ASSIGNMENT-6 : MANUAL MEMORY MANAGEMENT FOR EFFICIENT CODING

GROUP 34
ASHISH REKHANI
VISHAL RAVIPATI
AMAN SHARMA
ROHIT RANJAN

PAGE TABLE

STRUCTURE OF PAGE TABLE ENTRY:

Firstly, a structure **Node** has been defined to represent each memory block. It is specified by the following variables:

1. **data**(*int*): The value to be stored in the block.
2. **prev**(***Node****): A pointer to the previous memory block.
3. **next**(***Node****): A pointer to the next memory block.
4. **free**(*bool*): A boolean variable indicating if a block is available to be used.

A memory segment is represented by a class **List** which is a linked list of memory blocks represented by **Node**. **List** is specified by the following variables:

1. **name**(*string*): The identifying name of the memory segment.
2. **head**(***Node****): A pointer to the head of the linked list of memory blocks.
3. **size**(*int*): Gives the number of blocks in the linked list.
4. **print**(*void function(void)*): Prints the list

Each page table entry is represented by a data structure **pageTableEntry** specified by the following data:

1. **offset**(*int*): The offset of the entry from the start of the page table.
2. **scope**(*int*): Scope of the page table entry. When a function makes a function call, the scope variable is incremented to represent the local scope of the function. The scope needs to be manually controlled by the programmer.
3. **list**(***List****): Pointer to the memory segment allocated to the page table entry.

Finally, the page table itself is represented as a class with the following member variables and functions:

1. **table**(*map:string ->* ***pageTableEntry***): Maps all the names of the page table entries to the corresponding entry.

2. **addEntry**(*void function(string name, int offset, **List*** *list)*): Adds an entry given the name of the process, offset of the entry and pointer to the memory segment. The function throws an error if the entry with the given name exists or if the space required is insufficient.
3. **findEntry**(*bool function(string name)*): Returns true if the page entry with the given name already exists and false otherwise.
4. **getOffset**(*int function(string name)*): Gets the offset of the entry with the given name.
5. **getScope**(*int function(string name)*): Gets the scope of the entry with the given name.

Each level of memory management has been represented with an appropriate data structure that allows us to manage all the different levels seamlessly and also represent the hierarchy of the memory organization.

FUNCTIONS AND DATA STRUCTURES:

ASSIGNMENT-SPECIFIED FUNCTIONS:

1. **createMemory**(int function(int)): Creates the memory block given the size of the block. Returns 0 if successfully created, -1 in case of any error.
2. **createList**(**List*** function(string, int)): Creates the list of memory nodes given the name of the memory segment and the size of the list. The function finds free blocks in a **First Fit** manner, adding all blocks found to be free immediately to the list. Returns the list object on successful completion, or a NULL if the list to be created is too big or the segment with the given name already exists.
3. **assignVal**(int function(string, int, int)): Assigns the given value at the given index of the node in the memory segment of the given name. Returns 0 on successful completion or -1 in case of an error if the index is not found in the list or if the list does not exist.
4. (renamed from **freeElem** to better describe what the function executes) **freeList**(int function(string)): Frees the memory allocated to the segment with the given name. Returns 0 if the memory has been freed and -1 in case the list was not found.

ADDITIONAL FUNCTIONS:

1. **startScope**(void function(void)): A user-callable function used to manage the scope of the function. Increments the scope to represent the new scope the program is running when a function is called.
2. **endScope**(void function(void)): A user-callable function used to manage the scope of the function. Decrements the scope to represent the return to the previous scope from which the function had been called. The function then iterates over all the nodes in the page table that are in a more local scope (represented by a greater value of the scope variable) and frees the node to be available as the function has ended and the scope is destroyed.

3. **freeMemory**(int function(void)): An additional function that frees up the initial big block of memory allocated, is called at the end of the function to free up all the memory allocated. Returns 0 on successful execution and -1 if the function has already been called before.

In addition, the structures and functions used for the page table were also used which have been already described earlier.

EFFECT OF freeElem/freeList:

The results after averaging over 100 runs are as follows:

1. **With freeList:**
    a. Memory footprint: 259.159766 MB
    b. Time taken: 140.70834 seconds
2. **Without freeList:**
    a. Memory footprint: 259.146289 MB
    b. Time taken: 147.219699 seconds

We note that while the memory usage is almost the same (it is dominated by the 250 MB requested at the start), there is a **significant (~ 5%) increase** in **time taken** when freeList is **not** called. Hence, freeList calling has the effect of speeding up the process.
There are two factors involved:

1. The additional time taken by freeList to iterate through and free the pages.
2. The additional time taken by the program when freeList is not called as all the initial pages are taken up the initial function calls due to the First Fit strategy.

The two factors counter each other. The experimental results show that the second factor is the dominating factor which results in a net slowdown when freeList is not called.

We note that in the case of Best Fit, not calling freeList is expected to improve performance as finding free pages involves an entire search regardless of the previous allocation, removing the second factor and improving performance.

LOCKS

The following locks have been used in the implementation:

1. **lockMemory**: This lock has been used to ensure that only one particular caller can access the initially allocated big memory block at any point. This is to ensure that no two processes/callers write on the same memory block and create a race condition.
2. **lockPageTable**: This lock has been used to ensure that only one particular caller can access the page table. This is to ensure that multiple processes/callers do not write on the same page and cause ambiguities.
3. **lockScope**: This lock has been used to lock over the current_scope variable to ensure that appropriate cleaning up of the blocks is done when current scope is to be destroyed

and not create ambiguities when multiple functions at the same scope level exit at the same time.

4. **printMutex**: The lock has been used to ensure that only process/caller prints at a time to ensure that the debugging messages can be printed properly to be viewed.