **reduce access time**.
   - Acts as a buffer between CPU and primary memory (RAM).

3. **Primary Memory (Main Memory / RAM):**
    - Larger than cache, but slower.
    - Holds the data and instructions that are currently being used by the CPU.
    - Volatile — data is lost when power is off.
4. **Secondary Memory (Storage devices like HDD/SSD):**
    - Non-volatile, large-capacity storage.
    - Stores data and programs **permanently**.
    - Much slower than primary memory.

Explanation:

1. **From CPU to Cache Memory:**
    - When the CPU needs data, it first checks the cache (fastest access).
    - If data is found (cache hit), it's used directly.
2. **From Cache to Primary Memory:**
    - If data is not found in cache (cache miss), it is fetched from RAM.
    - The data is then stored in cache for faster future access.
3. **From Primary Memory to Secondary Memory:**
    - When RAM is full or on program start/load, data is brought in from secondary memory (disk).
    - Similarly, data not needed immediately may be swapped back to secondary memory.
4. **Direct CPU ↔ Primary Memory:**
    - CPU can also access RAM directly if data is not in cache.



The above figure illustrate how the data is access from cache memory.

Types of Caching in OS

Type	Description	Example Use Cases
CPU Cache	Small, fast memory on the processor that stores frequently accessed instructions and data.	Speeds up execution of programs and system calls
Disk Cache	Stores recently read or written disk blocks in RAM.	File system operations, buffering I/O
Page Cache	Caches pages of memory-mapped files and virtual memory.	Virtual memory management, file access
Buffer Cache	Caches raw disk blocks used by the file system.	Improves performance of block device access
File System Cache	Stores metadata and file contents to reduce disk access.	Directory lookups, file reads/writes
DNS Cache	Stores domain name resolutions locally.	Faster hostname resolution
Web Cache	Caches HTTP responses and static content.	Used in browsers and proxies
Application Cache	Stores data within an app's memory (e.g., user sessions, API responses).	Reducing database load
Database Cache	Caches query results or frequently accessed rows.	Improving database performance
Client-Side Cache	Caching on the user's device (browser, mobile app).	Offline access, reduced server requests
Server-Side Cache	Caching on the server using tools like Redis or Memcached.	Faster API responses, session storage

Type	Description	Example Use Cases
Distributed Cache	Cache spread across multiple servers or nodes.	Scalable systems, microservices
CDN Cache	Content Delivery Networks cache static content on edge servers.	Global content delivery, reduced latency

How OS Uses Caching

- **Memory Hierarchy Optimization**: OS uses multiple levels of cache (CPU L1/L2/L3, RAM, disk) to reduce latency.
- **Virtual Memory**: Page caching helps avoid frequent disk access.
- **I/O Efficiency**: Buffer and file system caches reduce the cost of disk I/O operations.
- **Process Scheduling**: Caching helps maintain context and data locality for efficient CPU

Applications of Caching

- Web Browsing
- Content Delivery Networks (CDNs)
- Database Systems
- Operating Systems
- CPU and Memory Architecture
- Mobile Applications
- Cloud Computing
- Gaming and Graphics
- Machine Learning and AI
- File Systems
- Networking
- E-commerce Platforms
- Search Engines
- Streaming Services
- IoT Devices

Policy Name	Description	Pros	Cons
LRU (Least Recently Used)	Evicts the item that hasn't been accessed for the longest time.	Simple, effective in many cases	Can be costly to implement due to tracking access order
FIFO (First-In-First-Out)	Removes the oldest item in the cache.	Easy to implement	May evict frequently used items
LFU (Least Frequently Used)	Discards the item accessed the least number of times.	Good for repetitive access patterns	Can be skewed by old data that was once frequently used
Random Replacement	Evicts a random item from the cache.	Fast and simple	Unpredictable performance
Optimal (Belady's Algorithm)	Removes the item that won't be used for the longest time in the future.	Best theoretical performance	Not implementable in practice (requires future knowledge)

Advantages

- **Faster Data Access**: Reduces latency and speeds up performance.
- **Lower Resource Usage**: Minimizes disk I/O and CPU load.
- **Improved Scalability**: Helps systems handle more users or processes efficiently.
- **Offline Access**: Cached data can be accessed even without a network connection.

Disadvantages

- **Stale Data**: Cached data may become outdated if not refreshed.
- **Limited Size**: Cache memory is small, requiring smart eviction strategies.
- **Complexity**: Managing cache consistency and replacement policies adds complexity.
- **Volatility**: Data in cache is lost when the system shuts down.