Operating Systems Laboratory
Assignment 5 - Report

Ashutosh Kumar Singh - 19CS30008

Vanshita Garg - 19CS10064

Contents

1 Structure of Internal Page Table

1.1 Page Table Entry

Each page table entry is of the following type:

```
struct PageTableEntry {
    unsigned int addr : 30;
    unsigned int valid : 1;
    unsigned int marked : 1;
};
```

Each page table entry is a 32-bit unsigned integer. The most significant 30 bits are used to store the `addr` field, the next bit is used for the `valid` field and the last bit is used for the `marked` field.

Data Members:

- `addr`: It is the offset (in words, 1 word = 4 bytes) from the beginning of the memory address of the `malloc` memory segment. Using 30 bits, we can use this representation for an address space of size $2^{30} * 4$ bytes = 4 GB.
- `valid`: This bit denotes whether we presently have a valid variable/array at this entry. It is 1 if the variable/array is valid, 0 if it is invalid.
- `marked`: This bit is 1 if the variable/array at this index is currently accessible, and 0 if it has gone out of scope and is currently inaccessible. This information is needed by the garbage collector to free up space for the elements which have gone out of scope.

Functions:

- `init()`: Initializes all data members to 0.
- `print()`: Displays the entry in a suitable format.

1.2 Page Table

The page table has the following structure:

```
struct PageTable {
    PageTableEntry pt[MAX_PT_ENTRIES];
    unsigned int head, tail;
    size_t size;
    pthread_mutex_t mutex;
};
```

The page table is essentially an array of page table entries. Whenever, we need to add a new entry to the page table, we find an empty spot in the array (`valid` bit = 0) and insert the element at that position. But, this process of finding an empty spot is done using a clever trick explained below.

Data Members:

- `pt`: The array of page table entries.

- `head` and `tail`: For maintaining which indices in the page table are free, we use an implicit list (without any space overhead). `head` stores the first index which is free, and `tail` stores the last index that is free. Now, the `addr` field is utilised only when that position is filled with some entry. So, for every free index, starting from `head`, we store the next index that is free in the `addr` field itself, creating an implicit list. Thus, whenever a new entry has to be inserted, it is inserted at the index `head`, and `head` is moved one place forward by using `head = pt[head].addr`, also, whenever a spot in the page table is freed, it is added to this implicit list after `tail`. This allows us to find an empty spot and insert a new entry in $O(1)$ time.
- `size`: The present number of valid entries (filled spots) in the page table.
- `mutex`: Mutex lock to ensure mutual exclusion.

Functions:

- `init()`: Initializes all entries and the mutex lock.
- `insert(unsigned int addr)`: Adds a new entry to the page table.
- `remove(unsigned int idx)`: Removes an entry from the page table.
- `print()`: Displays the entries of the page table in a suitable format.

1.3 Translation Mechanism

- Each variable is assigned an integer counter which is a multiple of 4 (due to word alignment), which serves as the local address.
- This counter value divided by 4 gives us the index in the page table where the entry for this variables/array is present.
- The `addr` field in the page table entry stores the offset in words from the start (base) address in the `malloc` memory segment.
- The offset is then added to the start address to get the address of this variable/array in the `malloc` memory segment.

Functions:

- `counterToIdx(unsigned int p)`: Converts the counter value to the index in the page table array.
- `idxToCounter(unsigned int p)`: Converts the index in the page table array to the counter value.

2 Additional Data Structures

2.1 Memory and Implicit Free List

For finding free blocks in the memory, allocating free blocks to new variables/arrays, and for garbage collection while compaction, we need to know which blocks in the memory are free and/or which are used. For this, we use an implicit free list, i.e., we store this information about the blocks in the `malloc` memory segment itself (see Figure 1).

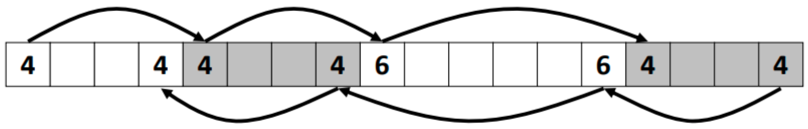

Figure 1: Implicit Free List

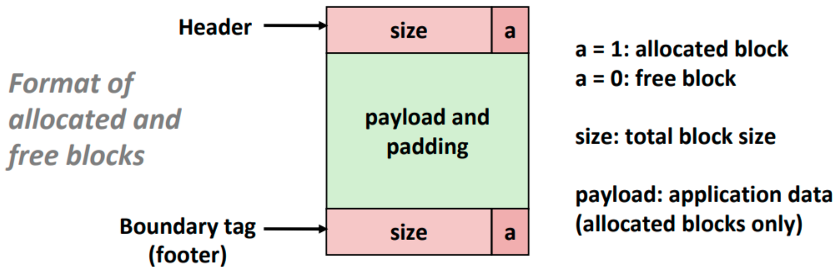

Figure 2: Block Structure

From now, we will talk in terms of words (4 bytes). Suppose we have a memory block of x words, starting from word offset 1 and going till offset x (inclusive). So, we will have a header of 1 word (= 32 bits) at word offset 0, the first 31 bits of which will store the size of the block in words (x in this case), and the least significant bit will store whether the block is free (0) or allocated (1). We will also have a footer of 1 word at offset $x + 1$ which will store exactly the same information as the header.
So, using the header we can traverse the memory segment in blocks, get the size of the blocks, and find free blocks of a required size. Say, we are at address `p` in the memory segment, so the next block will start from `p + (*p >> 1)`, whether the block is free or allocated can be known by checking `(*p & 1)`.
The footer is required to know about the previous block, which is needed while coalescing blocks with its next and previous blocks after freeing.
Since we store this bookkeeping information in the memory segment itself, if the user requests for b bytes of memory, then we allocate $(b * 1.25)$ bytes.

2.1.1 Operations Using the Implicit Free List

Finding Free Block
We use the first-fit strategy for finding an empty memory block of atleast the required size. We keep on traversing the memory segment, and advancing the current pointer using `p = p + (*p >> 1)`, till we find a free block that satisfies our requirements. This might take $O(N)$ time, where N is the number of blocks. However, it is actually much less because we have large blocks in the memory, and the pointer advances to the next block.

Allocating a Block
Since the allocated space might be smaller than the free space, we might have to split the block into an allocated and a free block. This requires changing 2 headers and 2 footers, which can be done in $O(1)$ time (see Figure 3).

Freeing a Block
For freeing a block, just changing the allocated bit in the header and footer to 0 is not enough. We need to coalesce the block with the previous and next blocks if they are empty. This can also be done in $O(1)$ time.

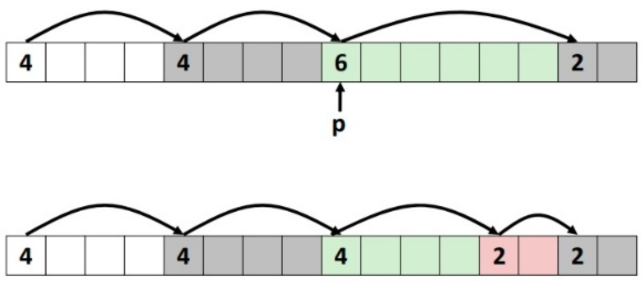

Figure 3: Allocating a Block

```c
struct Memory {
    int *start;
    int *end;
    size_t size;
    size_t totalFree;
    unsigned int numFreeBlocks;
    size_t currMaxFree;
    pthread_mutex_t mutex;
};
```

Data Members:

- `start`: The start address of the `malloc` memory segment.
- `end`: The address just after the last block of the memory segment.
- `size`: The size of the memory segment (in words).
- `totalFree`: Total amount of free space in the memory segment (in words).
- `numFreeBlocks`: The number of free blocks (or holes).
- `currMaxFree`: Size of the largest free block.
- `mutex`: Mutex lock to ensure mutual exclusion.

Functions:

- `init()`: Allocates the required amount of memory and initalizes the mutex lock.
- `getOffset(int *p)`: Converts the absolute address in the `malloc` memory segment to offset from the start.
- `getAddr(int offset)`: Converts the offset to the absolute address in the `malloc` memory segment.
- `findFreeBlock(size_t sz)`: Finds a free block of memory for `sz` words.
- `allocateBlock(int *p, size_t sz)`: Allocates memory for `sz` words at address `p` and sets the appropriate headers and footers.
- `freeBlock(int *p)`: Deallocates the memory block at address `p` and sets the appropriate headers and footers.
- `displayMem()`: Display the current memory blocks.

2.1.2 Advantages of the Implicit Free List

- We do not require any external data structure to keep track of the free and used blocks, and hence it is very simple and clean to implement.
- Compare it to a linked list implementation for the free and used blocks. So, each node would have to store an address, size of the block, and a pointer to the next node, requiring a total of 12 bytes. In the implicit free list, we have an overhead of 8 bytes per block (header and footer). Also, finding a free block would take linear case in a linked list, and freeing would also take linear time if we were to coalesce as then we would have to keep the linked list sorted by address. We achieve freeing a block in $O(1)$ time using the implicit free list. Also, something like a balanced binary search tree would come with its own share of difficulties like balancing, heavy implementation, and having to modify the entire tree during compaction.

2.2 Global Variable Stack

The global variable stack keeps track of which variables/arrays are in scope presently, and are accessible. Whenever a new variable is created it is pushed onto the stack, and when it goes out of scope it is popped. It is required for the garbage collector to know which variables have gone out of scope. The details of detecting the scope are given in section 3 (Garbage Collection Mechanism).

```
struct Stack {
    int st[MAX_STACK_SIZE];
    size_t size;
};
```

Data Members:

- `st`: The contents of the stack, stored in an array.
- `size`: The present number of elements in the stack.

Functions:

- `init()`: Initializes the stack.
- `top()`: Returns the element at the top of the stack.
- `push(int v)`: Pushes an element onto the stack.
- `pop()`: Pops the element at the top of the stack.
- `print()`: Displays the contents of the stack.

2.3 Combined Data Type - `MyType`

We have 4 data types - `char` (1 byte), `medium int` (3 bytes), `int` (4 bytes), and `boolean` (1 bit). We encapsulate all these into a type called `MyType`. This also allows us to store more data which has to be stored per variable/array.

```
struct MyType {
    const int ind;
    const VarType var_type;
    const DataType data_type;
    const size_t len;
};
```

- `ind`: The counter for each variable/array, which is always a multiple of 4. Dividing this by 4 gives us the index of the entry for this variable in the page table, thus allowing $O(1)$ page table lookup time.
- `var_type`: Indicates whether the symbol is a `PRIMITIVE` or an `ARRAY`.
- `data_type`: The data type of the variable or the data type of the elements of an array.
- `len`: For a `PRIMITIVE`, it is 1, for an `ARRAY`, it is the number of elements in the array.

This `MyType` structure allows us to write:

```
MyType a = createVar(INT);
MyType arr = createArr(CHAR, 10);
```

3 Garbage Collection Mechanism

- The garbage collector runs as a separate thread, which is initialized by calling `gcInitialize`, and uses a mark and sweep algorithm.
- The marking and unmarking is performed in an amortized manner. Whenever we enter a function or a scope, we call the `initScope` function which pushes -1 on the global stack. This -1 acts as a marker for the current scope. Then, whenever a variable/array is created, we set the `mark` bit for it to 1 in the page table, and its counter (`ind` in `MyType`) is pushed to the stack. When the function or scope ends, we keep popping from the stack, and keep on setting the `mark` bit to 0 for these variables, till a -1 is encountered in the stack. In this manner, all accessible variables are marked, and all unaccessible variables are unmarked.
- The garbage collector thread sleeps for a specified period of time (10 μs in our case), after which it wakes up, performs its task, and goes to sleep again, and this keeps on repeating. When the garbage collector wakes up, it calls the function `gcRun`. It then scans the entire page table, and frees those variables/arrays for which the `valid` bit is 1 and the `mark` bit is 0. It may also compact the memory depending on whether some criteria is satisfied (described in the next section).
- To free a variable/array, the page table entry is invalidated (`valid` bit set to 0), and that spot will now be treated empty. Also, the memory in the `malloc` memory segment is deallocated by changing the headers and footers and coalescing if required.

Functions:

- `gcRun()`: Performs the actual garbage collection (mark and sweep), and checks if compaction needs to be done, and if yes, calls the appropriate function for compaction.
- `gcActivate()`: A function provided to the user to signal the garbage collector to run.
- `gcThread(void *arg)`: The function that is run by the garbage collection thread. The garbage collector sleeps and periodically wakes up to call `gcRun()`.
- `initScope()`: Indicates that a new scope has been entered.
- `endScope()`: Indicates that the current scope has ended.

4 Compaction in Garbage Collection

4.1 Logic for Performing Compaction

- Compaction is done when external fragmentation becomes very high, and there are many holes in the memory.
- Let us call `totalFree` = total amount of free memory (in words), and `currMaxFree` = largest free block currently present. We calculate the ratio `totalFree/currMaxFree` and if this comes out of be greater than a specific threshold (3 in our case), then we perform compaction.
- We also try to compact the memory when we fail to find a free block for a variable/array, and then try to find a free block again after compaction. If we fail at the second attempt too, that means the memory is full. This is demonstrated in the file `demo4.cpp`.
- The intuition behind this logic is that we compact when external fragmentation is high, which means that in total there is sufficient memory (`totalFree`), however this total free memory is spread across many holes, and as a result the largest free block (`currMaxFree`) is not very big, and hence allocating memory any further to variables/arrays might be unsuccessful. Hence, we consider this ratio. When the largest free block is of sufficient size, then the ratio will be low, and we will not compact, however when the total free memory will be sufficient and the largest free block will not be very large, then the ratio will be high and we will compact.
- However, computing the exact value of the largest free block would require a pass over all memory blocks using the implicit free list, and since the number of blocks (and holes) would be large when compaction is needed, this step could be expensive. So, we use certain heuristics for keeping track of the current largest block. Note that the largest free block will be bigger than the average size of a free block (the value of which can be computed exactly). So at each updation step we take a max with the average free block size as well. Apart from that, when memory is being allocated from the largest block, then we update its value, and take a max with the average too. Also, when a block is being freed, then if the size of this free block is larger than the current maximum, then we update the current maximum, and then also take a max with the average free block size. This is how we approximately keep track of the size of the largest free block.

4.2 Compaction Algorithm

- First, we calculate the new memory addresses (offsets from start) for all the blocks. Essentially, if the total free space encountered till now is x words, the present block will be shifted x words earlier. We store this new offset in the footer itself, to avoid any space overhead. We can do this because we do not need the footer during compaction.
- Then we update the page table entries with the new addresses. For each valid entry, we have the current memory address, so we go there, and from there we can go to the footer as the header has the size of the block. We then read the new memory address (offset) from the footer and update the entry in the page table with the new value.
- Finally, we perform the operation of moving all blocks to their new locations. For this, say if the current block is free and the next block is allocated, we swap these two blocks, and then coalesce the free block with any next free block (if any). In this manner, we keep shifting blocks to smaller addresses till ultimately all holes are removed and we have one large block of free memory at the end.

Functions:

- `calcNewOffsets()`: Calculates the new memory addresses (offsets from start) for all the blocks.
- `updatePageTable()`: Updates the page table entries with the new offsets for compaction.
- `compactMemory()`: Performs all the steps mentioned in the third point of the compaction algorithm.

5 Impact of Garbage Collection

5.1 Memory footprint with and without garbage collector

5.1.1 `demo1.cpp`

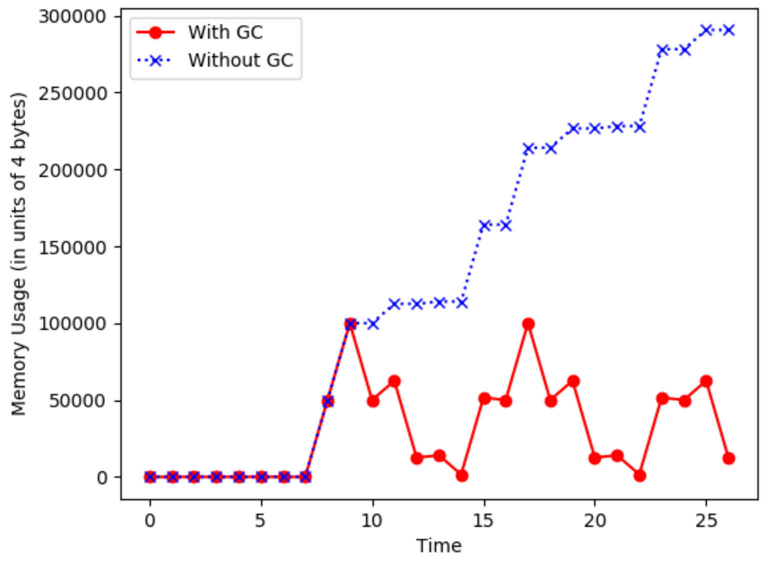

Figure 4: Memory Footprint for `demo1.cpp`

Statistics (all in units of 4 bytes):
With Garbage Collector

Maximum	**Average**	**Std. Dev.**
100028	29999.92	31120.85

Without Garbage Collector

Maximum	**Average**	**Std. Dev.**
290670	129890.44	104425.19

The impact of using the garbage collector is very prominent in this case. Without it, the memory used keeps increasing monotonically, while when the garbage collector is on, the memory footprint is much lower, as it keeps on freeing up the space occupied by variables/arrays that have gone out of scope.

5.1.2 `demo2.cpp`

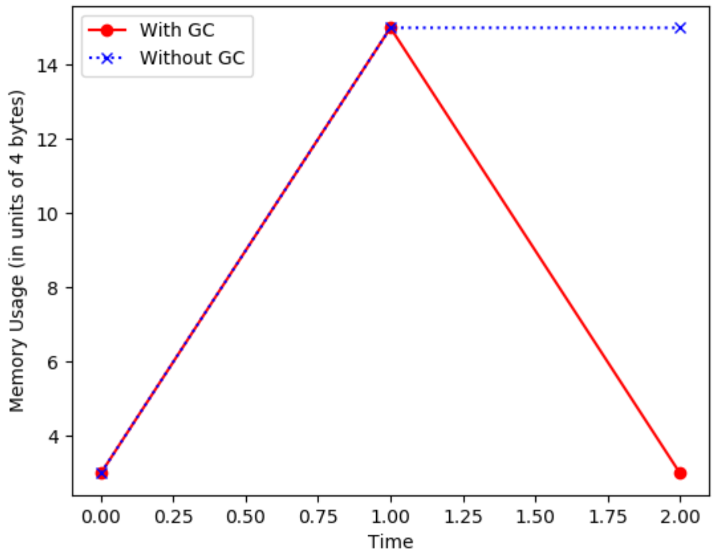

Figure 5: Memory Footprint for `demo2.cpp`

Statistics (all in units of 4 bytes):
With Garbage Collector

Maximum	**Average**	**Std. Dev.**
15	7.0	5.65

Without Garbage Collector

Maximum	**Average**	**Std. Dev.**
15	11.0	5.65

In this case since we do not have very less number of variables, the difference is not very high, however the line plot when the garbage collector is off will always be increasing, however this is not the case for when the garbage collector is on.

5.2 Time for Garbage Collector to Run

5.2.1 `demo1.cpp`

With Garbage Collector

Iteration No.	1	2	3	4	5	6	7	8	9	10
Time (in s)	0.129	0.126	0.128	0.126	0.127	0.125	0.131	0.125	0.137	0.128
Avg. Time (in s)	**0.1282**									

Without Garbage Collector

Iteration No.	1	2	3	4	5	6	7	8	9	10
Time (in s)	0.119	0.115	0.116	0.114	0.116	0.118	0.116	0.114	0.119	0.114
Avg. Time (in s)	**0.1161**									

Hence, time for garbage collector to run in this case = 0.1282 s - 0.1161 s = **0.0121 s**.

5.2.2 `demo2.cpp`

With Garbage Collector

Iteration No.	1	2	3	4	5	6	7	8	9	10
Time (in s)	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.003	0.003
Avg. Time (in s)	**0.0038**									

Without Garbage Collector

Iteration No.	1	2	3	4	5	6	7	8	9	10
Time (in s)	0.004	0.002	0.001	0.002	0.002	0.001	0.001	0.001	0.001	0.001
Avg. Time (in s)	**0.0017**									

Hence, time for garbage collector to run in this case = 0.0038 s - 0.0017 s = **0.0021 s**.

5.2.3 `demo3.cpp`

With Garbage Collector

Iteration No.	1	2	3	4	5	6	7	8	9	10
Time (in s)	0.005	0.005	0.005	0.005	0.004	0.004	0.003	0.003	0.002	0.002
Avg. Time (in s)	**0.0038**									

Without Garbage Collector

Iteration No.	1	2	3	4	5	6	7	8	9	10
Time (in s)	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.003	0.003	0.003
Avg. Time (in s)	**0.0020**									

Hence, time for garbage collector to run in this case = 0.0038 s - 0.0020 s = **0.0018 s**.

Overall, we can see that the running time of the garbage collector doesn't add a large overhead to the total running time. Hence, we can say that it is a good choice to have the garbage collector.

6 Use of Locks in the Library

- We have used mutex locks in the library - one associated with the memory segment, and one with the page table.
- The memory is shared between the main thread and the garbage collection thread, and can be updated by both, hence we need a lock to prevent data inconsistencies. For example, the garbage collector, during compaction changes the position of all the blocks in the memory. Meanwhile, if we try to access or create a variable/array from the main thread, it might lead to several issues. Issues can also happen when the garbage collector is freeing the space for a variable, and the main thread tries to find space to create a new variable/array.
- The page table also needs to be locked at appropriate times, because the addresses in the page table are updated during compaction by the garbage collector thread, and meanwhile trying to access the page table for inserting or removing an entry might lead to data corruption and inconsistencies.