Αντωνης Καλαμακης SDI1800056

for Main:
For argument using input we have 4 cases,
user can give no input and the program will work with the default.
user can only give the algorithm to be used only.
user can give all the possible input except max_traces.
And user can give every single possible input.

To use the different algorithms i have 2 functions one for LRU and one for second chance. They are exactly the same, only difference is that in the first line the main_memory object is a lru_memory or a secondchance_memory.
Its good to mention that in this implementation first we read traces from the bzip file and then from the gcc.
Simply, inside our while we read the lines from the .trace files untill no more traces (for each file) exists or until we have reached the max_traces number readed from both files combined(for example, if we have q= 2 and max_traces=4 we read 2 traces from bzip, 2 from gcc and the loop ends).
To find the page number of our page we simply calculate the (32-log(4096))/4 to find the first 5 digits of our hex trace. Then because of the fact that in this simulator the offset is unnecessary we just resize the mem string and convert it from hex str to int to pass it to the insert function.
And finally when done, close the files and print the statistics.

for class process:
this class is used as a hash table.
here we have a vector with lists, each list can contain struct list_struct nodes.
this is initialized inside the contractor with the size that we gave.
used a custom struct for the nodes because we need to store the page number and the index of the main memory's array that the page is.
with have functions to control the hash table,
insert receives the page number and the index and creates a new node in the list that the specific page should belong(from the hash function).
delete deletes the page's list node inside tha hash table and returns the index of the main memory that hosts this page.
search searches the page and if it exists returns 1.
and finally we have the hash function, simple hash function does the job decently.

Generally for the implementation of the main memory i used a vector of ints as an array that i first of all fill the array and when it is full i simply replace the data of the index in the array with the desired.
Inside my two main memory classes i have stored the procceses so its easier for me to search the hash tables without needing to pass them in the functions as arguments.
Finally simple counters used for disk reading/writing and counting the page faults that happened.

For class lru_memory:
First of all for the implementation of the queue i have a struct named lru_queue_item, here we have 3 members. First the page number that we have inside the main memory, the htn int is to see from which process the current page is from and finally the dirty_bit is used to see if the page, in case of

victimize, we need to we-write it into the disk.

We have 2 functions for inserting, one for reading pages from the first process and one for the second process. Each function gets 2 arguments, the page number and the type of action we do to the memory(R/W from the traces files).

The only difference between the 2 functions is the hash tables that we search/insert to.

In more detail for the insert functions, first of all we run the search function for the hash table to see if the page that the process requests is already inside our main memory, if the function returns 1 then we simply go through the queue to find the page and put it in the front of the queue again so there is less of a chance to victimize it next. Of course we need to check if we using using the page differently(in case of first we read but now we write etc.).
In case of page not already in main memory, we need to top our page fault and read counters because we don't have the page and we need to read it from the disk, then in this implementation i just push the item in front of the queue and if the size of the queue exceeds the maximum size of the main memory we simply remove the last item and choose it as a victim to be replaced from our new page.
To see in which pt the victim is into i used the htn member of my struct.
Finally we need to check if we need to "re-write" the page into the disk, update the main memory array in the location where the victim was and insert the page's number with the index of the main memory's array into the pt.
Now in case that we have empty slots in our main memory, we just push back the new page into the array and insert it to the pt. There is no way of having an empty slot that is not in order(for example cant delete completely a page from the main memory without replacing it) so array.size()-1 works.

For class secondchance_memory:
Here, similarly with LRU we have most of the implementation the same with differences in the queue's struct members and the replacement logic.
int the queue_item struct that we use in the second chance class I've added a reference number member to implement the second chance logic. If this member is 1 we make it 0 and skip it, if the member is 0 we use this page to victimize.

Again as LRU we have 2 insert functions.
For these functions the main differences are that if we already have the page in our memory we simply change the refnum into 1.
In case that the page isn't in our main memory, because of the fact that we don't use the LRU logic we cant just push the new page in and pop the last item.
Here we need to loop through the queue again and again like if it was a circled list until we find an item with refnum equal to 0. If we find a refnum not equal to 0 we need to replace the refnum with 0 and give the page a second chance by skipping it. Finally as before after finding the victim we remove it from the page table, check if we need to re-write to the disk, erase the old page from the queue, pushing the new page number into the end of the queue(because we need a FIFO logic), replace the old page with the new one inside the main memory's array and finally insert the page and the main memory's index in the page table. Unlike when we have full memory, for the case that the main memory has empty slots we do the same thing as done in LRU.