Custom Malloc Function

Executive Summary:

The Malloc Assignment involved implementing a custom memory allocator library capable of managing heap memory on behalf of a user process. The assignment required implementing malloc and free functions along with three additional heap management strategies: Best Fit, Worst Fit, and Next Fit. Additionally, the assignment required the implementation of realloc() and calloc() functions. The objective was to compare the performance of the custom memory allocator against the standard system call malloc() through benchmarking.

Description and Algorithm Explanation:

1] Free block

- Checks if the pointer **ptr** is **NULL**. If it is, the function exits because there's nothing to free.
- If the input pointer is valid, it increments a counter to keep track of the number of successful **free** calls.
- It marks the memory block associated with the pointer as "free" so that it can be reused for future allocations.
- It checks adjacent blocks to see if they are also free. If neighbouring blocks are free, it merges them into a single larger block to avoid fragmentation of memory.

Pseudocode:

```
free(ptr):
   if ptr is NULL:
      return
   num_frees += 1
   curr = BLOCK_HEADER(ptr)  // Get the header of the block associated with ptr
   assert(curr->free == 0)    // Ensure that the block is not already free
   curr->free = true         // Mark the block as free

   free_check = heapList   // Start checking from the beginning of the free list
   while free_check and free_check->next:
      if free_check->free and free_check->next->free:
         // Combine the current free block with the next free block
         free_check->size += sizeof(struct _block) + free_check->next->size
         free_check->next = free_check->next->
         free_check = heapList // Start checking from beginning of free list
         num_coalesces += 1     // Increment the count of coalesced blocks
         num_blocks -= 1       // Decrement the count of blocks in free list
      else:
         free_check = free_check->next  // Move to the next block in free list
```

2] Three additional heap management strategies:

a. **Next Fit:** It starts the search for a free block from the block immediately after the last block where a search was conducted. If no suitable block is found in the subsequent blocks, it continues the search from the beginning of the list until a suitable block is found.

Pseudocode:

```
// Start the search from the block where the previous search ended
```

```
Set curr to next_cont

// Iterate through the list of blocks starting from next_cont
While curr is not NULL and (curr is not free or curr size is less than requested size):
   Update last to point to curr
   Move to the next block

// If no suitable block is found, start the search from the beginning of the list
If curr is NULL:
   Set curr to the beginning of the list
   While curr is not NULL and (curr is not free or curr size is less than requested size):
      Update last to point to curr
      Move to the next block
      If curr reaches next_cont:
         Break the loop to avoid infinite searching

// Return the current block (either NULL if not found or a suitable block)
Return curr
```

b. Worst Fit: It iterates through the linked list of memory blocks, searching for the block with the largest size that is still large enough to accommodate the requested size.
Pseudocode:

```
// Initialize variables to keep track of the worst-fit block and its size
Set worst_fit to NULL
Set check to 0
// Iterate through the list of blocks
While curr is not NULL:
   If curr is free, its size is greater than or equal to requested size, and curr size is larger than check:
      Update check to curr size
      Update worst_fit to point to curr
   Update last to point to curr
   Move to the
```

c. Best Fit: It iterates through the linked list of memory blocks, searching for the block with the smallest size that is still large enough to accommodate the requested size.
Pseudocode:

```
// Initialize variables to keep track of the best-fit block and its size
Set best_fit to NULL
Set check to maximum possible size
// Iterate through the list of blocks
While curr is not NULL:
   If curr is free, its size is greater than or equal to requested size, and curr size is smaller than check:
      Update check to curr size
      Update best_fit to point to curr
   Update last to point to curr
   Move to the next block in the list
// Return the block with the best fit (either NULL if not found or the best-fit block)
Return best_fit
```

3] calloc() function

The calloc() function dynamically allocates memory for a specified number of elements, initializing all bytes to zero. It first allocates memory using malloc, checks for successful

allocation, initializes the memory block to zero, updates the heap size, and returns the pointer to the allocated memory.

4] realloc() function

The realloc() function reallocates memory for an existing block pointed to by ptr with a new size. It first allocates memory with the new size using malloc, calculates the size of the existing memory block, checks if the original pointer is not NULL, copies the contents of the original memory block to the new memory block, frees the original memory block, updates the heap size, and returns the pointer to the newly allocated memory block.

Test Implementation:

As instructed, firstly we setup the libraries and then to avoid the operating system for calling the default malloc function, the code is run using $env followed by the libraries which should be tested upon using LD_PRELOAD.

In simple words the command is:

$ env LD_LOAD=lib/___(1)__ tests/__(2)___

For **(1):** *libmalloc-bf.so*(Best-Fit), *libmalloc-ff.so*(First-Fit), *libmalloc-nf.so*(Next-Fit*), libmalloc-wf.so*(Worst-Fit)

For **(2):** We will place the name from the 8 test modules given.

Test Results:

When running the command to test the working of the program, we get the following results:

Strategy	First Fit	Next Fit	Best Fit	Worst Fit
Chosen Address	0x563970958018	0x55f078e83c58	0x5568421000c4	0x563fb2cf6018
mallocs	12	12	7	7
frees	3	3	2	2
reuses	12	12	7	7
grows	10	10	6	6
splits	0	0	1	1
coalesces	0	0	0	0
blocks	9	9	6	6
requested	16048	16048	73626	73626
max heap	9064	9064	72636	72636
Time taken	0.040s	0.018s	0.017s	0.016s

Now, the program is tested on the four test cases given:

Test 1:

Allocator	Real Time (s)	User Time (s)	System Time (s)	Mallocs	Frees	Reuses	Grows	Splits	Coalesces	Blocks	Requested	Max Heap
First Fit	0.022	0.010	0.012	2	1	2	2	0	0	1	66559	66560
Next Fit	0.012	0.004	0.006	2	1	2	2	0	0	1	66559	66560
Worst Fit	0.011	0.009	0.002	2	1	2	2	0	0	1	66559	66560
Best Fit	0.014	0.000	0.014	2	1	2	2	0	0	1	66559	66560

Test 2:

Allocator	Real Time (s)	User Time (s)	System Time (s)	Mallocs	Frees	Reuses	Grows	Splits	Coalesces	Blocks	Requested	Max Heap
First Fit	0.031	0.011	0.020	1027	514	1027	1026	1	2	1024	1180670	1115136
Next Fit	0.025	0.013	0.013	1027	514	1027	1026	1	2	1024	1180670	1115136
Worst Fit	0.027	0.014	0.012	1027	514	1027	1026	1	2	1024	1180670	1115136
Best Fit	0.035	0.021	0.014	1027	514	1027	1026	1	2	1024	1180670	1115136

Test 3:

Allocator	Real Time (s)	User Time (s)	System Time (s)	Mallocs	Frees	Reuses	Grows	Splits	Coalesces	Blocks	Requested	Max Heap
First Fit	0.017	0.011	0.005	4	3	4	3	1	2	1	5472	3424
Next Fit	0.015	0.005	0.011	4	3	4	3	1	2	1	5472	3424
Worst Fit	0.012	0.003	0.007	4	3	4	3	1	2	1	5472	3424
Best Fit	0.014	0.009	0.005	4	3	4	3	1	2	1	5472	3424

Test 4:

Allocator	Real Time (s)	User Time (s)	System Time (s)	Mallocs	Frees	Reuses	Grows	Splits	Coalesces	Blocks	Requested	Max Heap
First Fit	0.015	0.015	0.000	3	2	3	2	1	1	1	4096	3072
Next Fit	0.016	0.005	0.010	3	2	3	2	1	1	1	4096	3072
Worst Fit	0.014	0.013	0.001	3	2	3	2	1	1	1	4096	3072
Best Fit	0.015	0.011	0.003	3	2	3	2	1	1	1	4096	3072

Interpretation of results:

1. **Test 1: Simple Malloc and Free**:

   All four allocation strategies (**First Fit**, **Next Fit**, **Worst Fit**, **Best Fit**) perform similarly in this test case. Real time for allocation and deallocation is relatively low across all strategies. Also, no significant difference in heap management statistics such as mallocs, frees, reuses, grows, splits, coalesces, blocks, requested memory, and max heap size.

2. **Test 2: Exercise Malloc and Free**:

All four allocation strategies again perform similarly in terms of real time for allocation and deallocation. Heap management statistics are also similar across all strategies, indicating balanced performance.

3. **Test 3: Test Coalesce**:

   Similar performance across all allocation strategies with low real time for allocation and deallocation. Heap management statistics are consistent across all strategies.

4. **Test 4: Test Block Split and Reuse**:

   All strategies exhibit similar performance with relatively low real time for allocation and deallocation. Heap management statistics remain consistent across all strategies.

Conclusion:

The Malloc Assignment involved the implementation and testing of a custom memory allocator library capable of managing heap memory using different allocation strategies. The implementation followed a clear algorithmic approach for each function, ensuring efficient memory management and minimizing fragmentation. The free function was designed to mark memory blocks as free and merge adjacent free blocks to prevent fragmentation. The additional heap management strategies provided options for optimizing memory allocation based on different criteria such as block size.

During testing, the custom memory allocator library was benchmarked against the standard system call malloc() across various test cases. The results showed consistent performance across different allocation strategies, with minimal variations in real-time for allocation and deallocation. Heap management statistics remained balanced, indicating robust performance and efficient memory utilization across all test scenarios. The custom memory allocator demonstrated reliable performance and memory management capabilities comparable to the standard malloc() function. The assignment provided valuable insights into memory allocation strategies and their impact on performance, highlighting the importance of efficient memory management in software development.