Implementation notes:

This is a very basic implementation of file cache using multi threading.File cache is maintained as a doubly linked list structure as follows.

struct file\_cache {

char filename[MAXSIZE]; // Name of the file

int pin; // {0,1} Pin value of 1 indicates file is in cache, 0 not in cache.

int dirty; // {0,1} Dirty bit. 1 indicates that buf contents are not written to local file

system. 0 indicates cache and local filesystem are consistent.

char \*buf; // buffer to hold the data already read or data to be written to file system.

file\_cache\* next; // pointer to the next element in the cache list. For tail node(last node in

cache list) next is NULL.

file\_cache\* prev; // pointer to the prev element in the cache list. For head node (first node

in cache list)) presv is NULL.

};

It is assumed that there is only a single cache which client can access and it can hold a maximum of 100 entries. I am not using file\_cache\_construct() specified in the given .h file. Hence i have removed it and a global variable max\_entries hold the no of elements in the cache. I assume its OK.

Compile the program with DDEBUG = 1 to print debug messages

(gcc -w -DDEBUG=1 –o file\_cache file\_cache.h file\_cache.c main.c -lpthread)

Program flow is as follows.

To simulate and use the advantage of multi threading I have used a structure called request\_list. request\_list has the following structure.

struct request\_list {

int req\_number; // Indicating the request number . incremented by 1 for each request.

char operation; // Variable indicating what operation to be done on file

char \*filename; // File on which request to be performed.

request\_list\* next; // Pointer to the next request\_list object.

};

Client can call add\_request() to place a new request to be processed. To simplify the implementation, files on which client can operate is already defined by "char \*filename[6]={"a","b","c","d","e","f"};". Client calls add\_request() with a filename and an operation(randomly selected as specified in main()). operation can be of 5 types .

'p' to pin files to cache,

'r' to read data from cache,

'w' for write data to cache,

'u' to unpin file from cache

'd' to flush all dirty cache.

main() create NUM\_OF\_THREADS and all threads wait on process\_list() function. A global variable "n\_requests" tracks the total no of pending request to be processed. Initially all threads wait on "cond\_var" as "n\_requests" is 0. As client adds request by calling add\_request() "n\_requests" gets incremented and threads get signalled to process request. "process\_list()" calls "get\_req()", which decreases "n\_requests" and removes a request from request\_list and send it to process\_list() to process it. "process\_list()" process the request by calling "do\_request()". "do\_request()" performs the operation by checking the operation value associated with the request structure.

Function: void add\_request(int , char \*, char , pthread\_mutex\_t\*, pthread\_cond\_t\*);

====================================================================================

To add a new request to the request\_list structure. Request is added to request\_list exclusively using mutex variable. "n\_requests" count is incremented and conditional variable

is signaled to indicate the presence of new request.

Function: request\_list\* get\_req(pthread\_mutex\_t\* );

=====================================================

Remove a request from the front of request\_list and send it to process\_list () to perform specified operation. "n\_requests" count is decremented. request\_list access is exclusive.

Function: void do\_request(request\_list\*);

=========================================

Perform the desired operation by checking the value of operation field associated with request\_list object.

'p' to pin files to cache , calls file\_cache\_pin\_files()

'r' to read data from cache, calls file\_cache\_file\_data()

'w' for write data to cache, calls file\_cache\_mutable\_file\_data()

'u' to unpin file from cache, calls file\_cache\_unpin\_files()

'd' to flush all dirty cache, calls file\_cache\_destroy()

Function: void \*process\_list(void\*)

=====================================

Threads wait on process\_list(). As "n\_requests" becomes non zero, thread gets signalled and process the request .It unlocks the ,utex before processing the request so that any thread can process it.

Funciton : void file\_cache\_destroy(file\_cache\*)

===============================================

This function is used to flush all dirty cache entries to local file system. It takes a pointer to the beginning of the list and iterate through each file\_cache object. If any object is encountered with dirty bit 1 , its contents are written to disk , dirty bit is changed to 0 and buf associated with that file\_cache object is emptied.

Function : file\_cache\_pin\_files() {to add a list of files to cache}

===================================================================

When the client calls file\_cache\_pin\_files() to add a list of files to cache, the cache is searched using search\_cache() with filename to check if the file is already cached or not.If the file is already, cached pin is incremented, it returns . If it is not cached already, a function called add\_to\_cache() is called . add\_to\_cache() checks if the cache is already full. If so it returns.If it is not full, it checks if the file exist in local file system or not, using exclusive open ().If file exist in local file system , it is opened in read only mode and is pinned to cache. Also it reads MAXSIZE bytes to cache buffer pointed file\_cache->buf. If the file doesn't exist in local file system, it is created with mode 0777 and truncated to 10240 bytes and is added to cache list.

Function: file\_cache\_mutable\_file\_data() {to write data to cache}

==================================================================

Used to write data to cache. First function checks if file exist in cache. If it doesnt exist in cache add\_to\_cache() is called to add the file to cache.Else if file exist in cache(ie if pin associated with it is 1) , check the value of dirty bit.If dirty bit is 0 , cache and local file system are consistent and data to be written is copied to buf (file\_cache->buf) associated with file\_cache.If dirty bit is 1, cache and local file system are not consistent(file\_cache->buf contains data that is not yet written to disk) and so the data already existing in buf is written to local file system using write\_to\_file and then new data is copied to buffer and dirty bit is made 1.

Function: file\_cache\_file\_data () {to read data from cache }

=============================================================

Used to read data from cache.Cache is searched using search\_cache() to check if file already exist. If yes a pointer to the data existing in buffer is returned.If file doesn't exist in cache, it is added to cache using add\_to\_cache() and MAXSIZE data is copied to buf and a pointer to the same is returned.

Function: file\_cache\_unpin\_files() { to remove a list of files from cache}

===========================================================================

A list of files is given to file\_cache\_unpin\_files() to be removed from cache. Each file in the list is searched in the cache using search\_cache() to check for existence.If file exist in cache, dirty flag is checked to see if it has some unwritten data. If dirty is 1, contents of buf are written to local file system. Pin is decremented and if pin becomes 0, file\_cache object is deleted from the list, pointers are adjusted to reflect the change and max\_entries count is decremented.