OPERATING SYSTEMS LABORATORY SPRING SEMESTER 2021-2022

Hands-on experience for creating better memory management systems

GROUP - 1 LAB REPORT

ABHISHEK GANDHI | 19CS10031
SAJAL CHHAMUNYA | 19CS10051

Internal Page Table (Symbol Table)

Symbol Table basic building block

```
struct symbol{
    int offset;
    bool status = 0;
};
```

This data structure is used to represent a variable or array in the symbol table. It contains two fields.

Data Members

- offset: physical offset for that specific variable or array. memPtr->membase+offset will give us the starting address for that variable.
- status: if the status is zero, then the memory is not freed yet, otherwise the memory is freed, this variable is required to prevent the freeing of already freed memory by the garbage collector.

Symbol Table

```
struct symbolTable{
    symbol *stack;
    int idx;
    symbolTable();
};
```

This data structure is used for storing information regarding variables, arrays, and the scope of a variable and array. We have a micro MAX_SYMBOL_TABLE_SIZE used to specify the maximum size of the stack

Data Members

- stack: This is the symbol table stack(Implemented as an Array). index of a variable in this stack is their logical address
- idx: This variable is used to maintain the top of a stack.

Data structures

Type

```
enum Type {
```

```
    INT,
    CHAR,
    MEDIUM_INT,
    BOOLEAN
};
```

Type is an enum used to specify the type of the variable.

There are 4 Types allowed in our system.

Members

- int: size is 4 bytes
- medium int: size is 3 bytes
- bool: size is 1 bytes
- char: size is 1 bytes

Memory Structure

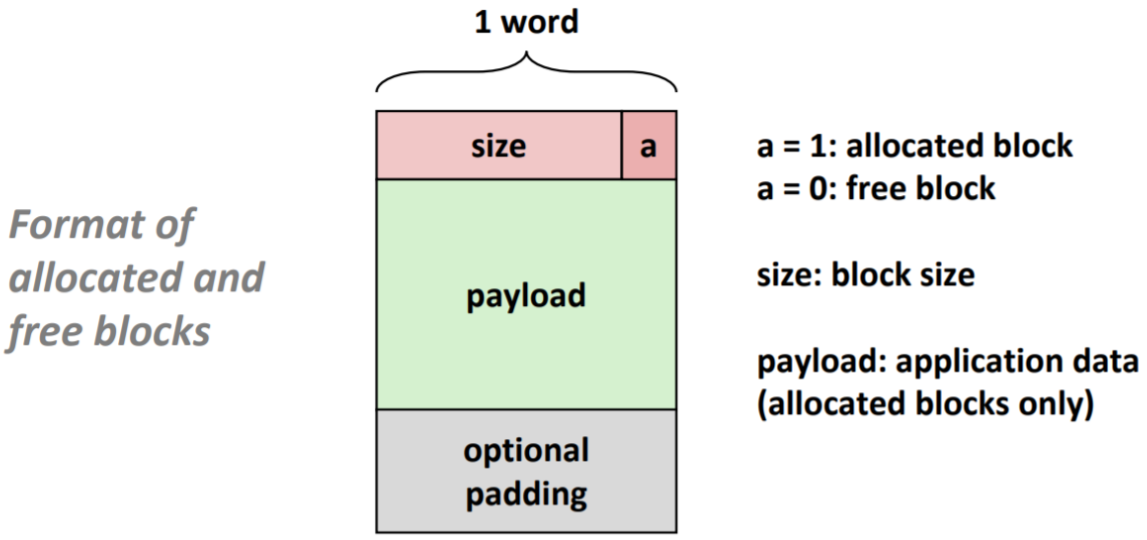
```
struct mem{
    int *memBase;
    int size;
    int freeSize;
    int numBlocks;
    pthread_mutex_t mutex;
    mem(int size);
};
```

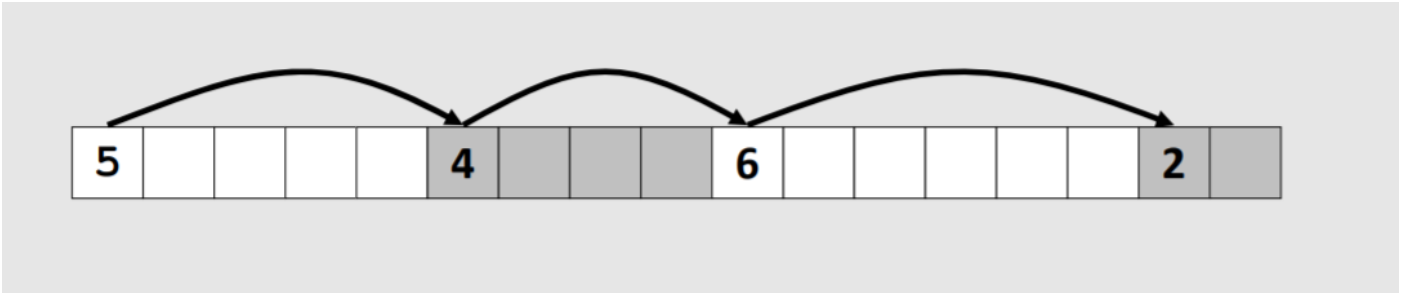
This data structure is used for storing the information regarding initially allocated memory. So that we can convert logical address to physical address.

Data Members

- memBase: Contains the Base address of allocated memory so that we can convert our logical address to the physical address
- size: It contains the total size of our allocated memory so that we can prevent overflow
- freeSize: It shows how much freeSize is available. Can be used to improve garbage collector running time.
- numBlocks: The number of blocks memory is divided into right now. This is used inside the entry condition of the compression block.
- mutex: Our memory can be changed by two threads, one garbage collector during compression, or our library functions createVar and createArr. Therefore we need a mutex lock to handle this critical section.

Format of free and allocated blocks





For moving from this block to next block we can simply add its size to the start of this block and then we will move on to the next block.

If we maintain blocks using these way then there is no need dor any external data structure for maintaining free and allocated blocks inside our memory

Variable representation

```
struct Ptr{
    int idx;
    Type type;
    int size;
    bool isArray;
    int num_elements;
    Ptr(int idx,Type type);
    Ptr(int idx,Type type,int size,int isArray,int num_elements);
};
```

This data structure is used for representing a variable of the array stored in our allocated memory.

Data Members

- Idx: This represents the logical address of a variable. At this index in the symbol table. we can find the physical offset
- type: This represents the type of variable of an array or a variable
- size: Size of the variable or array
- isArray: false if it's not an array otherwise this field is true.
- num_elements: number of elements in an array

Mark Array

```
struct markArray{
    int idx;
    int *arr;
    pthread_mutex_t mutex;
    markArray();
};
```

This data structure is used for marking a variable or array for its deletion by a garbage collector. We have a micro `MAX_SYMBOL_TABLE_SIZE` used to specify the maximum size of the stack. This stack stores physical address rather than logical address because a variable is pooped from the symbol table as soon as its scope ends.

Data Members

- idx: Initial value is 0, this variable represents the top of the mark stack.
- arr: It's a stack, implemented as an array with a size equal to a micro defined in memlab.h `MAX_SYMBOL_TABLE_SIZE`
- mutex: we cannot mark an element and delete it at the same time. As marking involves moving to mark stack and deletion involves removing from mark stack

Library Functions

```
void createMem(int size);
// It takes size in number of words to the total space allocated is 4*size

Ptr createVar(Type varType);
// It takes variable type as input, finds the fist empty slot and assign memory
there
```

```
// If no empty space It prints and error and exits

void assignVar(Ptr &p,int var);
void assignVar(Ptr &p,bool var);
void assignVar(Ptr &p,char var);
// Assign value to variable generated by createVar i.e to p with intensive type
checking

void getVar(Ptr &p,int& var);
void getVar(Ptr &p,bool& var);
void getVar(Ptr &p,char& var);
// Extract variable from Ptr and store it in var

Ptr createArr(int size, Type varType);
// It takes variable type and array size as input, finds the fist empty slot and
assign the memory
// If no empty space It prints and error and exits

void assignArr(Ptr &p,int a[],int n);
void assignArr(Ptr &p,char a[],int n);
void assignArr(Ptr &p,bool a[],int n);
// Assign array to array generated by assignArr with extensive type and size
checking

void assignArr(Ptr &p,int idx,int val);
void assignArr(Ptr &p,int idx,char val);
void assignArr(Ptr &p,int idx,bool val);
// Assign val at index idx on array stored at p

void getArr(Ptr &,int a[],int n);
void getArr(Ptr &,char a[],int n);
void getArr(Ptr &,bool a[],int n);
void getArr(Ptr &,int idx,int &val);
void getArr(Ptr &,int idx,char &val);
void getArr(Ptr &,int idx,bool &val);
// Extract values from array same as assign arr

void freeElem(Ptr &);
void freeElem(int pysical_offset);
// free the space in elem depending on variable or its physical offset

void freeMem();
// destroy all the structure and exit the program
```

```
string getVarS(Type type);
// get variable name in string // just for printing purpose

void *garbageCollector(void *);
// initially calls gc_initialize

void gc_initialize();
// initializes mark data type

void gc_run();
// check all the marked variables // delete them // if compression condition are
met -> compress the memory

void startScope();
// mark a start of new scope

void endScope();
//mark an end of a scope // so the we can mark all variables created between this
call and last startScope call

void compactMem();
// contains compaction logic // discu
```

Garbage Collector

Garbage Collector consists of various functions including startScope(), endScope()

The garbage collector's job is to clear marked memory blocks and compact the memory to join two continuous memory blocks into one

startScope() : User can call to indicate start of a scope. Recommended to be called before any function call. It adds a -1 in symbolTable

endScope() : User can call to indicate end of a scope. Recommended to be called after function returns and return value (if any) has been stored. It marks variables that have gone out of scope by removing them from the symbol table and adding them to the mark stack.

Rest all the functions are specified in the functions section

Statistics of Garbage Collector

Running time

- Demo1.cpp

- Time taken without GC (avg over 100 runs) = 0.53 sec
- Time taken with GC (avg over 100 runs) = 0.54 sec

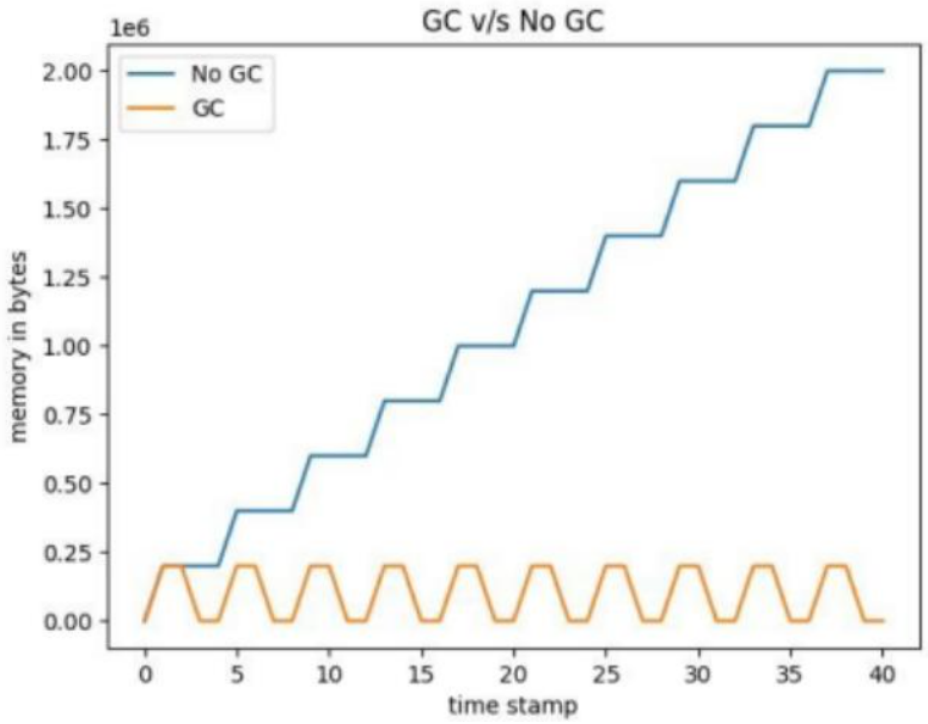
As we can see there is very little overhead in the compression step as my compression is O(k) and the value of k remains very close to 5 if we keep on manually freeing elements, without manual clearing. value of k will increase exponentially and the overall code will slow down by 100 times

- Demo2.cpp
  - Time taken without gc (avg over 100 runs) = 0.009785 sec
  - Time taken with gc (avg over 100 runs) = 0.00187 sec

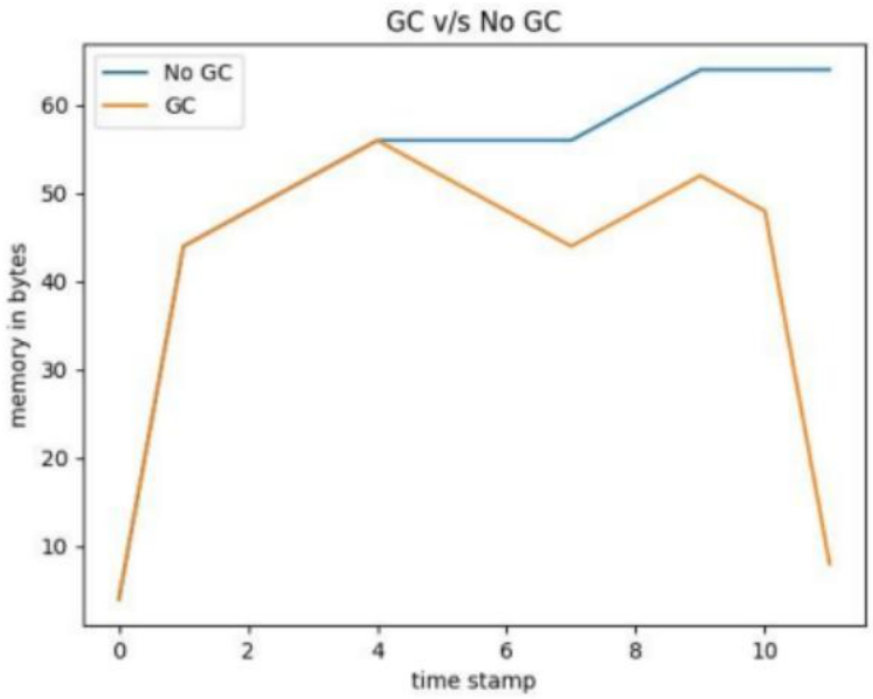
In all the two demos, it can be observed that running the program with garbage collection enabled takes longer than running it without it. However, the time difference is not significantly large and along with it, we get more free memory to work with during execution. Garbage collection is quite fast as our freeElementMem() function frees the associated memory with a symbol in constant time

Memory Footprint

- demo1.cpp : This program has a high running time and higher memory requirements and thus the effect of garbage collection is very pronounced.
  - Without garbage collection: The program's memory footprint increases as it runs for longer because the memory used by it is never freed up
  - With garbage collection: The program's memory footprint increases first then the garbage collector wakes up and so the footprint decreases, and this same behavior is followed periodically. Thus, in the end, we have significantly more free memory than in the previous case. We save almost 1 MB of memory



- demo2.cpp: This program has a very low running time and lesser memory requirements and thus the effect of garbage collection is not very pronounced.
    - Without garbage collection: The program's memory footprint increases as it runs for longer, because the memory used by it is never freed up
    - With garbage collection: The program's memory footprint increases and the garbage collector doesn't even have the time to wake up (as it's a short program), so memory is not freed up.



Compaction logic

compaction is done by garbage collector thread. compaction or compression of memory is a heavy process that stalls the entire process. so we do compression-only when we have deleted at least one variable and the number of blocks in our memory is greater than the total space divided by AVERAGE_FREE_BLOCK_SIZE (Micro defined in memlab.h with an experimental value of 100000) (`memPtr->size/AVERAGE_FREE_BLOCK_SIZE`)

If both the conditions are true:

- If the prevOffset is not set and we encounter a free block, set it.

- If it's set and the current block is free, then simply add the current size to the size of the block stored in prevOffset.
- If the current block is allocated, then unset the prevOffset.
- Move to the next block and repeat.

The entire process is expensive as it takes O(k) time where k is the number of blocks present which can have a maximum value equal to the size allocated initially by 2

Locks in the system

We have used in total two mutex locks, one inside mark structure and the other inside memory structure.

Mark Mutex

This mutex lock is used while marking variables, and arrays. That is poping them from the symbol table and inserting them inside the mark table. To prevent multiple simultaneous access of the mark stack

Mem Mutex

Garbage Collector Compaction algorithm. Changes memory structure. So during compaction. We cannot create new Variables or arrays so because of that we need a mutex lock. To prevent simultaneous creation and compaction and to ensure that memory structure remains intact.