# CACHE MEMORY SIMULATION

| | |
|---|---|
| **Project Name:** | Simulation of cache memory. |
| **Due Date:** | 06-09-20 |

We have implemented the objective given to us. The format of input for cache memory is fixed. For run the program we have to run "make" command in the terminal and for the input file we have to make change in Makefile.

In this assignment, we developed a simulation software for cache memory. The simulated cache has the functionality that was discussed in the cache memory lecture videos, but with a new Replacement Policy that works as follows. Each Cache Set is divided into two groups:

1. One group contains the HIGH PRIORITY lines of the set.

2. The other group contains the LOW PRIORITY lines of the set.

The priority is established, if a line is accessed again after the initial access that fetches it into the cache, it is promoted to the HIGH PRIORITY group. If a line is not accessed for sufficiently long (T cache accesses) after being moved to the HIGH PRIORITY group, it is moved to the LOW PRIORITY group. Within a priority group, the Least Recently Used policy may be used to manage the lines.

## Summary of Implementation.

### What is cache?

A cache is a reserved storage location that collects temporary data to help website, browsers, and apps to load faster. Whether it's a computer or phone, web browser or app, you will find some variety of a cache.

## How does it work?

Cache stores some of recently accessed data, it may be anything like data related to any app or any website or something else. When our system tries to read some data then first of it will try to find it out in cache memory. If it finds out then Hit, as cache memory is faster than main memory so data can be copied to RAM very quickly. If data does not find out in cache memory then it will go for main memory and copy whole block from main memory to cache memory, it happens to be miss.

## LRU

**LRU** is also short for **least recently used**, which is an Oracle algorithm that replace oldest data to make room for newest coming data.

Based on the implementation in this assignment, here whenever we try to copy the data from main memory to cache memory and cache memory is full then we remove the block from cache which access least recently.

## Project wiki.

Here, we have a node struct for denoting the block/line in cache memory.

```
Struct Node
{
Int key;
Int val;
Int dirty;
Int valid=0;
Int periority=0;
Int access=1;
Bool first= true;
Std::vector<int> vect;
```

Struct Node* next;

Struct Node* prev;

}

From the structure of Node, we can say that every node is storing its value and key(tag). As every node is containing vector<int>, which has the length equal to size of block.

First of all, we convert the given memory address into its binary format (32 bits). From this we retrieve the set index and block offset according to other given inputs.

During our time copying the block from main memory to cache memory, we put it in low priority and make first access is equal to false so that if this block gets access next time, we can change it priority.

After having set index, we find out the set from the vector(sets). Now this set is LinkedList and every node is mapped to its tag value using HashMap. So now we have HashMap and tag from the memory address we can find out block(node) corresponding to its address from HashMap. If return value of HashMap is valid then we know that the block exists in cache memory and using offset we can find out value. If the block does not exist then we create a new block(node) and inset it in LinkedList and mapped to its tag using HashMap only if LinkedList length is not less than set length.

If our set is full than we remove the block based on Least recently used (LRU) algorithm. We check that if there is any block is accessed least recently and having priority 0, if it exists then we remove that block. Otherwise same process we apply to block having priority 1.

After executing every instruction, that is read or write we go for swaping, it means that we check that if there is any block which has priority 1 and does get accessed too long (like T) then we change it priority from 1 to 0.

## Functions:

1. Int main(int argc, char *argv[]):

   In this function we only checking the condition of given input that the no of inputs given through terminal should be 2.

2. Int BinaryToDec(int n):

Here we are converting given int to its decimal format.

3. String decToBinary(int n):

In this function, we are converting given int to its binary format and returning it in string format.

4. Void indexing(int associativity, int block_size):

Based on given associativity and block_size, this function confirms how many bits we require for sets and offset.

5. put(int key, int value, int set, int offset, vector<map<int, struct Node*>> &set):

First of all, we check that if given key(tag) exist or not. If it exists then it is hit and overwrite the data making it valid and also focusing that does this block has first is equal to false, if yes then change the priority from 0 to 1.

If the given tag does not exit then first of all we check that does this set has any vacancy if not then we remove the block from set using LRU and then copy the block from main memory to cache memory and also put the value in the block corresponding to its offset and remaining value in the block we fill by –1.

6. Int get(int key, int set, int offset, vector<map<int , struct Node*>> &resposeArray):

Here first of all we check that does in given set, it contains that tag or not. If yes then it is hit and check that does its first is equal to false if yes then make it true and return the value.

If the given tag does not exists the miss++ and we call put function again.

7. Void remove(vector<map<int, struct Node*>> &responseArray, int set):

During removing the value we use LRU algorithm. First of all, we apply LRU on the low priority and then high priority. If we find out any block in any low priority then we remove it otherwise we go for removing high priority block.

## Input format:

The format for input file should be like:

Cache size in bytes

Cache line/block size in bytes

Cache associativity

T

Memory address W/R data

## Time Complexity:

So, in general we think, if we use array as memory then for finding particular block having desire tag then we have to iterate whole array so it's time complexity would be O(n) and similarly for finding the block removing then again, we have to iterate whole array so this would also take O(n) time.

But I have tried that I used LinkedList and HashMap simultaneously, HashMap contains mapping tag to block it means during fetching the of particular tag I can do this in constant time so it goes for O(n). In case of finding the block for removing I also iterating whole LinkedList so it goes for O(n).

**THANK YOU,**

**KAPIL VERMA**

**2018CS10348**