INDIAN INSTITUTE OF TECHNOLOGY, KHARAGPUR
DEPARTMENT OF COMPUTER SCICENCE AND ENGINEERING

Operating Systems Laboratory

Assignment-6

Implementing manual memory management for efficient coding

Advisor: Prof. Mainak Mondal, Prof. Saptarshi Ghosh

IIT KHARAGPUR, APRIL 2023

Group Members

Name	Roll No.
Yashraj Singh	20CS10079
Rishi Raj	20CS30040
Vikas Vijaykumar Bastewad	20CS10073
Aditya Choudhary	20CS10005

Contents

1 The structure of internal page table

Start Index	End Index	Pointer

The above figure shows the structure of a page table entry. Each of the page table entry may have multiple rows and its contains the data about a list. Each row in the entry will contain the start index and end index of the list which have been alloted continuous virtual addresses of our heap memory, and the pointer contains the address of the first block of the segment.

If the list has been alloted fragmented segments (which may happen because of freeElem), then each entry will consist of multiple rows, each row containing information of a continuous segment. An example of such a page table entry is :

Start Index	End Index	Pointer
0	1	0x556fec60bb38
2	3	0x556fec60baa8
4	7	0x556fec60ba48

Our page table will be the collection of page table entries of all the lists.

2 Additional data structures and functions used

Additional Data Structures used :-

1. We have used a hash table to point to a stack of same list names. This is helpful to write recursions using the library. We are using the name of the list to hash the index. We have used chaining to create a stack of same list names.

2. Each of the elements in the stack contains the basic information of the list such as list name, list size, pointer to the page table entry, and a pointer to the next element in the stack.

3. We have used linked list in a number of places. First of all, the memory segments which are free form a linked list, so it makes it very easy for us to get the memory segments using a first-fit strategy and update the head pointer of the linked list.

4. A linked list is used to create stack of lists, and our hash table contains the head pointer of the stack, or top of the stack.

5. Structures for stack node and page table entries are linked using their own linked lists, with a separate list holding the free nodes for each of them. This makes finding free nodes in O(1) time as we only have to update the head pointer of the lists.

Additional functions implemented :-

1. getVal(char *, int) :- This function takes the list name and the index and returns the value at that index. Returns error if list does not exist or index is out of bounds.
2. reallocList(char *, size_t) :- This function takes the list name and changes its size as per the need, by preserving the previous contents of the list. It also updates the page table entry corresponding to the list.
3. printEntirePageTable() :- This function prints the entire page table at the moment it is called. The page table entries of all the list are printed after calling this function.
4. printPageTable(char *) :- This function take the list name and prints the page table entry of the list.
5. printList(char *) :- This function take the list name and prints the entire list.

3 Impact of freeElem() for merge sort

We wrote a C program that forks a child process and loads the mergesort executable, the executable then returns the time of execution to the program, which then computes the average running time. After 100 runs, we got the following data :-

1. After Freeing
   - The average running time is calculate to be 1.951438 seconds. When freeing the memory, the maximum number of blocks that the program used only stayed at 100006 (9.600582 percent of total).
   - For fastmergesort, our time is 1.122 seconds and the maximum number of blocks that the program used only stayed at 100000 (9.600006 percent of total).
2. Without Freeing
   - The average running time is calculate to be 3.460616 seconds. When freeing the memory, the maximum number of blocks that the program used only stayed at 834470 (80.109177 percent of total).
   - For fastmergesort, our time is 1.159 seconds and the maximum blocks used without freeing became 834464 (80.108597 percent of total).

It is evident that freeElem() has very positive impact on running time as well as memory footprint.

4 In what type of code structure will this performance be maximized, where will it be minimized? Why?

Our performance will be maximised in situations where continuous segments are allocated for the lists. In that case, our page table entries for the list will be small and hence accessing the element will involve less traversing of the page table entries of that list.

However, in situations where our memory is fragmented, such as allocating small lists and then deallocating them, out virtual address space will be fragmented and this will result in big page table entries for a list, as it will be allocated fragmented virtual addresses. In this situation, our performance will be slower as accessing an index may involve iterating a big page table entry.

We have implemented our merge sort in two ways, one in which page table entries are bigger, and the other in which it is smaller, and the difference of performance is evident.

5 Did you use locks in your library? Why or why not?

Yes, some locks were used in library to prevent concurrent access in case the library was run by multiple threads in parallel :-

1. A lock on each entry of the hashtable, to prevent concurrent access/modification of the same stack by two threads.

2. A lock on the free list, which contains the head and tail node of the free list as well as its size. A lock is used when accessing it as two concurrent threads modifying this data structure can result in invalid head/ tail pointers or incorrect size.

3. A lock on stack memory, while updating the stack data structure that holds the information of the free memory nodes for stack.

4. A lock for page table entry memory, while updating the page table data structure that stores the information about the free nodes for page table entries.

   There are other places where concurrent access can cause problems, such as multiple threads declaring lists of same name. However, these can be easily handled by the user, either by imposing locks or preventing use of same name by multiple threads.