Operating Systems Laboratory (CS39002)
Assignment 5

Group 26
Seemant G. Achari (19CS10055) Rajas Bhatt (19CS30037)

Design Considerations for Assignment 5

A: Structure of the internal page table

Hierarchy: Each function creates a new page. If more pages are required, more pages are allocated by a function. The size of a page is specified as 128 words (Each word is 4 bytes as per assignment definition). The number of pages are variable, i.e. the total allocation size divided by the page size.

```
#define MAX_FUNC 100    // Maximum number of functions
```

There is also an upper bound on the number of functions as shown above. The structure of a page is shown below. A page table is a doubly linked list of pages, to ensure easy iteration.

```
typedef struct pageNode{
   bool markBit;            // Mark bit for mark and sweep
   usedMem *pointer;        // Pointer to a used memory node
   int16_t currOffset;
   int16_t funcIndex;       // Function this page belongs to
   int index;               // Index in the queue of pages
   struct pageNode *prev;   // Previous Page in the Table
   struct pageNode *next;   // Next Page in the Table
}pageNode;

pageNode **pageStackHead;   // Head of the linked list of pages
pageNode **pageStackTail;   // Tail of the linked list of pages
pageNode *pageNodes;        // Pool of pageNodes
int *pageQueue;             // Queue of page indices
int *pageQueueHead;         // Head of the queue
int *pageQueueTail;         // Tail of the queue
int  pageQueueCount;         // Pages in used currently
```

The above shows the structure of a page. It can be seen that each page has a mark bit which will be used in the Mark and Sweep Algorithm. It also has a pointer to a used memory node. The page table is basically a doubly linked list of pages, arranged in the form of a stack named `pageStackHead` as can be seen above. Each page also has a `currOffset`, which forms a boundary between allocated and non-allocated memory inside a page. `pageQueue` is a circular queue of integers, in which we will put indices of the node being used/freed. This index denotes the position of the node from `pageNodes`. `pageQueueCount` is used to measure how many nodes are being used currently (active pages).

In addition to a page table, we also have a symbol table, which stores symbols. A symbol table is implemented as a Hash Table. The value of the number of rows is set to **HASH** (a sufficiently large prime number).

```
#define HASH 113        // Number of rows in the Symbol Table (Hash Table)
```

```
typedef struct ST{
   STNode *table[HASH];
}ST;
ST *symTable;        // Global Symbol Table
```

The symbol table points to a chained list of Symbol Table Nodes (**STNode**). Each **STNode** has the internal `counter` value of a variable and the ID of the function it belongs to. It also has a pointer to the global variable stack (**globalStack**). For an array, we have exactly one symbol, but we have a list of pages which are denoted by the **arrayBook** structure. `pageOffset` stores the word position in the page where the symbol is referenced, while `pageIndex` stores a pointer to the page.

```
/* Symbol Table Node */
typedef struct STNode{
   unsigned long long counter; // Counter assigned to the variable
   int16_t functionID;         // The function the Symbol belongs to
   pageNode*  pageIndex;
   globalStack* stackLoc;      // Pointer in the globalStack entry
   int8_t type;                // 1, 2, 3 or 4: defined above
   int16_t  pageOffset;        // Starting from which position in the page
   int size;                   // The size of the array (1 for non-arrays)
   bool isArray;               // Whether symbol is an array or not
   int16_t index;
   struct arrayBook *book;
   struct STNode *next;
}STNode;

STNode *symNodes;               // Pool of nodes to be used
int16_t *symQueue;              // Queue of indices which can be used
int16_t *symQueueHead;          // Head of the queue
int16_t *symQueueTail;          // Tail of the queue
int16_t symQueueCount;          // Number of symbols
```

B: Data structures and functions used

In addition to the Symbol Table and the Page Table, we have the following data structures.

1. **freeMem**: This simulates free Memory. It is a singly linked list. Each **freeMem** is basically a modified version of a free memory frame. The `offset` denotes the index of the first byte of this frame and the `index` is used to keep track of the pool of freeMem nodes (which are allocated from a circular queue).

   ```
   typedef struct freeMem{
      int offset;     // like 0, 512, etc.
      int index;      // Index in the queue of freeMem nodes
      struct freeMem *prev;    // Previous freeMem node
   }freeMem;

   freeMem **freeHead;    // Header of the linked list
   ```

```
freeMem *freeNodes;    // The pool of nodes to be used
int *freeQueue;        // Indices queue
int *freeQueueHead;    // Head of the indices queue
int *freeQueueTail;    // Tail of the indices queue
int freeQueueCount;    // Number of free pages
```

2. **usedMem**: This simulates a used Memory. It is a doubly linked list, because we need to do garbage collection on it. The structure is similar to a **freeMem**, except the fact that we have another pointer, `next`.

```
typedef struct usedMem{
   int index;     // Index in the queue of usedMem nodes
   int offset;    // like 0, 512, etc
   struct usedMem *next;   // Next freeMem node
   struct usedMem *prev;   // Previous freeMem node
}usedMem;

usedMem **usedHead;
usedMem **usedTail;
usedMem *usedNodes;
int *usedQueue;
int *usedQueueHead;
int *usedQueueTail;
int  usedQueueCount;      // Number of used pages
```

3. **globalStack**: This is the global stack of symbols, formulated as a doubly linked list. Each symbol has a back pointer (`stnode`) to a Symbol Node. An index to select from the symbol queue and pointers to the next and previous entry, here the top and the bottom entry.

```
typedef struct globalStack{
   STNode *stnode;          // Back pointer to the symbol table entry
   int16_t index;               // Index in the symbol queue
   struct globalStack *next;    // Next entry
   struct globalStack *prev;    // Previous entry
}globalStack;

globalStack **globalStackHead;  // Head of the queue
globalStack **globalStackTail;  // Tail of the queue
globalStack *globalStackNodes;  // All the nodes allocated
int16_t *globalStackQueue;      // Indices in global stack
int16_t *globalStackQueueHead;  // Head of the queue
int16_t *globalStackQueueTail;  // Tail of the queue
int16_t  globalStackQueueCount; // Count of the number of used nodes
```

4. **arrayBook**: This data structure stores all the pages required by an array in terms of a singly linked list.

```
typedef struct arrayBook{
   pageNode* pageIndex;    // The page it points to
   int16_t pageOffset;
   int index;
   struct arrayBook *next; // The next page of the array
}arrayBook;

arrayBook *bookNodes;int *bookQueue;int *bookQueueHead;
int *bookQueueTail;int  bookQueueCount;
```

5. Other structures like **funcT** which store a function table and **funcInfo**, which store more information about a function, like the number of pages in the function and the ID of the function.

We use the following additional functions:

1. **gc_run**: This runs the mark and sweep algorithm for garbage collection
2. **compact**: This runs compaction which is to be performed in addition to garbage collection.
3. **gc_initialize**: This runs the runner function for the gc thread which does the garbage collection.
4. **demandPage**: This is used to ask for an empty page from the free memory.
5. **getSTEntry**: This is used to get the STNode pointer corresponding to a given counter value from the symbol table.
6. **initFunction**: This is used in the beginning of a function and is used to create a new entry in the function table and put a NULL value into the global stack, so that the function boundary can be detected.
7. **returnFromFunc**: This is used to return from a function. The return value is saved and variables are popped from the stack.
8. **getSize**: This is a helper function which gets the size of a data type.
9. **getArrayEntry**: Given an array variable and an index, this function gets the value at that index as a 32 bit integer.
10. **getVar**: This function returns the value of a variable as a 32 bit integer.

C: Impact of Garbage Collection

In our system, garbage collection has mainly two functions. It periodically frees the memory freed up by the user using freeElem and it frees up the memory used in a function after during its return. Both of these use cases largely help in dynamic allocation and reuse of memory. It allows the user to manage memory more efficiently in the functions internally, deallocating the memory which is not needed anymore and allocating more memory for different data structures/variables.

For demo1 the impact is very crucial, since we are allocating huge chunks of memory, which cannot be accessed across functions and hence, if not freed up will lead to hogging up.

For maximum array size 100000, around 2200 pages were used without garbage collection.
For maximum array size 100000, a maximum 202 pages were used at a point with garbage collection.

The observation is consistent with our claim, as memory is getting freed up after the scope of each function and we can reuse the pages.

D: Logic for the compact() function

We maintain special data structures to keep track of used memory and free memory. Both of them store the offsets of the physical memory they are associated with. We sort the free memory list in the ascending order of its offsets and the used memory list in the descending order of its offsets. Now traversing from the start of both the lists, if offset in free memory < offset in used memory, we copy the content of the used memory fragment to the free memory fragment and swap the offsets in both. The pages associated with the used memory segments have a pointer to these nodes in the list and hence we would not have to make any further change in the mapping.

E: Locks Used

We have decided to use the following mutex locks:

1. **lock**: For simplicity, we have decided to keep only one mutex lock for the entire shared memory. Even though this is not very optimal, our garbage collection takes a short amount of time to complete in practice.
2. **input**: This lock is used to hold back garbage collection when input is being taken from the user, say in the case of the demo2.c file. If the user delays input, garbage collection may start at the same time and this may result in problems.

Gc compact average time
Memory footprint : number of pages