Mallopt() – Set Memory Allocation Parameters

Mallopt() function present in **malloc.h** "adjusts parameters that control the behavior of the memory-allocation functions".

Arena: An Arena represents a pool of memory that can be used by malloc calls to service allocation requests. Arenas are thread safe and hence may have multiple concurrent memory requests.

Syntax:

int mallopt(int param, int value);

For the **param**:

M_ARENA_MAX – If this has a nonzero value, it defines a hard limit on the maximum number of arenas that can be created.

The default value is 0, indicating that the limit on the number of arenas is determined according to the setting of ***M_ARENA_TEST***.

Demonstration:

Malloc without modifying M_ARENA_MAX:

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void* allocate_memory(void* arg) {
    for (int i = 0; i < 1000; i++) {
        char* data = (char*) malloc(1024);
        if (data) {
            free(data);
        }
    }
    printf("Thread %ld finished memory allocation using default M_ARENA_MAX.\n", pthread_self());
    return NULL;
}

int main() {
    printf("Starting program without modifying M_ARENA_MAX.\n");

    const int NUM_THREADS = 10;
    pthread_t threads[NUM_THREADS];

    // Launch threads
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, allocate_memory, NULL);
    }

    // Wait for threads to finish
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("Finished program without modifying M_ARENA_MAX.\n");
    return 0;
}

```

Output:

```
Starting program without modifying M_ARENA_MAX.
Thread 140598592341568 finished memory allocation using default M_ARENA_MAX.
Thread 140598583887424 finished memory allocation using default M_ARENA_MAX.
Thread 140598505834048 finished memory allocation using default M_ARENA_MAX.
Thread 140598497379904 finished memory allocation using default M_ARENA_MAX.
Thread 140598488925760 finished memory allocation using default M_ARENA_MAX.
Thread 140598472017472 finished memory allocation using default M_ARENA_MAX.
Thread 140598480471616 finished memory allocation using default M_ARENA_MAX.
Thread 140598463563328 finished memory allocation using default M_ARENA_MAX.
Thread 140598455109184 finished memory allocation using default M_ARENA_MAX.
Thread 140598103180864 finished memory allocation using default M_ARENA_MAX.
Finished program without modifying M_ARENA_MAX.
```

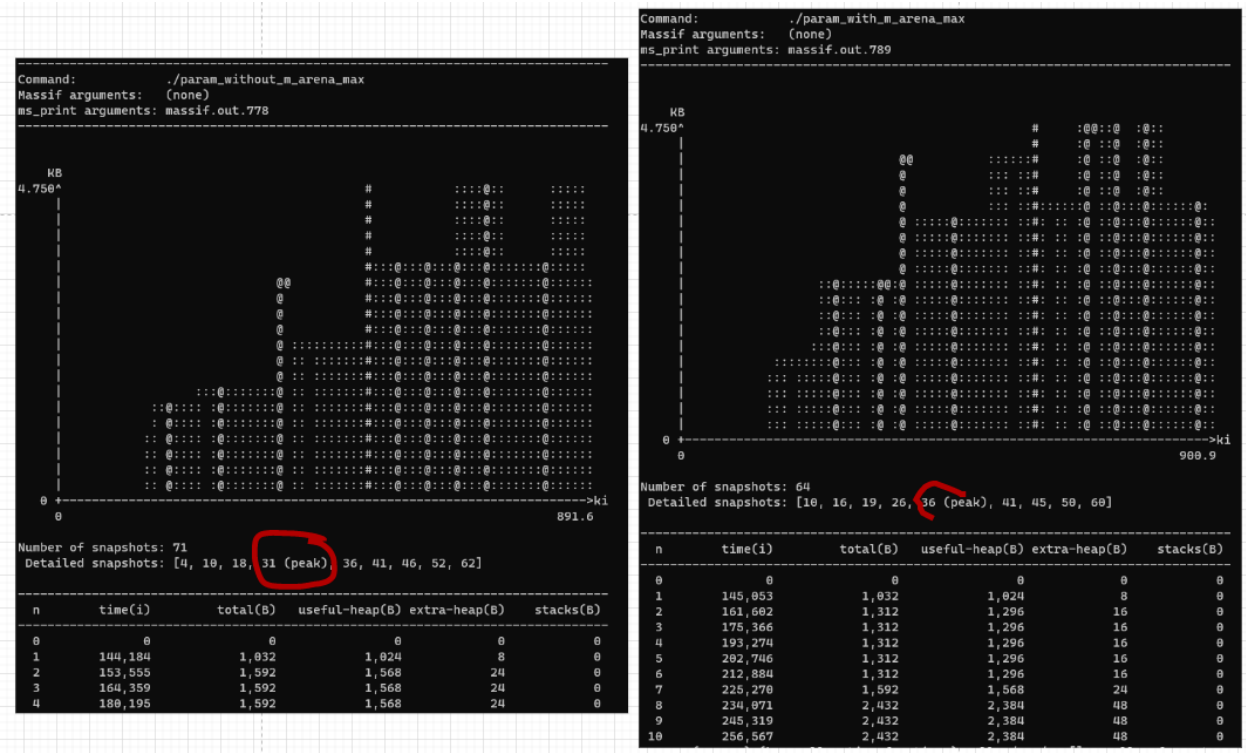
Malloc with modifying M_ARENA_MAX:

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <malloc.h>

void* allocate_memory(void* arg) {
    for (int i = 0; i < 1000; i++) {
        char* data = (char*) malloc(1024);
        if (data) {
            free(data);
        }
    }
    printf("Thread %ld finished memory allocation with M_ARENA_MAX set to 2.\n", pthread_self());
    return NULL;
}

int main() {
    printf("Starting program with M_ARENA_MAX set to 2.\n");

    // Set the maximum number of arenas to 2
    mallopt(M_ARENA_MAX, 2);

    const int NUM_THREADS = 10;
    pthread_t threads[NUM_THREADS];

    // Launch threads
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, allocate_memory, NULL);
    }

    // Wait for threads to finish
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("Finished program with M_ARENA_MAX set to 2.\n");
    return 0;
}
```

Output:

```
~ ./param_with_m_arena_max
Starting program with M_ARENA_MAX set to 2.
Thread 140304380659264 finished memory allocation with M_ARENA_MAX set to 2.
Thread 140304372205120 finished memory allocation with M_ARENA_MAX set to 2.
Thread 140304363750976 finished memory allocation with M_ARENA_MAX set to 2.
Thread 140304355296832 finished memory allocation with M_ARENA_MAX set to 2.
Thread 140304346842688 finished memory allocation with M_ARENA_MAX set to 2.
Thread 140304338388544 finished memory allocation with M_ARENA_MAX set to 2.
Thread 140304329934400 finished memory allocation with M_ARENA_MAX set to 2.
Thread 140304321480256 finished memory allocation with M_ARENA_MAX set to 2.
Thread 140304313026112 finished memory allocation with M_ARENA_MAX set to 2.
Thread 140304233465408 finished memory allocation with M_ARENA_MAX set to 2.
```

By using **valdgrind**, we can see a detailed comparison of the above



```
Command:            ./param_without_m_arena_max
Massif arguments:   (none)
ms_print arguments: massif.out.778
--------------------------------------------------------------------------------
    KB
4.750^                                    #          ::::@::      :::::
     |                                    #          ::::@::      :::::
     |                                    #          ::::@::      :::::
     |                                    #          ::::@::      :::::
     |                                    #          ::::@::      :::::
     |                                    #:::@:::@:::@:::@::::::@:::::
     |                          @@        #:::@:::@:::@:::@::::::@:::::
     |                          @         #:::@:::@:::@:::@::::::@:::::
     |                          @         #:::@:::@:::@:::@::::::@:::::
     |                          @         #:::@:::@:::@:::@::::::@:::::
     |                          @ ::::::::::#:::@:::@:::@:::@::::::@:::::
     |                          @ :: ::::::#:::@:::@:::@:::@::::::@:::::
     |                          @ :: ::::::#:::@:::@:::@:::@::::::@:::::
     |                    :::@::::::@ :: ::::::#:::@:::@:::@:::@::::::@:::::
     |              ::@::: :@::::::@ :: ::::::#:::@:::@:::@:::@::::::@:::::
     |              : @::: :@::::::@ :: ::::::#:::@:::@:::@:::@::::::@:::::
     |             :: @::: :@::::::@ :: ::::::#:::@:::@:::@:::@::::::@:::::
     |             :: @::: :@::::::@ :: ::::::#:::@:::@:::@:::@::::::@:::::
     |             :: @::: :@::::::@ :: ::::::#:::@:::@:::@:::@::::::@:::::
   0 +-------------------------------------------------------------------->ki
     0                                                                  891.6

Number of snapshots: 71
 Detailed snapshots: [4, 10, 18, 31 (peak), 36, 41, 46, 52, 62]

--------------------------------------------------------------------------------
  n        time(i)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  0              0                0              0             0            0
  1        144,184            1,032          1,024             8            0
  2        153,555            1,592          1,568            24            0
  3        164,359            1,592          1,568            24            0
  4        180,195            1,592          1,568            24            0
```

```
Command:            ./param_with_m_arena_max
Massif arguments:   (none)
ms_print arguments: massif.out.789
--------------------------------------------------------------------------------
    KB
4.750^                                                    #     :@@::@  :@::
     |                                                    #     :@ ::@  :@::
     |                                @@          ::::::#     :@ ::@  :@::
     |                                @           ::: ::#     :@ ::@  :@::
     |                                @           ::: ::#     :@ ::@  :@::
     |                                @           ::: ::#::::::@ ::@:::@::::::@:
     |                                @ :::::@:::::::: ::#: :: :@ ::@:::@::::::@::
     |                                @ :::::@:::::::: ::#: :: :@ ::@:::@::::::@::
     |                                @ :::::@:::::::: ::#: :: :@ ::@:::@::::::@::
     |                                @ :::::@:::::::: ::#: :: :@ ::@:::@::::::@::
     |                      ::@::::::@@:@ :::::@:::::::: ::#: :: :@ ::@:::@::::::@::
     |                      ::@::: :@ :@ :::::@:::::::: ::#: :: :@ ::@:::@::::::@::
     |                      ::@::: :@ :@ :::::@:::::::: ::#: :: :@ ::@:::@::::::@::
     |                      ::@::: :@ :@ :::::@:::::::: ::#: :: :@ ::@:::@::::::@::
     |                    :::@::::: :@ :@ :::::@:::::::: ::#: :: :@ ::@:::@::::::@::
     |                :::::::::@::: :@ :@ :::::@:::::::: ::#: :: :@ ::@:::@::::::@::
     |              ::: :::::@::: :@ :@ :::::@:::::::: ::#: :: :@ ::@:::@::::::@::
     |              ::: :::::@::: :@ :@ :::::@:::::::: ::#: :: :@ ::@:::@::::::@::
     |              ::: :::::@::: :@ :@ :::::@:::::::: ::#: :: :@ ::@:::@::::::@::
   0 +-------------------------------------------------------------------->ki
     0                                                                  900.9

Number of snapshots: 64
 Detailed snapshots: [10, 16, 19, 26, 36 (peak), 41, 45, 50, 60]

--------------------------------------------------------------------------------
  n        time(i)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  0              0                0              0             0            0
  1        145,053            1,032          1,024             8            0
  2        161,602            1,312          1,296            16            0
  3        175,366            1,312          1,296            16            0
  4        193,274            1,312          1,296            16            0
  5        202,746            1,312          1,296            16            0
  6        212,884            1,312          1,296            16            0
  7        225,270            1,592          1,568            24            0
  8        234,071            2,432          2,384            48            0
  9        245,319            2,432          2,384            48            0
 10        256,567            2,432          2,384            48            0
```

- **Peak Memory**: Both runs have a similar peak memory usage of around 4.750 KB. This suggests that the total amount of memory requested by the program was similar in both scenarios.
- **Allocation Patterns**: The memory allocation seems more staggered in the default case compared to the one with **M_ARENA_MAX** set to 2. This might be because, by default, the number of arenas is typically tied to the number of available CPU cores. Each thread might be getting its own arena, leading to a different allocation pattern.

- **Allocator Overhead**: The overhead (**extra-heap**) added by the memory allocator is small in both cases and doesn't show a significant difference between the two runs.

While the overall memory usage is similar between the two runs, the allocation patterns differ slightly. This is in line with what we would expect: M_ARENA_MAX doesn't necessarily change how much memory is used, but it can change how that memory is allocated among threads. The real impact would be more evident in more complex, real-world applications with varying allocation and deallocation patterns across multiple threads.

Most of these options are to check the efficiency of the program and individually testing them for small code wouldn't show much difference. At the end of this article we will be seeing the overall difference.

M_ARENA_TEST – It determines when the system should calculate a hard limit on the number of arenas(memory pools for allocations). When the number of created arenas reaches the value of ***M_ARENA_TEST***, the system decides on a hard limit for total number of arenas. After this hard limit is set, no more than that number of arenas can be created. The default value is sizeof(long) i.e. 2 (on 32-bit systems.

M_CHECK_ACTION- Controls how glibc responds to certain programming errors related to memory allocation, like double freeing a pointer. The 3 least significant bots (2,1,0) of the values assigned to this parameter determine the glibc behavior.

- **Bit 0** – If set, print a detailed message about the error on stderr. The message includes details like the program name, the memory-allocation function where the error occurred , a description of the error and the memory address of the error.
- **Bit 1**- If set, the program is terminated using abort(3) after any message from Bit 0 is printed. If Bit 0 is also set, a stack trace is printed and the processes memory mapping is displayed.
- **Bit 2**- Only relevant if Bit 0 is set. Simplifies the error message to only show the function where the error occurred and a brief description.

The following numeric values are meaningful for **M_CHECK_ACTION**.

- **0**: Ignore errors and continue.
- **1**: Print a detailed error message and continue.
- **2**: Terminate the program.
- **3**: Print a detailed error message, stack trace, memory mappings, and then terminate.
- **5**: Print a simplified error message and continue.
- **7**: Print a simplified error message, stack trace, memory mappings, and then terminate.

M_MMAP_MAX – Sets the maximum number of allocation requests that can be serviced using mmap() simultaneously. Some systems have a limited number of internal tables for mmap() and overusing them can degrade performance. By setting a limit on the number of **mmap()** calls, it helps

in ensuring performance doesn't degrade due to overuse of these tables. The default value is set to **65,536**.

M_MMAP_THRESHOLD – Determines the size threshold(in bytes) for which memory allocations will use **mmap()** instead of **sbrk()** when the request can't be satisfied from the free list. The default value is set to **128*1024** bytes. Its maximum is **512*1024** bytes on a ***32-bit system*** and **4*1024*1024*sizeof(long)** on a ***64-bit system***. Why exactly these values?

On a **32-bit system**, the addressable memory space is limited. Hence, it makes sense to have a smaller threshold to avoid over-reliance on mmap which might quickly-fragment this limited address space. 512 KB is reasonable size that's large enough to cover many typical application's memory allocation needs without needing to invoke **mmap()** to frequently, which is small enough to avoid excessive fragmentation.

On a **64-bit system**, they can address much larger memory space(Possibly 18.4 Million TB). This means, they can handle larger memory mappings via mmap without significant fragmentation concerns. The value presented above is a large threshold reflecting the larger addressable memory and the capability to handle bigger mamp() regions without major downsides.

In simple terms, **M_MMAP_THRESHOLD** defines the byte-size threshold at which memory allocations will prefer **mmap()** over **sbrk()**.

M_MXFAST – It sets the maximum size(in bytes) for memory allocation requests that will be serviced using '**fastbins'**.

Fastbins: These are a set of bins that store deallocated memory blocks of the same size without merging them. The advantage is that subsequent allocations of the same size can be done very quickly by reusing these blocks, but might lead to memory fragmentation.

The default value is 64*sizeof(size_t)/4 i.e ***64*** on a 32-bit architecture.

M_PERTURB – It is used for intentionally initializing allocated memory with non-zero values and setting specific values in memory when it's released. The goal is to help detect programming mistakes related to memory usage.

The default value is 0, indicating the behavior is off.

M_TOP_PAD – Determines the padding used when adjusting the program break via sbrk() system call. It's mechanism to avoid frequent adjustments by allocating a bit more than immediately required.

- **Icreasing the Program Break**: When the program break is expanded, **M_TOP_PAD** bytes are added to the **sbrk(2)** request. This means that instead of just increasing the heap by the exact amount needed, it's increased by that amount plus the padding defined by **M_TOP_PAD**.
- **Trimming the Heap**: When the heap is trimmed due to a call to **free(3)** (related to the **M_TRIM_THRESHOLD** parameter), **M_TOP_PAD** bytes of free space are preserved at the top of the heap. This ensures there's some buffer space available for future allocations without needing immediate adjustments.

- **Page Alignment**: The padding amount is always aligned to a system page boundary, ensuring that memory is allocated and managed in units that align with the system's memory management.

The default value is set to **128*1024 bytes (128KB)**. This is a middle ground that provides a balance between performance and memory usage for typical applications.

M_TRIM_THRESHOLD - Determines the threshold (in bytes) at which a contiguous block of free memory at the top of the heap will be returned to the system.

- If a block of free memory at the heap's top grows beyond this threshold, the **free(3)** function will use the **sbrk(2)** system call to trim the heap and release this excess memory back to the system.
- This is particularly beneficial in long-running programs that might allocate large amounts of memory temporarily and then free it. Without this mechanism, the program's memory usage might remain unnecessarily high.

The default threshold is set to **128*1024 bytes**.

Demonstration:

Plain_malloc.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <malloc.h>

void print_mallinfo() {
    struct mallinfo info = mallinfo();
    printf("Total non-mmapped bytes (arena): %d\n", info.arena);
    printf("Number of free chunks (ordblks): %d\n", info.ordblks);
    printf("Total allocated space (uordblks): %d\n", info.uordblks);
    printf("Total free space (fordblks): %d\n", info.fordblks);
    printf("\n");
}

void* allocate_memory(void* arg) {
    for (int i = 0; i < 1000; i++) {
        char* data = (char*) malloc(1024);
        if (data) {
            free(data);
        }
    }
    printf("Thread %ld finished memory allocation without modifying malloc parameters.\n", pthread_self());
    return NULL;
}

int main() {
    printf("Starting program without modifying malloc parameters.\n\n");
    print_mallinfo();

    const int NUM_THREADS = 10;
    pthread_t threads[NUM_THREADS];

    // Launch threads
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, allocate_memory, NULL);
    }

    // Wait for threads to finish
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    print_mallinfo();
    printf("Finished program without modifying malloc parameters.\n");
    return 0;
}

```

Modified_malloc.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <malloc.h>

void print_mallinfo() {
    struct mallinfo info = mallinfo();
    printf("Total non-mmapped bytes (arena): %d\n", info.arena);
    printf("Number of free chunks (ordblks): %d\n", info.ordblks);
    printf("Total allocated space (uordblks): %d\n", info.uordblks);
    printf("Total free space (fordblks): %d\n", info.fordblks);
    printf("\n");
}

void* allocate_memory(void* arg) {
    for (int i = 0; i < 1000; i++) {
        char* data = (char*) malloc(1024);
        if (data) {
            free(data);
        }
    }
    printf("Thread %ld finished memory allocation with modified malloc parameters.\n", pthread_self());
    return NULL;
}

int main() {
    printf("Starting program with modified malloc parameters.\n\n");
    print_mallinfo();

    // Modify various malloc parameters using mallopt
    mallopt(M_ARENA_MAX, 2);
    mallopt(M_ARENA_TEST, 2);
    mallopt(M_CHECK_ACTION, 3);
    mallopt(M_MMAP_MAX, 2);
    mallopt(M_MMAP_THRESHOLD, 128*1024);
    mallopt(M_MXFAST, 64*sizeof(size_t)/4);
    mallopt(M_PERTURB, 0xff);
    mallopt(M_TOP_PAD, 128*1024);
    mallopt(M_TRIM_THRESHOLD, 128*1024);

    const int NUM_THREADS = 10;
    pthread_t threads[NUM_THREADS];

    // Launch threads
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, allocate_memory, NULL);
    }

    // Wait for threads to finish
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    print_mallinfo();
    printf("Finished program with modified malloc parameters.\n");
    return 0;
}

```

Output:

```
~ ./plain_malloc
Starting program without modifying malloc parameters.

Total non-mmapped bytes (arena): 135168
Number of free chunks (ordblks): 1
Total allocated space (uordblks): 1696
Total free space (fordblks): 133472

Thread 139662449968704 finished memory allocation without modifying malloc parameters.
Thread 139662441514560 finished memory allocation without modifying malloc parameters.
Thread 139662433060416 finished memory allocation without modifying malloc parameters.
Thread 139662424606272 finished memory allocation without modifying malloc parameters.
Thread 139662416152128 finished memory allocation without modifying malloc parameters.
Thread 139662202963520 finished memory allocation without modifying malloc parameters.
Thread 139662194509376 finished memory allocation without modifying malloc parameters.
Thread 139662186055232 finished memory allocation without modifying malloc parameters.
Thread 139662177601088 finished memory allocation without modifying malloc parameters.
Thread 139662169146944 finished memory allocation without modifying malloc parameters.
Total non-mmapped bytes (arena): 1081344
Number of free chunks (ordblks): 8
Total allocated space (uordblks): 20368
Total free space (fordblks): 1060976

Finished program without modifying malloc parameters.
~ ./modified_malloc
Starting program with modified malloc parameters.

Total non-mmapped bytes (arena): 135168
Number of free chunks (ordblks): 1
Total allocated space (uordblks): 1696
Total free space (fordblks): 133472

Thread 140623520204352 finished memory allocation with modified malloc parameters.
Thread 140623511750208 finished memory allocation with modified malloc parameters.
Thread 140623503296064 finished memory allocation with modified malloc parameters.
Thread 140623494841920 finished memory allocation with modified malloc parameters.
Thread 140623486387776 finished memory allocation with modified malloc parameters.
Thread 140623403222592 finished memory allocation with modified malloc parameters.
Thread 140623394768448 finished memory allocation with modified malloc parameters.
Thread 140623386314304 finished memory allocation with modified malloc parameters.
Thread 140623377860160 finished memory allocation with modified malloc parameters.
Thread 140623369406016 finished memory allocation with modified malloc parameters.
Total non-mmapped bytes (arena): 270336
Number of free chunks (ordblks): 2
Total allocated space (uordblks): 6832
Total free space (fordblks): 263504

Finished program with modified malloc parameters.
~ vi plain_malloc.c
```

Without Modifying malloc parameters:

1. **Before Memory Allocation:**
    - Total non-mmapped bytes (**arena**): 135,168 bytes
    - Number of free chunks (**ordblks**): 1
    - Total allocated space (**uordblks**): 1,696 bytes
    - Total free space (**fordblks**): 133,472 bytes
2. **After Memory Allocation:**
    - Total non-mmapped bytes (**arena**): 1,081,344 bytes
    - Number of free chunks (**ordblks**): 8
    - Total allocated space (**uordblks**): 20,368 bytes
    - Total free space (**fordblks**): 1,060,976 bytes

With Modified malloc parameters:

1. **Before Memory Allocation:**
    - Total non-mmapped bytes (**arena**): 135,168 bytes
    - Number of free chunks (**ordblks**): 1
    - Total allocated space (**uordblks**): 1,696 bytes
    - Total free space (**fordblks**): 133,472 bytes
2. **After Memory Allocation:**
    - Total non-mmapped bytes (**arena**): 270,336 bytes
    - Number of free chunks (**ordblks**): 2
    - Total allocated space (**uordblks**): 6,832 bytes
    - Total free space (**fordblks**): 263,504 bytes

Observations:

1. **Starting Conditions**: Both programs began with the same memory conditions, which is expected since both programs start off the same way.
2. **Memory Allocation Behavior**:
    - In the version without modified malloc parameters, the **arena** size has increased significantly after memory operations (1,081,344 bytes). This suggests that more memory arenas were created to handle the threaded malloc operations.

- In the version with modified malloc parameters, the **arena** size is lower (270,336 bytes). This indicates that limiting the number of memory arenas (using **M_ARENA_MAX** set to 2) constrained the memory allocation behavior.

3. **Memory Usage**:
    - Without modifying the parameters, more memory was allocated in total (20,368 bytes) as compared to when the parameters were modified (6,832 bytes).
    - The free space (**fordblks**) was also significantly more in the non-modified version.

The **mallopt** configurations have a clear effect on the memory allocation behavior in a multithreaded environment. Limiting the number of arenas results in more conservative memory usage and fewer memory arenas being created. This can be beneficial for applications that want to constrain memory usage in multithreaded scenarios. On the flip side, limiting the number of arenas too much might cause contention among threads, potentially leading to reduced performance in highly concurrent scenarios. As always, the optimal settings would depend on the specific use case and workload.

Follow the tag #memorywithyash to get updates regarding upcoming articles on Understanding memory in depth in C.

~~~~~~
Do follow me to receive updates regarding Embedded Systems, Space and Technology concepts.
~~~~~~

Happy learning.
Learn together, Grow together.

#earlycareer #embeddedsystems #learning #embeddedc #learnwithyash #linuxfun #memoryallocators #mallopt