Operating Systems Laboratory (CS39002)

Spring Semester 2022-2023

Members:
Utsav Basu (20CS30057)
Swarup Padhi (20CS30062)
Rounak Saha (20CS30043)
Anamitra Mukhopadhyay (20CS30064)

Assignment 6: Manual memory management for efficient coding

Report

The memory management strategy used:

Since the memory chunk is to be used only for linked lists of the same kind, the whole memory is converted into a big linked list of **Elements**.

Now, a **freeList** is maintained, which is a linked list and links all the free **Elements**. So initially it contains the whole memory block, and all the **Elements** in contiguous memory blocks linked together.

Whenever we need to create a list, the head of the **freeList** is shifted (not contiguously in the memory segment but by traversing the linked list) and that part of the **freeList** is given to a new list.

Whenever we need to delete a **List**, say L, the tail of the **freeList** is linked to the head of L and then the tail of **freeList** is just assigned the tail of L.

Advantage of this strategy:

This strategy ensures that even if there is no contiguous block of memory available to allocate a list, say L, we can still allocate a list if the sum of all the holes in the memory is enough for L. This is why all the holes are maintained as a linked list (named **freeList**). This ensures there is no fragmentation in the pre-allocated memory segment.

MACROS:

1. **constexpr int32_t TABLESIZE = (1 << 15);**
2. **constexpr int32_t NAMESIZE = (1 << 12);**
3. **using ptr_t = int32_t;**

GLOBAL VARIABLES:

1. **Element * mem;**

   Description: Pointer to the large chunk of memory allocated

2. **Table T;**

   Description: Keeps track of the lists created, also does a bookkeeping of function scope

3. **List freeList;**

   Description: Keeps track of the free chunks of the memory segment (mem)

DATA STRUCTURES USED:

1. **Element: Node of the linked list**

   Fields:

Name	**Type**	**Purpose**
prev	ptr_t	Index(index as in mem[index]) of previous Element of the linked list
next	ptr_t	Index(index as in mem[index]) of next Element of the linked list
data	int32_t	The data in the Element

2. **List: A linked list of ‘Element’s**

Fields:

Name	Type	Purpose
head	ptr_t	Index(index as in mem[index]) of first Element of the linked list
tail	ptr_t	Index(index as in mem[index]) of last Element of the linked list
size	int32_t	The number of Elements in the linked list

3. Tablerow: A record in the ‘Table’ structure (explained later)

Fields:

Name	Type	Purpose
name	char[NAMESIZE]	Name of the list. A special name ‘__fn_call’ has been used for bookkeeping of function scope.
li	List	An instance of List structure to keep track of head, tail and size of the list

4. Table:

Fields:

Name	Type	Purpose
tab	Tablerow[TABLESIZE]	Keeps track of the lists currently in use and some bookkeeping of function scope
size	int32_t	Size of the table

Methods:

1. **int32_t find(const char * name);**
   Description: Returns the index of the first occurrence of the List ‘name’ from the end of the table. -1 if not found.

2. **int32_t findInScope(const char * name);**
   Description: Returns the index of the first occurrence of the List ‘name’ from the end of the table in the current function scope (by searching up to the special name ‘__fn_call’).

DESCRIPTION OF THE FUNCTIONS OF THE LIBRARY:

1. **bool createMem (size_t size);**

   Arguments:
   size: Size in bytes of the chunk of memory to be allocated

   Return: true if successfully allocated the memory, otherwise false

   Description: Allocates memory, creates a linked list of Elements out of the whole memory, and initializes the freeList

2. **bool createList (const char * list_name, int32_t num_elememts);**

   Arguments:
   list_name: Name of the linked list
   num_elements: Number of Elements in the list

   Return: true if successfully created, false otherwise. Returns false if a list of the same name exists in the current function scope.

   Description: Allocates the list by providing a part of freeList, updates (reduces) the freeList, and makes an entry in the table T.

3. **bool assignVal (const char * list_name, int32_t idx, int32_t val);**

   Arguments:
   list_name: Name of linked list
   idx: Index of the Element in the list

val: Value to be assigned

Return: true if successfully assigned, false otherwise (if index out of bounds)

Description: Finds the newest list created with name ‘list_name’ (not necessarily in the function scope), finds the Element by parsing the linked list and assigns the value

4. **int32_t getVal (const char * list_name, int32_t idx);**

Arguments:
list_name: Name of linked list
idx: Index of the element in the list

Return: The data in the element

Description: Returns the idx-th element in the list list_name

5. **int32_t freeElem (const char * list_name);**

Arguments:
list_name: Name of linked list

Return: Number of lists deleted, in this case 0 or 1

Description: Deletes the newest instance of list_name. Returns 0 if no list deleted, otherwise 1

6. **int32_t freeElem ();**

Return: Number of lists deleted

Description: Deletes all the lists in the current function scope. Returns the number of lists deleted

7. **void fn_beg();**

Description: Must be called before calling a function (not the functions of the memory management library). Necessary for the bookkeeping of function scope. Inserts a special entry ‘__fn_call’ in table T.

8. **void fn_end();**

Description: Must be called after calling a function (not the functions of the memory management library). Necessary for the bookkeeping of function scope. Deletes all lists in that function scope, i.e. deletes all lists up to the special entry ‘__fn_call’.

9. **void printList(const char * list_name);**

   Arguments:
   list_name: Name of linked list

   Description: Prints the list ‘list_name’

10. **ptr_t listBegin(const char *list_name);**
    Description: Returns pointer (ptr_t) to first element of the list

11. **ptr_t listPtr(const char *list_name, int32_t idx);**
    Description: Returns pointer to the idx-th element in the list list_name

12. **ptr_t listEnd(const char *list_name);**
    Description: Returns pointer to one past the element of the list

13. **ptr_t listNext(ptr_t ptr);**
    Description: Returns pointer to next element.

14. **ptr_t listPrev(ptr_t ptr);**
    Description: Returns pointer to the previous element

15. **int32_t listGetElem(ptr_t ptr);**
    Description: Returns the value stored in the current element

16. **void listSetElem(ptr_t ptr, int32_t val);**
    Description: Sets val as the value of the current element

17. **deleteMem()**
    Description: Deletes the memory segment

IMPACT OF freeElem()

Using freeElem() the average time is 200 msec. Without using freeElem() the number of lists getting created is too high and the table T (used to keep track of the lists) is getting full.

PERFORMANCE IN DIFFERENT CODE STRUCTURE

In this memory management scheme, we are using linked lists. Linked lists are supposed to have sequential access. So, the assignVal() and getVal() functions, although giving the impression of random access, internally use sequential access. As a result, it increases the access time. The average time is **34.47 seconds** for running the mergesort using these functions (main.cpp). (These functions were implemented as per the instructions in the assignment)

To actually get sequential access to the linked lists, we

implemented additional functions (number 10 to 16) that actually access the lists as a linked list. This gave significant improvement in running time, average time: **200 milliseconds** (about 150 times decrease in time)
Note: These are all run using freeElem()

USAGE OF LOCKS

No locks have been used. This memory management is not supported for threading.