Multiprogramming with Dynamic Partitions

In dynamic partitions, a partition is created dynamically for a requesting process.

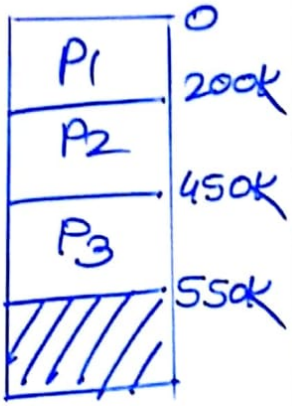
- Compared to fixed partitions, in dynamic partitions, neither the size nor the number of dynamically allocated partitions can be fixed.
- Memory manager continues creating and allocating positions to requesting processes until all physical memory is exhausted.
- When a process is terminated, the vacated memory space is returned to the pool of free memory areas.
- The main problem with fixed size partition is the wastage of memory by programs that are smaller than their partitions. This problem is also known as internal fragmentation.
- With dynamic partitioning, the number and the size of partitions vary dynamically. This improves memory utilization but it also complicates the process of allocation and deallocation of memory.

In variable partition, operating system keeps track of which parts of memory are available and which are allocated.

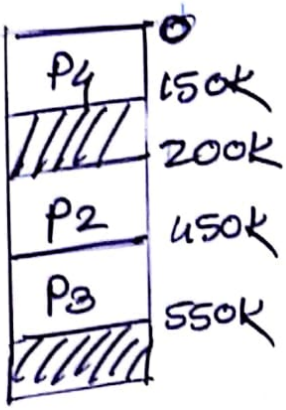
- Assume that we have book of memory available for the processes. Following processes are waiting for execution.

Process No	Size of the process
P_1	200K
P_2	250K
P_3	100K
P_4	150K
P_5	100K

- The Process P_1 to P_3 can be immediately allocated in memory. P_4 can not be accommodated as there is not sufficient memory for P_4.



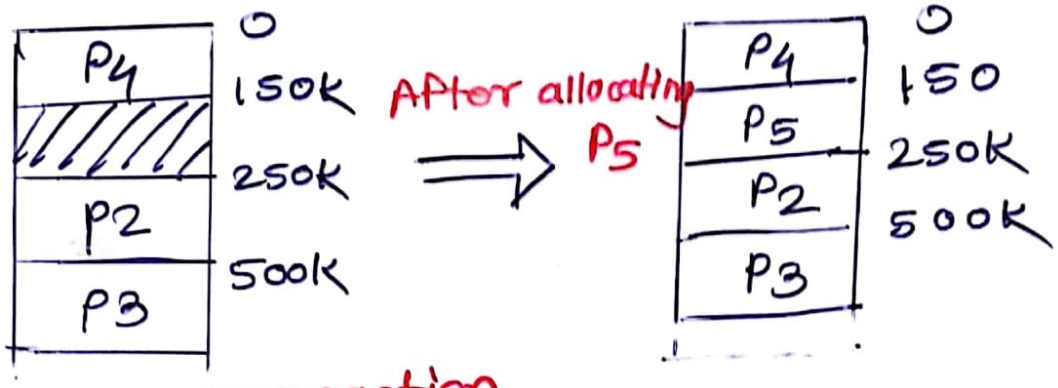
- Let us assume that P_1 terminates It will free 200K memory This space can be given to P_4 as shown



- There are two empty fragments, each of 50K. There is empty space large enough for the requesting process P_5, but the process P_5 can not run as the empty spaces are not contiguous.

storage is fragmented into small number of holes (free space). This problem is also known as external fragmentation.

- one solution to external fragmentation is compaction. It is possible to combine all the holes into a large block by moving existing processes as shown



compaction

Advantages of dynamic partitioning:

(1) Higher utilization of memory.
(2) It supports dynamic memory management. This size of the process can grow during run time
(3) There is no problem of internal fragmentation.

Disadvantages of dynamic partitioning

(1) Dynamic memory management reqeires lots of operating system space, time, complex memory management algorithm.
(2) Compaction is needed due to external fragmentation. Compaction is the time intensive algorithm.

High Speed Memories

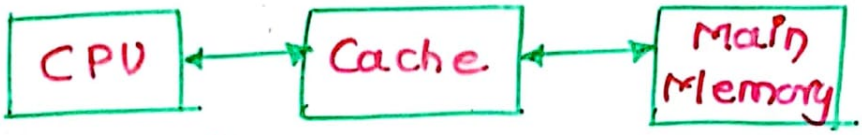
The basic objective of a computer is to execute instruction at a high speed. Similarly, the basic objective of memory system is to provide fast, uninterrupted access by the processor to the memory such that the processor can operate at its own speed.

- A speed gap exists between processor and memory speed.
- A high speed memory is costly
- Memories with smaller cost have very high access time
- Technology puts a restriction on the speed of the memory

Putting high speed memory to increase the speed of the system will make the cost of computer system very high. There are methods through which we can increase the effective speed of the memory without noticeable increase in system cost. These methods are:

- Cache memory: Insert a high-speed memory of smaller size between the main memory and processor.
- Interleaved memory: Access more words in a single memory access cycle. This is instead of accessing one word from the memory in a memory access cycle.

Cache Memory



Use of cache memory

Cache memory is a fast memory of much smaller size between CPU and main memory. Objective is to support CPU access with a minimum delay.

Functioning of cache memory is hidden from both user and system programs. Cache and main memory appears as a single memory to the software being executed.

A system with cache memory offers high throughput despite a smaller size of cache memory.

[Basic principle behind is the locality of reference, which allows a memory hierarchy to deliver high performance;] nearest pages

During program execution, when CPU issues a read instruction, cache memory is checked to see if the contents of the required memory location are in cache. If the contents are found in cache memory, same is read from cache. Otherwise, the same is read from main memory into cache and the process continues.

Analysis of program shows that most of the execution time of a program is taken by loops. A loop is executed several times. When instructions constituting a loop move into cache memory then subsequent execution can continue from cache memory.

Locality of reference :

The

Locality of Reference

This memory hierarchy was developed based on program behaviour known as locality or reference. Memory references are generated by the CPU for either instruction or data access. These accesses tend to be localized in certain region in time, space and ordering. It has been found after analysis of several programs that 90% of execution time is spent on only 10% of the code such as the innermost loop of a nested looping operation

Locality has three properties:

1. Temporal locality
2. Spatial locality
3. Sequential locality.

Temporal locality: Recently reference instruction or data are likely to be referenced again in the near future. This is caused by program loops, local variables of subroutines. Once a loop is entered a small code segment will be referenced repeatedly many times.

Spatial locality: This refers to the tendency for a process to access items whose addresses are near to one another. We are aware of searching and sorting algorithms. most of these algorithms are based on accessing of adjacent elements.

* The temporal aspect of the locality of reference suggests that whenever an instruction or data is first needed, it should be brought into the cache and it should remain there until it is needed again.

* The spatial aspect suggests that instead of bringing just one instruction or data from main memory to the cache, it is wise to bring several instructions and data items that reside at adjacent addresses as well. We use the term block to refer to a set of contiguous addresses of some size.

Sequential locality: In typical programs, the execution of instructions follows a sequential order unless branch instructions create out-of-order execution. The ratio of in-order execution to out-of-order execution is roughly 5 to 1 in ordinary programs. Besides, the access of a large data array also follows a sequential order.

Each type of locality affects the design of memory hierarchy. The temporal locality leads to the popularity of least recently used (LRU) replacement algorithm. The spatial locality assists us in determining the size of unit of data transfer between adjacent memory levels. The temporal locality also helps to determine the size of memory at successive levels. Advance prefetch of instructions are heavily affected by the locality properties.

Cost and Performance of Memory Hierarchy

Using the access frequencies f_i for $i = 1, 2, 3 \cdots n$ we can find the effective access time of the memory hierarchy.

f_i = Access frequency of memory M_i (memory at level i)

h_i = The hit ratio h_i at M_i, is the probability that an information item will be found in M_i

$1 - h_i$ = Miss ratio of M_i

Access frequency of $M_i = (1-h_1)(1-h_2) \cdots (1-h_{i-1})\, h_i$

M_i will be accessed successfully if there are $i-1$ misses at lower levels and hit at M_i,

t_i = Access time of M_i

We have, $t_1 < t_2 < t_3 \cdots t_n$ in memory hierarchy

∴ effective access of time of memory hierarchy

$$T_{eff} = \sum_{i=1}^{n} f_i t_i$$

$$= h_1 t_1 + (1-h_1) h_2 t_2 + (1-h_1)(1-h_2) h_3 t_3 + \cdots$$

The total cost of memory hierarchy is estimated as follows:

$$C_{total} = \sum_{i=1}^{h} C_i S_i$$

Where C_i is the cost per unit of M_i and S_i is the size of M_i

Cost and Performance of two-level memory hierarchy

Let t_1 and t_2 be access time of M_1 (cache) and M_2 (Main memory) respectively. In most two level hierarchy, a request of word not in M_1 will cause a block transfer from M_2 to M_1. Thus a word will be accessed after (t_1+t_2) time interval if the same word is not found in M_1.

Effective access time:

$$t_e = h_1 t_1 + (1-h_1)(t_1+t_2)$$

Where h_1 = Hit ratio of M_1

$(1-h_1)$ = Miss ratio of M_1

Hit ratio of M_2 can be taken as 1.

$$\therefore t_0 = h_1 t_1 + t_1 - h_1 t_1 + t_2 - h_1 t_2$$

$$= (t_1+t_2) - h_1 t_2 = t_1 + (1-h_1) t_2$$

If the cost per unit of M_1 is C_1 and the cost per unit of M_2 is C_2 then the average cost of memory $= \frac{C_1 M_1 + C_2 M_2}{M_1 + M_2}$

Access efficiency of two level memory is defined as:

$$e = \frac{t_1}{t_e} = \frac{t_1}{t_1 + (1-h_1) t_2} = \frac{1}{1 + (1-h_1) t_2/t_1}$$

$$= \frac{1}{1 + (1-h_1) r} \quad \text{where } r = \frac{t_2}{t_1}$$

The goal in memory hierarchy design is to achieve a performance close to that of the fastest device M_1 and cost per bit close to that of the cheapest device M_2. The performance of a memory system depends on various factors some of them are.

- The address generation pattern of the program under execution
- The access time t_i of each level M_i
- The storage capacity of each level.
- The size of the block transferred between adjacent levels.
- Memory replacement algorithms used in case of cache miss.

(1) In two level memory $t_{A1} = 10^{-7}$ s and $t_{A2} = 10^{-2}$ s. What must the hit ratio it be in order for the access efficiency to be at least 90% of its maximum possible value?

Solution: $e = \frac{1}{1+(1-h_1)r}$ where $r = \frac{t_{A2}}{t_{A1}}$

$\therefore r = \frac{10^{-2}}{10^{-7}} = 10^5$

$\therefore 0.9 < \frac{1}{1+(1-h_1)\times 10^5}$

or $1+(1-h_1)\times 10^5 < \frac{1}{0.9}$ or $(1-h_1)\times 10^5 < \frac{1}{0.9} - 1 = \frac{1}{9}$

$1-h_1 < \frac{1}{9\times 10^5}$

or $-h_1 < \frac{1}{9\times 10^5} + 1$

or $h_1 > 1 - \frac{1}{9\times 10^5}$

$h_1 > 1 - 0.1111 \times 10^5$

or $h_1 > 0.9999988$

(2) Suppose a cache is 10 times faster than main memory and suppose that the cache can be used 90% of the time. How much speed up do we gain by using the cache?

Solution

$t_2 = 10 t_1$

and $h_1 = 0.9$

$\therefore$ Average access time $= t_1 + (1-h_1) t_2$

$= \frac{t_2}{10} + (1-0.9) t_2$

$= \frac{t_2}{10} + 0.1 t_2 = 0.2 t_2$

$\therefore$ Speed up $= \frac{t_2}{\text{Average Access time}} = \frac{t_2}{0.2 t_2}$

$= 5$ times

(3) Suppose the access time of cache memory is 80ns and that of main memory is 800ns. It is estimated that 80% of the memory requests are for read and the remaining for write. The hit ratio for read access only is 0.8. Assume a write-through procedure is used. Then.

(1) Determine the average access time of the system consisting only memory read cycle.

(2) Determine the hit ratio taking into consideration the write cycles.

Solution: i) Determining the average access time of the system considering only memory read cycle.

Hit ratio of cache = $h_1 = 0.8$

Hit ratio of main memory, $h_2 = 1$ (It will always be found)

cache memory access time $t_1 = 80$ ns.

Main memory access time, $t_2 = 800$ ns.

∴ Average access time $= h_1 \times t_1 + (1-h_1) \times h_2 \times (t_2+t_1)$

$= h_1 t_1 + h_2 (t_2+t_1) - h_1 h_2 (t_2+t_1)$

$= h_1 t_1 + (t_2+t_1) - h_1 (t_2+t_1)$

$= h_1 t_1 + t_1 + t_2 - h_1 t_2 - h_1 t_1$

$= t_1 + t_2 - h_1 t_2$

$= t_1 + t_2 (1-h_1)$

$= 80\text{ns} + (1-0.8) \times 800\text{ns}$

$= (80 + 0.2 \times 800)\text{ns} = (80+160)\text{ns}$

$= 240\text{ns}$

ii) Determine the hit ratio taking into consideration the write cycle.

Since a write operation in a write through cache is treated as cache miss.

The effective hit ratio $= 0.8 \times \frac{80}{100} = 0.64$

Cache Organization

Cache consists of

(a) cache data memory

(b) tag

- Memory words are stored in a cache data memory. These words are grouped into a block. Basic data transfer unit between main memory and cache is a block of words.
- Each cache block is marked with its block address (address of memory block residing in cache) referred to as a tag.
- Tag indicates to what part of the memory space the block belongs.
- The collection of tag addresses is stored in special memory known as the cache tag memory or directory.
- Whenever a memory block with block address Mi is assigned to a cache block Bi then Mi is in the cache tag memory.

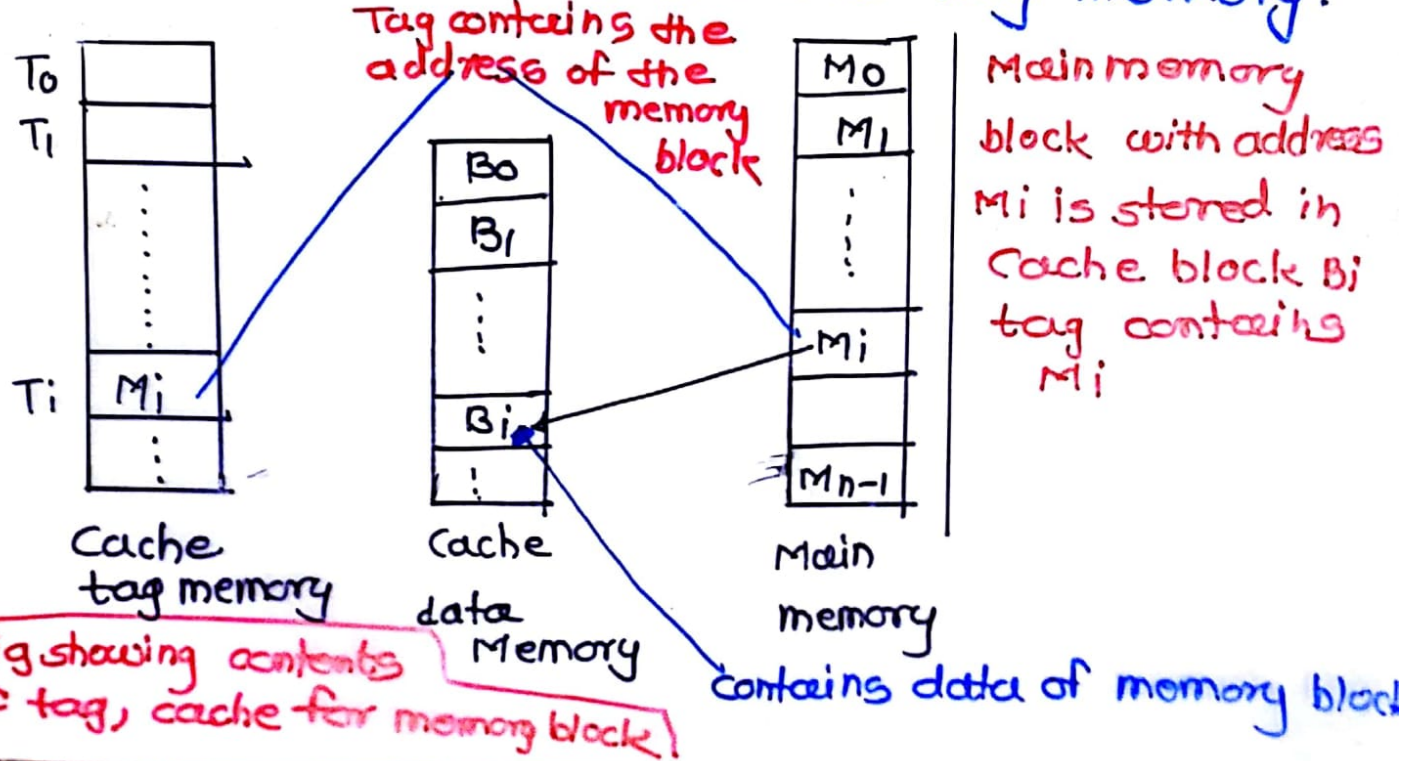
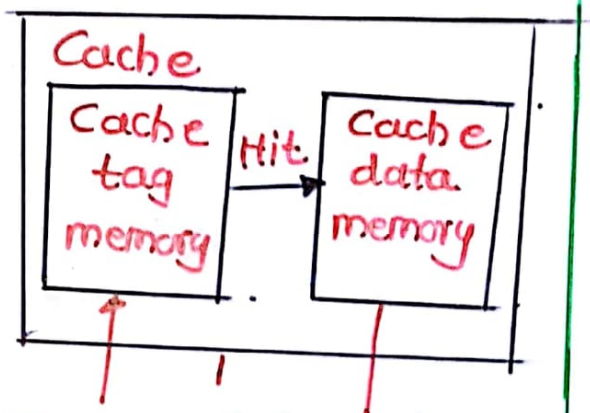
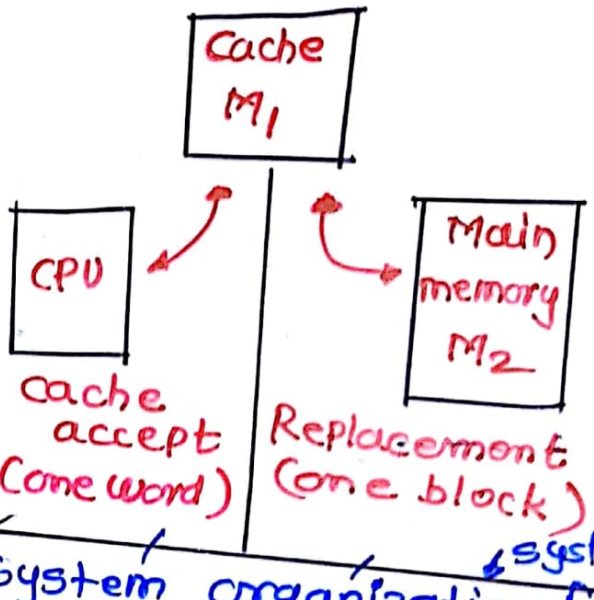


Fig showing contents of tag, cache for memory block



Basic Structure of Cache



System organization for Cache

- For a computer to function with high speed, the time required to check tag address and access the cache's data memory must be less than the time required to access main memory.
- The general way of introducing cache into a computer shown in above figure. The cache and the main memory are directly connected to the system bus. The CPU initiates a memory access by placing an address A on the memory address bus. The cache M_1 immediately compares A to the tag addresses currently residing in the tag memory.
- If a match is found in M_1, the access is completed by a read or write operation executed in cache, main memory M_2 is not involved.
- If no match for address A is found in the cache, that is a cache miss occurs, then the desired access is completed by a read or write operation directly to M_2. In response to cache miss a block of data addressed by A is transferred from M_2 to M_1. The corresponding tag in cache M_1 is updated.

Cache Operation

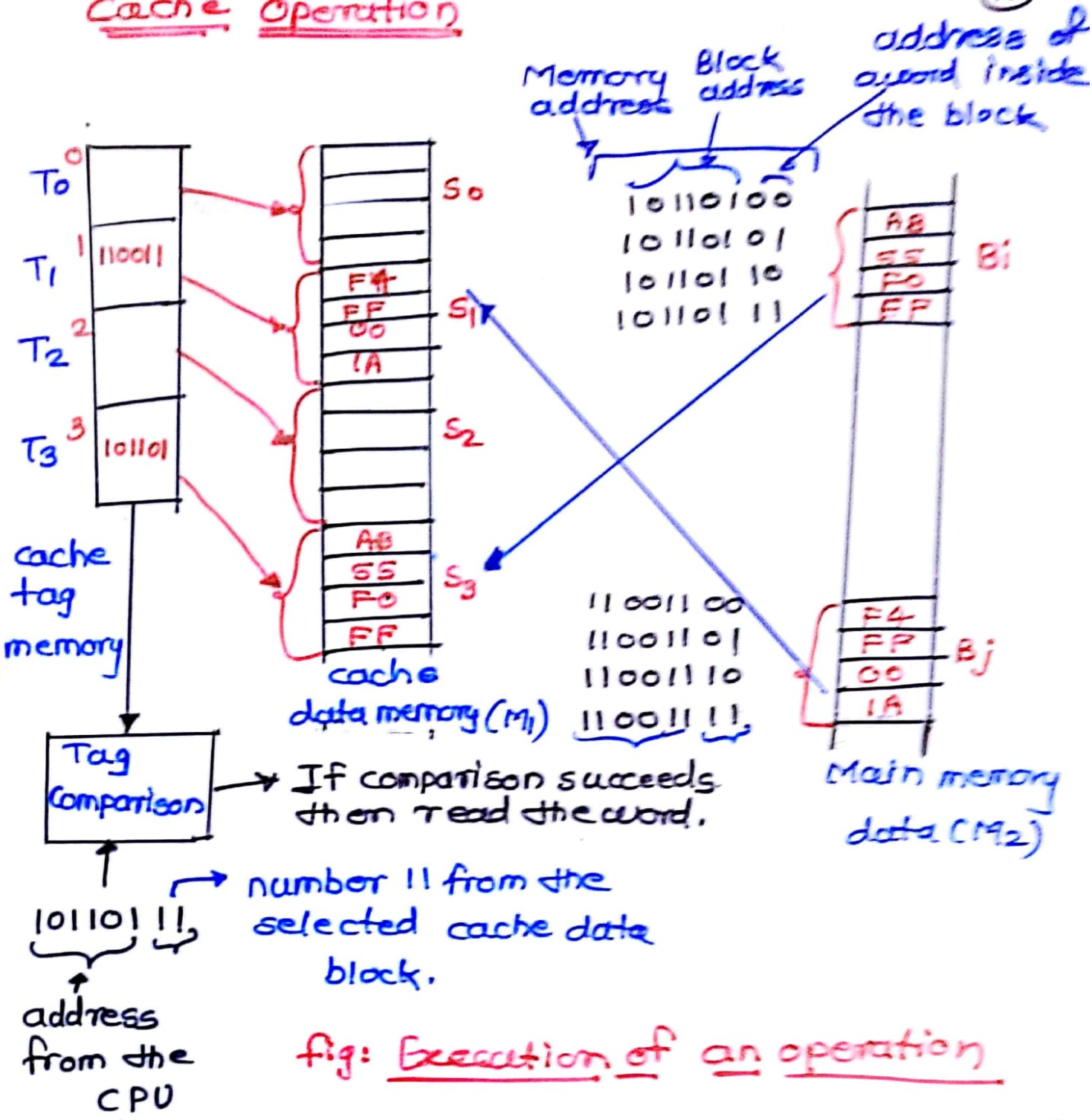


fig: Execution of an operation

Fig shows relationship between the data stored in the cache M_1 and the data stored in main memory M_2.

- Memory address is 8 bit long.
- A block of memory contains four words or bytes.

- Cache memory is of size 16 words. These 16 bytes will form four blocks. These four blocks are designated as set 0 (S_0), set 1 (S_1), set 2 (S_2) and set 3 (S_3)
- Since the cache memory has four blocks (sets), cache tag memory has 4 words.
- Memory addresses (of main memory):

```
101101 00 – First word of block
101101 01 – Second word of block
101101 10 – Third word of block
101101 11 – Fourth word of block
```

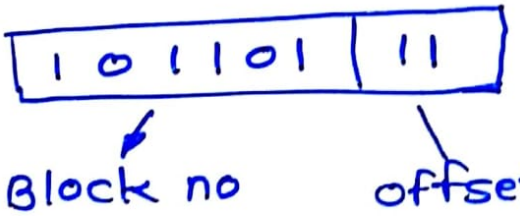
Address of block (Block no) — Address of a word inside a block

Leading six bits of the memory address form the tag or block address. Low order 2 bits define the displacement inside a block. For all four words (with address 10110100 to 10110111) leading six bits are identical (ie 101101). Difference lies in trailing 2 bits, which gives the word no (00, 01, 10, 11) inside a block. A block is identified by the leading six bits.

- Block number B_i (memory address 10110100 to 10110111) is stored in the set number S_3 of cache data memory (M_1)
- Data contents (AB, 55, F0, FF) of block B_j of M_2 are stored in set S_3 of M_1
- T_3 of cache tag memory contains block number (101101) of B_i
- Block B_j (memory address 11001100 to 11001111) is stored in set number S_1 of cache data memory (M_1)

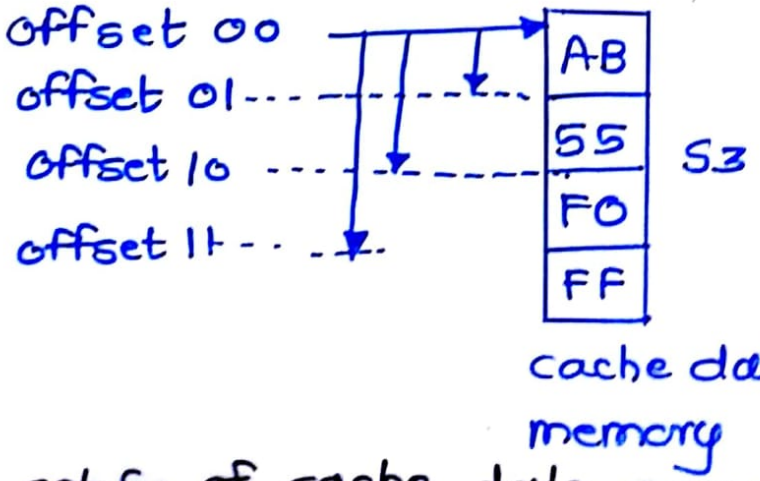
- Data contents (F4, FF, 00, 1A) of block B_j of M_2 are stored in set S_1 of M_1
- T_1 of cache tag memory contains the block number (1100011) of B_j

Read operation : To read a word with address 10110111 (it could be any address), it is sent to cache memory M_1.



Cache memory compares stored tag (T_0=11011 T_3=101101) with the leading six bits of the address (101101 of address 10110111)

If it finds a match (hit), the corresponding memory block is stored in cache data memory. The 2-bit offset (11 of address 10110111) is used to read the target word to the CPO



The set S_3 of cache data memory contains the Block B_i main memory.