OPERATING SYSTEMS LABORATORY

CS39002

Assignment 6:

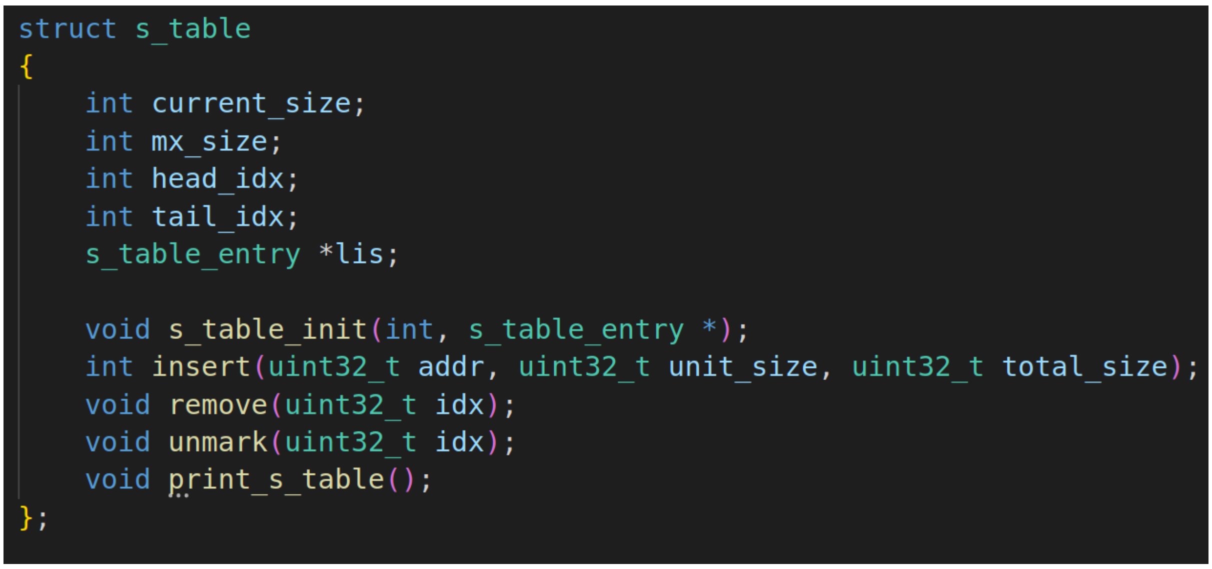
Implement manual memory management for efficient coding

Report

Group No. 21
Nikhil Sarashwat (20CS10039)
Abhijeet Singh (20CS30001)
Amit Kumar (20CS30003)
Gopal (20CS30021)

➢ Structure of Internal Page Table

Structure of **s_table**:



```
struct s_table
{
    int current_size;
    int mx_size;
    int head_idx;
    int tail_idx;
    s_table_entry *lis;

    void s_table_init(int, s_table_entry *);
    int insert(uint32_t addr, uint32_t unit_size, uint32_t total_size);
    void remove(uint32_t idx);
    void unmark(uint32_t idx);
    void print_s_table();
};
```

Structure **s_table** contains four integers and a pointer to a **s_table_entry** object:

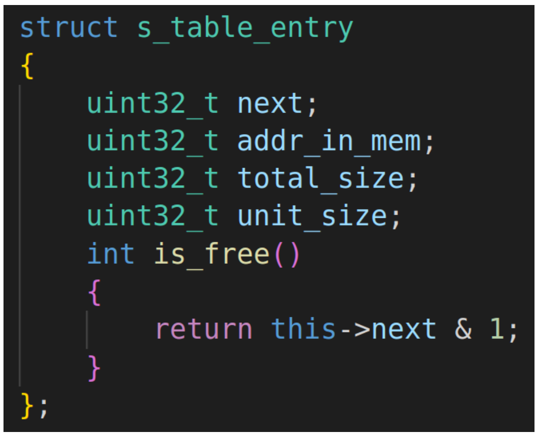
- **current_size**: the current number of entries in the symbol table
- **mx_size**: the maximum number of entries that the symbol table can hold
- **head_idx**: the index of the first in the symbol table
- **tail_idx**: the index of the last entry in the symbol table

The structure also defines several member functions:

- **s_table_init(int, s_table_entry*)**: This function initializes the symbol table with the given maximum size and an array of **s_table_entry** objects.
- **insert(uint32_t, uint32_t, uint32_t)**: This function inserts a new entry into the symbol table with the given address, unit size, and total size.
- **remove(uint32_t)**: This function removes the entry at the given index from the symbol table.
- **unmark(uint32_t)**: This function unmarks the entry at the given index in the symbol table.
- **print_s_table()**: This function prints the contents of the symbol table to the console.

The above approach to symbol table implementation is utilized in order to provide efficient insert and remove operations, which run in O(1) time complexity. By using an array to store the symbol table entries, we can perform random access, enabling us to quickly retrieve the desired element. Additionally, for unused array elements, we chain them together using a linked list, where the **next** field of each entry is used to indicate the index of the next unused element in the array. This allows us to fully utilize the array, without worrying about potential fragmentation that can occur when deleting elements from the middle. Since deleted entries are added to the end of the linked list, this ensures that there are no gaps in the array, which could lead to wasted memory. Overall, this approach provides an efficient and memory-efficient solution for managing symbol tables.

Structure of **s_table_entry**:



```
struct s_table_entry
{
    uint32_t next;
    uint32_t addr_in_mem;
    uint32_t total_size;
    uint32_t unit_size;
    int is_free()
    {
        return this->next & 1;
    }
};
```

Structure **s_table_entry** represents an entry in a symbol table. The struct contains four integer fields:

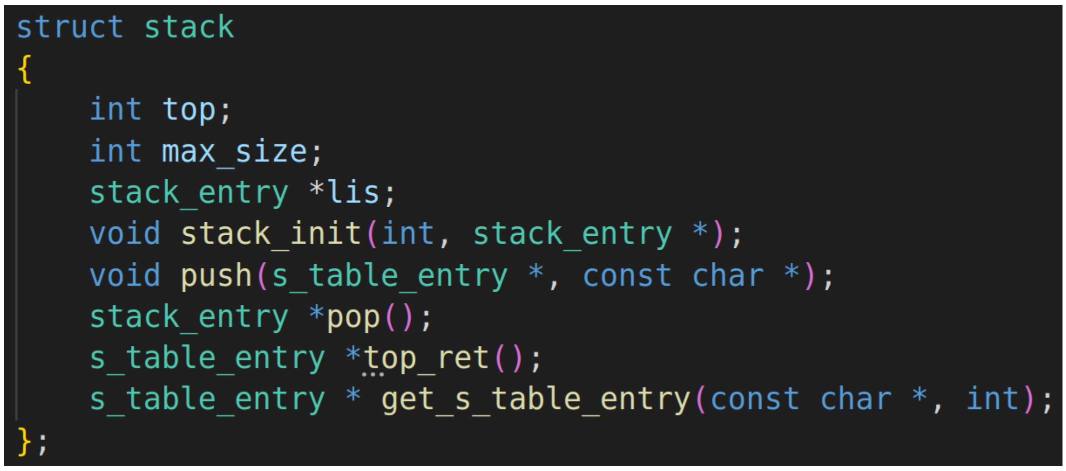
- **next**: an integer that contains the index of the next **s_table_entry** object in the symbol table, with the least significant bit representing whether this block is to be freed or not.
- **addr_in_mem**: represents the index of the entry in memory
- **total_size**: represents the total number of bits used by this entry in memory
- **unit_size**: represents the size of a single unit in bits. For example, the **bool** type takes up 1 bit, an **int** takes up 32 bits, a **char** takes up 8 bits, and a hypothetical **medium_int** type takes up 24 bits.

The **is_free()** member function returns a boolean value indicating whether the current entry is free or not. It does this by checking the least significant bit of the next field, which is used to indicate whether the entry is marked as free or not.

➢ Additional Data Structures/Functions

➢ Data Structures

Structure of **stack:**



```
struct stack
{
    int top;
    int max_size;
    stack_entry *lis;
    void stack_init(int, stack_entry *);
    void push(s_table_entry *, const char *);
    stack_entry *pop();
    s_table_entry *top_ret();
    s_table_entry * get_s_table_entry(const char *, int);
};
```

The **stack** struct contains three fields:

- **top**: an integer which represents the index of the top of the stack
- **max_size**: an integer which represents the maximum size of the stack
- **lis**: a pointer to a stack_entry object which represents the stack implementation

The structure also defines several member functions:

- **stack_init**: a constructor function which initializes the stack with the given size and stack_entry object
- **push**: used to push a **s_table_entry** object onto the stack. It takes two arguments: the **s_table_entry** object to be pushed onto the stack and a string representing the name of the entry.
- **pop**: used to pop the top element from the stack and returns a pointer to the popped element as a **stack_entry** object.
- **top_ret**: returns the **s_table_entry** object at the top of the stack.
- **get_s_table_entry**: takes two arguments: a string representing the name of the entry and an integer representing the index of the symbol table. It returns the **s_table_entry** object with the given name from the symbol table.

Structure of **stack_entry**:

```
struct stack_entry
{
    s_table_entry *redirect;
    char name[20];
    int scope_tbf;
};
```

Structure **stack_entry** represents an element in the stack. The struct contains three fields:

- **redirect**: a pointer to a **s_table_entry** object which is used to redirect to the corresponding entry in the symbol table. This allows for easy access to the attributes of the corresponding **s_table_entry** object.
- **name**: a character array of size 20 which represents the name of the **s_table_entry** object. This allows for easy identification of the **stack_entry** object and its corresponding **s_table_entry** object.
- **scope_tbf**: an integer which represents the scope number of the **s_table_entry** object. The first 31 bits of this integer represent the scope number, and the last bit is used to determine whether the entry needs to be freed. This information is used to determine the lifetime of the **s_table_entry** object and whether it needs to be deallocated or not.

➢ Functions

- **accessList**: Takes in a variable name, an index, and a scope. It first looks up the variable in a global stack to retrieve its unit size and memory address. It then calculates the main index and offset within the memory address to retrieve the correct value based on the index and unit size. The retrieved value is masked to remove any unnecessary bits and then shifted to the correct position. Finally, the function returns the retrieved value. If the variable is not defined, an error message is printed and the program exits.

- **DeallocMainMemory**: This function frees a block of memory in the main memory partition by clearing its allocated flag and boundary tag. It then merges the freed block with any adjacent free blocks, if any, and updates their boundary tags accordingly.
- **AllocMainMemory**: This function creates a new partition in the main memory of the computer by dividing it into blocks with headers, data, and footers. It uses the first fit algorithm to find a free block that can accommodate the requested size and returns the location of the new free block if there is any unused memory left in the allocated block.
- **freeElem_inner**: This function removes a symbol table entry from the global stack and frees its corresponding memory partition, if any. It then removes the entry from the symbol table.
- **startScope**: This function increments the current scope level by 1 to indicate the start of a new scope.
- **endScope**: This function removes all entries from the global stack whose **scope_tbf** value matches the current scope level and unmarks their corresponding entries in the symbol table.
- **partial_compact**: This function performs one iteration of memory compaction by moving data segments to eliminate gaps in the **BIG_MEMORY** array. It updates the corresponding addresses in the **SYMBOL_TABLE** array and coalesces with the next block if it is free.
- **complete_compact**: This function continuously calls the **partial_compact()** function until there is no more space to be compacted. It returns the total number of times the **partial_compact()** function was called.
- **freeElem_helper**: This function removes all entries that need to be freed from the **GLOBAL_STACK**, unmarks their corresponding entries in the **SYMBOL_TABLE**, and compacts the memory using the **complete_compact** function. It then iterates through the **SYMBOL_TABLE** and calls the freeElem_inner function on all entries that are marked for deletion.
- **getScope**: This function simply returns the value of the global variable **CURRENT_SCOPE**, which is used to keep track of the current scope level in the program.

➢ Impact of freeElem() for merge sort

➢ Without freeElem

- Memory footprint is 3337864 bytes
- Code is approximately taking 12 seconds to run without freeElem.

➢ With freeElem

- Memory footprint is 1405218 bytes
- Code is hardly taking 2 seconds to run.

Hence there is a significant improvement in performance when **freeElem** is used both in terms of speedup and memory footprint.

➢ Maximized/Minimized Performance’s code structure

- **Maximum Performance**: Memory usage can be high in recursive functions where memory is not freed until the function returns from the recursion stack. Additionally, allocating lists of significant size can also increase memory utilization. Therefore, it is important to be mindful of memory usage and optimize code accordingly to avoid unnecessary memory usage.
- **Minimum Performance**: To minimize memory utilization, it is advisable for the user to allocate only a few lists of smaller sizes. However, if the program is already reserving a significant amount of memory for the Page Table and Stack, as much as 200 KB, allocating more memory for lists may lead to poorer performance. Therefore, it is important to balance memory usage and program requirements to optimize performance.

➢ Locks

The whole program is run as a single thread hence there is no need of locks. Since only one thread is executing, there is no risk of race conditions, and the program's state is always well-defined. Therefore, using locks in this program would not provide any additional benefits, as there is no contention for shared resources. In fact, using locks unnecessarily could potentially slow down the program by introducing additional overhead for acquiring and releasing locks, even though there is no actual contention.

----------: THE END :----------