Indian Institute of Technology, Kharagpur
Department of Computer Science and Engineering

CS39002 : OPERATING SYSTEMS LAB

ASSIGNMENT 6: IMPLEMENTATION OF MANUAL MEMORY MANAGEMENT FOR EFFICIENT CODING

14th April, 2023

Group Number : 24

Shivam Raj
20CS10056

Jatin Gupta
20CS10087

Kushaz Sehgal
20CS30030

Rushil Venkateswar
20CS30045

Contents

1. Structure of Page table

1.1 Page Table Entry

Each page table entry is a struct having the following structure:

```c
typedef struct page_table_entry{
    int frame;
    int used;
}page_table_entry;
```

- **frame:** It is the offset from the beginning of the memory address.
- **used:** This is a flag used to denote whether the frame at this entry is used or not.

1.2 Page Table

Page Table has the following structure:

```c
typedef struct page_table{
    page_table_entry *entries;
    int size;
}page_table;
```

- **entries:** an array of page table entries
- **size:** Number of page table entries

1.3 Explanation

- Each page frame is of 12 bytes.
- Every linked list also has a header node which contains the attributes head, tail and size. The head attribute denotes the index of the first node of the list in the page table. Tail denotes the index of the last node of the list in the page table and size represents the linked list's size. This header is stored in one page frame.
- Each node of the liked list is assigned one page frame.
- To keep track of the available page table indexes, we maintain a queue named `free_list`. This queue contains the indexes of the page table entries that are not currently in use and can be allocated for new pages.

2. Additional Data Structures

- `list`: This struct acts as linked list's header.

  ```c
  typedef struct list{
      int head; // offset
      int tail; // offset
      int size;
  }list;
  ```

`head:`	stores the index of the first node of the linked list in the page table.
`tail:`	stores the index of the last node of the linked list in the page table.
`size:`	the size of the linked list.

- `element:` Represents a node of the linked list.

```
typedef struct element{
    int val;
    int prev;
    int next;
}element;
```

`val:`	value stored in the node.
`prev:`	index of the previous node of the linked list in the page table.
`next:`	index of the next node of the linked list in the page table

- `map < string, int > symbolTable:` A map which stores the mapping of linked list's name with its header's index in the page table.
- `stack < int > global_stack:` This stack is used to keep track of the lists that are in scope and accessible. Whenever a new list is created, the index of the lsit's header in the page table is pushed to the stack and when a list goes out of scope, it is popped from the stack. When a new scope is created, we push -1 to the stack. When a scope ends, we pop the elements of the stack until we get a -1. This process helps to maintain the scope of the list.
- `queue < int > free_list:` This queue is used to maintain the indexes of the page table that are not used currently and can be assigned a new node.

3. Impact of freeElem()

This function is used to free the memory occupied by lists when they are no longer needed and accordingly update the page table and symbol table. After running mergesort 100 times, we got the following average results:

```
Without freeElem -> Memory used: 21227112 bytes Sorting Time: 11115 ms
With freeElem -> 1800492 bytes   Sorting Time: 14344 ms
```

From the above results, it's evident that the use of `freeElem` reduced the memory usage by a significant amount `91%` but slightly increased the execution time. This is expected behavior since calling `freeElem` releases memory that was allocated earlier, making it available for other parts of the program to use. However, calling `freeElem` also incurs overhead, as the lists stored in the global stack which are out of scope are popped out.

4. Performance of Code

Efficient memory management is essential for optimizing program performance. The memory management library described above can maximize performance by minimizing the number of memory allocations and deallocations, and by making efficient use of memory. Programs that use `createList` to allocate memory for linked lists can benefit from the library's ability to allocate a block of memory for the entire list. For instance, a program that use `createMem` to allocate a large block of memory for matrix calculations can avoid the overhead of allocating and deallocating individual elements and perform matrix calculations more efficiently. However, programs that frequently create and destroy lists or use inefficient algorithms for searching and updating lists may experience reduced performance due to increased memory management overhead.

In general, recursive code involving lists may be more affected by the memory management library than non-recursive code due to the potentially greater number of function calls and memory allocations and deallocations. However, the specific implementation and usage of the library functions will ultimately determine the impact on program performance.

5. Use of Locks

As there are no parallel threads running concurrently, we have not used any locks to manage access to shared resources.