Operating Systems Lab Assignment 5

Group 40
Shristi Singh (19CS10057)
Mayank Kumar (19CS30029)

Structure of Internal Page Table

The class declarations made for Page Table Entry and Page Table are as follows:

```
class PageTableEntry {
public:
    long int offset;
    int dtype;
    string name;
    int szarr;
    bool valid;
    bool marked;
    PageTableEntry(long int, int, string, int, bool=true);
    PageTableEntry(const PageTableEntry&);
    PageTableEntry();
};

class PageTable {
public:
    PageTableEntry pte[MAX_SIZE];
    PageTable* parent;
    long int entry_count;
    long int max_size;
    PageTable(PageTable* _parent, long int _max_size);
    PageTableEntry* lookup(string _name);
    int insertEntry(const PageTableEntry &p);
    void printTable();
};
```

Data Members in Page Table Entry
offset : stores local address of the variable
dtype : data type of variable (-1 for int, -2 for char, -3 for medium_int, -4 for bool)
name : name of the variable
szarr : number of elements in array; if it is variable, szarr is 1
valid : if the entry is valid or not
marked : if the entry is marked

Member Functions in Page Table Entry
PageTableEntry() : default constructor
PageTableEntry(long int, int, string, int, bool=true) : constructor
PageTableEntry(const PageTableEntry&) : copy constructor

Data Members in Page Table
pte[MAX_SIZE] : array of Page Table Entries
parent : parent of the Page Table, if any
entry_count : number of entries in Page Table
max_size : maximum size of Page Table

Member Functions in Page Table
PageTable(PageTable* _parent, long int _max_size) : constructor
PageTableEntry* lookup(string _name) : to find the entry with given name
int insertEntry(const PageTableEntry &p) : to insert the entry in Page Table
void printTable() : to print the Page Table

The Page Table is a contiguous memory segment, and the page and frame are of 4 bytes each. Whenever we have to insert an entry in the Page Table, we have to find an index which is unallocated. For this, we use the Best Fit algorithm, which returns the local address of a partition which is the smallest sufficient partition among the free available partitions. This address is then used to create a variable or an array.

Additional Data Structures and Functions

We also use three global arrays for the following:

MemSeg: Memory segment allocated by createMem function
MemSegEnd: Marks the memory segment end according to MemSize
MemStatus: Array for checking the status of memory frames, if it is alloted or free

Some other class/struct/functions used are:

- medium_int
  char num[3]
  medium_int(int = 0) : constructor for the medium_int class
  medium_int(const medium_int &) : copy constructor
  ~medium_int() : destructor for the class
  medium_int operator+(const medium_int &) : overloads the ‘+’ operator
  medium_int operator-(const medium_int &) : overloads the ‘-’ operator
  medium_int operator*(const medium_int &) : overloads the ‘*’ operator
  friend ostream &operator<<(ostream &, const medium_int &) : print the medium_int class variable

- Stack
  int _elems[MAX_STACK_SIZE] : array storing the index at which a Page Table Entry has been inserted in a Page Table
  int _top : stores index of the latest element in _elems
  Stack() : constructor for the struct
  void push(int elem) : adds elem to the _elems stack
  int pop() : removes the top element of the _elem stack and prints it
  int top() : returns the top element of the stack i.e. the latest added element to _elem stack

  For maintaining scope information and garbage collection, we need to keep track of variables in order of their declaration. A global stack ST is used where -1 is pushed to denote start of scope and the index for variables are pushed as they are declared.

- int readInt(string name, int index = 0) : returns the data stored in given int variable name. For array, it returns the data at given index (0 by default)

- char readChar(string name, int index = 0) : returns the data stored in given char variable name. For array, it returns the data at given index (0 by default)

- medium_int readMediumInt(string name, int index = 0) : returns the data stored in given medium_int variable name. For array, it returns the data at given index (0 by default)

- bool readBool(string name, int index = 0) : returns the data stored in given bool variable name. For array, it returns the data at given index (0 by default)

- bool checkAssignVar(PageTableEntry *var, int dt) : checks if the given Page Table Entry exists, and is of appropriate data type

- bool checkAssignArr(PageTableEntry *var, int index, int dt) : checks if the given Page Table Entry exists, is of appropriate data type, and the array index is not out of bounds

- bool checkRead(PageTableEntry *var, int index, int dt) : checks if the given Page Table Entry exists, is of appropriate data type, and the array index is not out of bounds

- bool mem_status(long int offset) : checks if the frame in the memory segment at a particular offset is free or not

- void mem_set(long int offset) : sets the status of the frame in the memory segment at a particular offset

- void mem_free(long int offset) : clears the status of the frame in the memory segment at a particular offset

- void initScope() : initialize a scope by pushing -1 into the stack

- void mark() : pops the variables from the stack till we get -1. Also, marks the elements which are popped in the page table

- void sweep() : frees the elements in the page table that are marked true
- void endScope() : ends the scope using mark and sweep functions

Garbage Collection Mechanism

Garbage collector thread is initialized after we allocate memory in the createMem function by calling gc_initialize() function. We perform an efficient version of the mark step where we save repeated traversals through the stack. Each function starts with initScope() being called, which pushes -1 to the global stack, and as variables are created in this scope, its index is pushed into the stack and the mark bit is set to false by default.

To end the scope of the variables, endScope() function is explicitly called, where the elements are popped until we get -1, and the mark bit of the popped variables in the page table is set to true. This saves the overhead of doing multiple marks and unmarks for active variables.

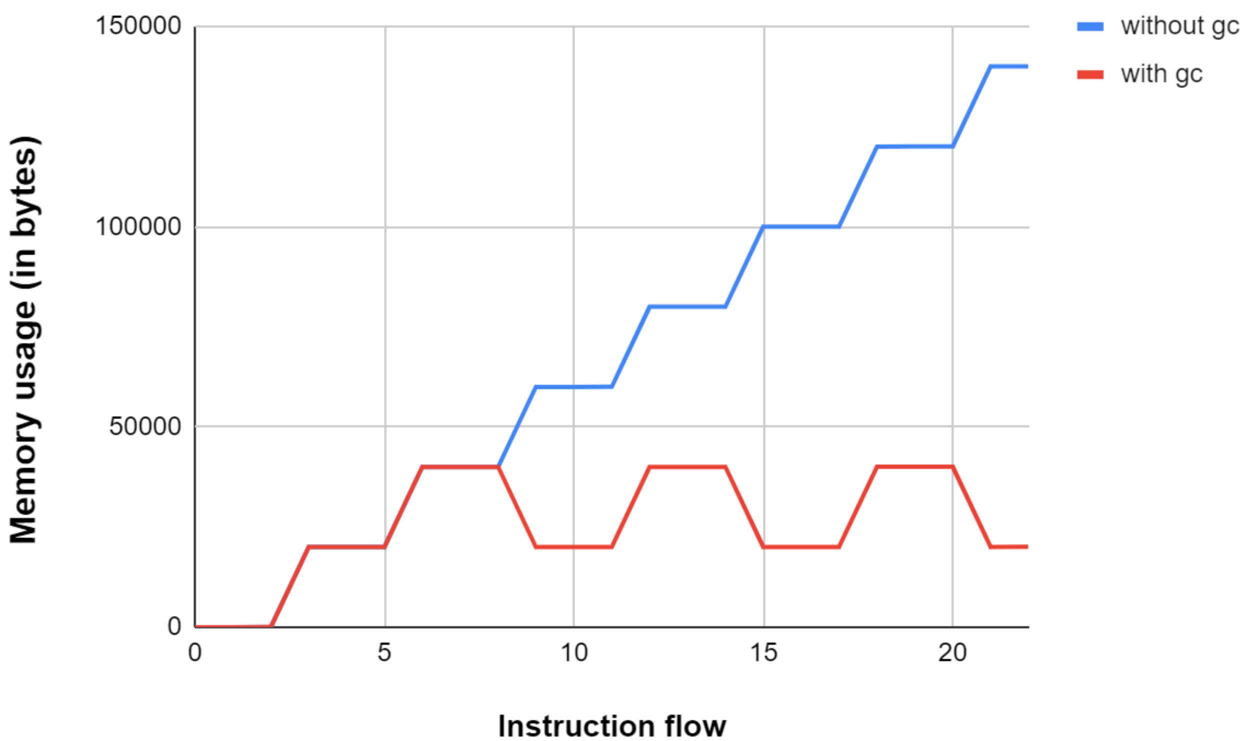
For garbage collection, the entire page table is scanned and the entries which have their mark bits as true and valid bits as false are freed. This is essentially the sweep step in the mark and sweep algorithm.

The thread also periodically checks a metric to find the extent of memory fragmentation and if it is high it calls the gc_compaction() function which performs the memory compaction.

Garbage Collection Compaction Mechanism

Whenever the memory is freed and allocated dynamically, holes are created in our memory. To run compact in Garbage Collection, the gc_compaction uses LISP2 mark-compact garbage collection algorithm to shift the live objects in memory together so the fragmentation is eliminated. At first, it changes the local address of the remaining Page Table Entry to use the freed space, then it copies the old memory space to the new address, and then resets the memory status accordingly. Basically, all the holes (free space in memory) are combined into one big memory block. This ensures that the used memory is as contiguous as possible so that bigger objects can be allocated.

Memory Footprint with and without Garbage collection



Memory Usage Plot for demo1.cpp

The above graph for demo1.cpp file shows how Garbage collection largely affects memory usage.
Time taken by demo1.cpp to run with Garbage collection: 9 sec
Time taken by demo1.cpp to run without Garbage collection: 5 sec
Thus, time taken for Garbage collection is around 4 sec

Time taken by demo2.cpp to run with Garbage collection: 0.17 sec
Time taken by demo2.cpp to run without Garbage collection: 0.08 sec
Thus, time taken for Garbage collection is around 0.09 sec

Use of locks

Yes, we did use locks in our library, so that multiple processes don’t access the same resource at any point of time and there is no confusion. During gc_compaction, the garbage collector thread changes the position of blocks in memory. Also, during normal garbage collection, the status of blocks is changed from allocated to unallocated. The coalescing of blocks is also performed in this step. If the main thread tries to access or create a variable from the main thread when the operations of garbage collector or compaction is underway, there might be data corruption or data loss which is undesirable. So, if the main thread accesses the page table after it has been updated but memory hasn’t been modified, then we will not be able to read from the right location in memory and might write to a wrong location. Hence, we require a lock on the symbol table.