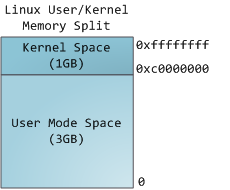
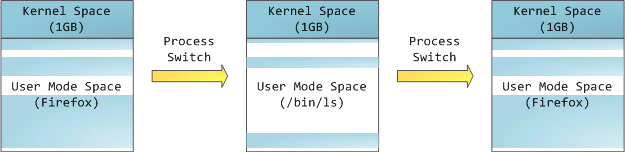
**Task 1 - Understanding the virtual memory address space and basic dynamic space allocation**

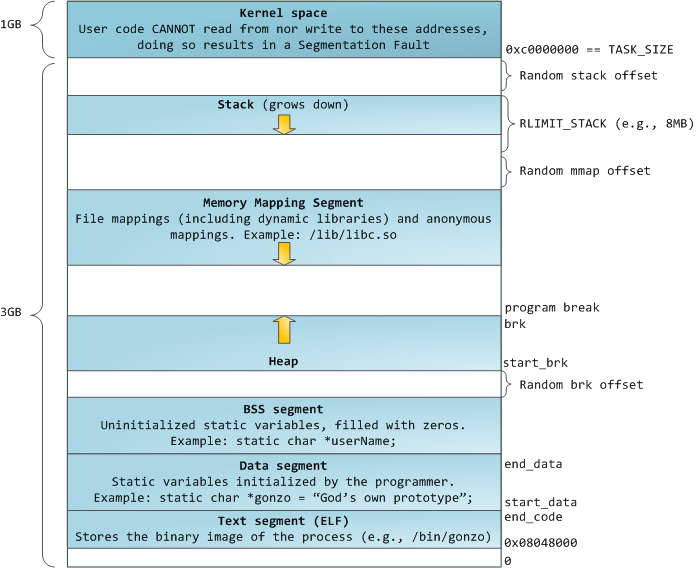
For additional details please refer to [1]. On 32 bit machines the size of a process virtual address space in Linux is 4Gb. This space is split 1 Gb for kernel space and 3 Gb for user mode. parameters of the driver deserve a short explanation. 512 is the size of chunks allocated



This does **not** mean the kernel uses that much physical memory, only that it has that portion of address space available to map whatever physical memory it wishes. Kernel space is flagged in the page tables as exclusive to privileged mode, hence a page fault is triggered if user-mode programs try to touch it. In Linux, kernel space is constantly present and maps the same physical memory in all processes. Kernel code and data are always addressable, ready to handle interrupts or system calls at any time. By contrast, the mapping for the user-mode portion of the address space changes whenever a process switch happens:



Blue regions represent virtual addresses that are mapped to physical memory, whereas white regions are unmapped. In the example above, Firefox has used far more of its virtual address space due to its legendary memory hunger. The distinct bands in the address space correspond to **memory segments** like the heap, stack, and so on. Here is the standard segment layout in a Linux process:

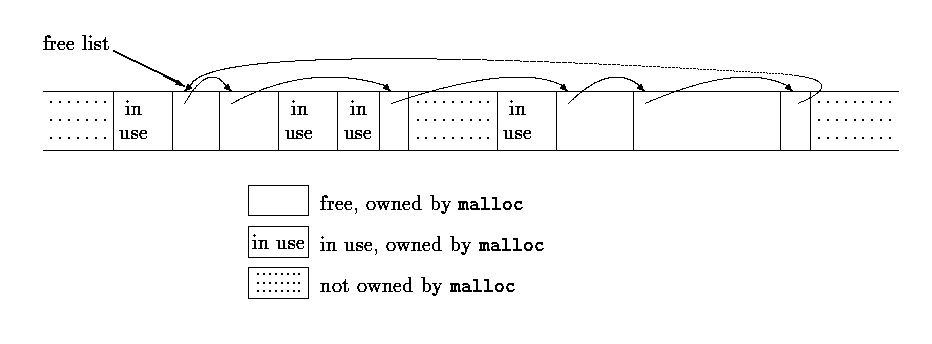


The topmost segment in the process address space is the stack, which stores local variables and function parameters in most programming languages. Calling a method or function pushes a new **stack frame** onto the stack. The stack frame is destroyed when the function returns. This simple design, possible because the data obeys strict [LIFO](http://en.wikipedia.org/wiki/Lifo) order, means that no complex data structure is needed to track stack contents – a simple pointer to the top of the stack will do. Pushing and popping are thus very fast and deterministic. Also, the constant reuse of stack regions tends to keep active stack memory in the [cpu caches](http://duartes.org/gustavo/blog/post/intel-cpu-caches), speeding up access. Each thread in a process gets its own stack.

Below the stack, we have the memory mapping segment. Here the kernel maps contents of files directly to memory. Any application can ask for such a mapping via the Linux [mmap()](http://man7.org/linux/man-pages/man2/mmap.2.html) system call. Memory mapping is a convenient and high-performance way to do file I/O, so it is used for loading dynamic libraries. It is also possible to create an **anonymous memory mapping** that does not correspond to any files, being used instead for program data. In Linux, if you request a large block of memory via [malloc()](.html), the C library will create such an anonymous mapping instead of using heap memory. ‘Large’ means larger than MMAP\_THRESHOLD bytes, 128 kB by default and adjustable via [mallopt()](.html).

Speaking of the heap, it comes next in our plunge into address space. The heap provides runtime memory allocation, like the stack. Bur heap is meant for data that must outlive the function doing the allocation, unlike the stack. Most languages provide heap management to programs. Satisfying memory requests is a joint affair between the language runtime and the kernel. In C, the interface to heap allocation is [malloc()](http://man7.org/linux/man-pages/man3/malloc.3.html) and friends.

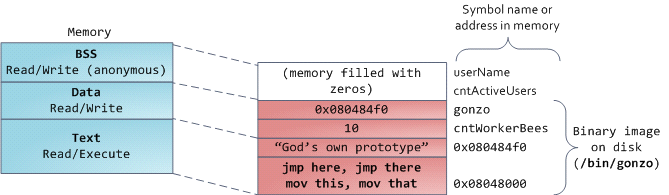
If there is enough space in the heap to satisfy a memory request, it can be handled by the language runtime without kernel involvement. Otherwise the heap is enlarged via the [brk()](.html) system call to make room for the requested block. Simple heap management is explained in chapter 8.7 of [2]. Read chapter 8.7 to see a very simplistic implementation of the heap management that is a linked list of free and used chunks of memory (aka heap segment).



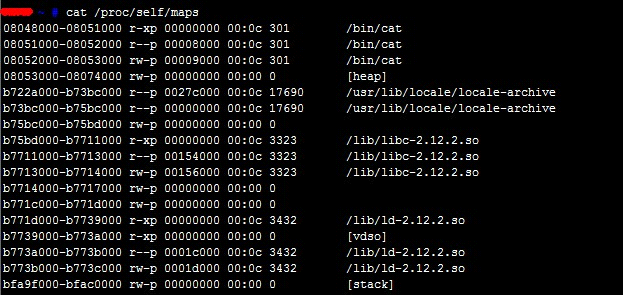
Finally, we get to the lowest segments of memory: BSS, data, and program text. Both BSS and data store contents for static (global) variables in C. The difference is that BSS stores the contents of *uninitialized* static variables, whose values are not set by the programmer in source code. The BSS memory area is anonymous: it does not map any file. If you say static int cntActiveUsers, the contents of cntActiveUsers live in the BSS.

The data segment, on the other hand, holds the contents for static variables initialized in source code. This memory area **is not anonymous**. It maps the part of the program’s binary image that contains the initial static values given in source code. So if you say static int cntWorkerBees = 10, the contents of cntWorkerBees live in the data segment and start out as 10. Even though the data segment maps a file, it is a **private memory mapping**, which means that updates to memory are not reflected in the underlying file. This must be the case, otherwise assignments to global variables would change your on-disk binary image. !

The data example in the diagram is trickier because it uses a pointer. In that case, the *contents* of pointer gonzo – a 4-byte memory address – live in the data segment. The actual string it points to does not, however. The string lives in the **text** segment, which is read-only and stores all of your code in addition to tidbits like string literals. The text segment also maps your binary file in memory, but writes to this area earn your program a Segmentation Fault. This helps prevent pointer bugs, though not as effectively as avoiding C in the first place. Here’s a diagram showing these segments and our example variables:



You can examine the memory areas in a Linux process by reading the file /proc/pid\_of\_process/maps. For example, run "cat /proc/self/maps" to see the memory map of cat when is runs on the system.



We can see from this figure that segment may contain many areas. For example, each memory mapped file normally has its own area in the mmap segment, and dynamic libraries have extra areas similar to BSS and data. As you can see the picture is quite complex. Also, sometimes the vocabulary might differ in several ways. For example, people may say “data segment” meaning all of data + bss + heap.

**Task 2 - how we are going to to implement dynamic memory allocation**

The implementation of memory management depends greatly upon operating system and architecture and even for the same operating system and architecture several memory allocators may exist [3]. In our assignment we will implement a simple version of a Hoard memory allocator [4].

Our version of the allocator will have P + 1 heaps. **According to Hoard, heap is a "mmap"-ed area in the virtual address space.** P is the number of processors of the system. The idea of having P heaps is to allow dynamic memory allocations management for each processor on a dedicated heap. The idea of having one more heap (which is also called a global heap) is to allow underutilized chunks of memory from per-processor heaps to be moved from these per-processor heaps to the global one.

Each heap “owns” a number of so called superblocks or in putting differently, each heap (both global and per-processor) is a collection of superblocks. When there is no memory available in any superblock on a thread’s heap, the allocator obtains a superblock from the global heap if one is available. If the global heap is also empty, Hoard creates a new superblock by requesting virtual memory from the operating system and adds it to the thread’s heap. Hoard does not return empty superblocks to the operating system. It instead makes these superblocks available for reuse.

Each superblock is an array of some number of blocks (objects) and contains a free list of its available blocks maintained in LIFO order to improve locality. All superblocks are the same size (S). The superblock size S is defined in *mtmm.h* as SUPERBLOCK\_SIZE. Objects larger than half the size of a superblock are managed directly using the virtual memory system (i.e., they are allocated via mmap and freed using munmap). All of the blocks in a superblock are in the same size class. By using size classes that are a power of b apart (where b is greater than 1) and rounding the requested size up to the nearest size class, Hoard bounds worst-case internal fragmentation within a block to a factor of b. In order to reduce external fragmentation, Hoard recycles completely empty superblocks for re-use by any size class. For clarity of exposition, we assume a single size class in the discussion below.

Memory allocator will maintain *usage statistics* for each heap . These statistics are:

*u(i)* - the amount of memory in use (“live”) in heap *i*

and

*a(i)*  - the amount of memory held in heap *i* that was allocated by the memory allocator from the operating system

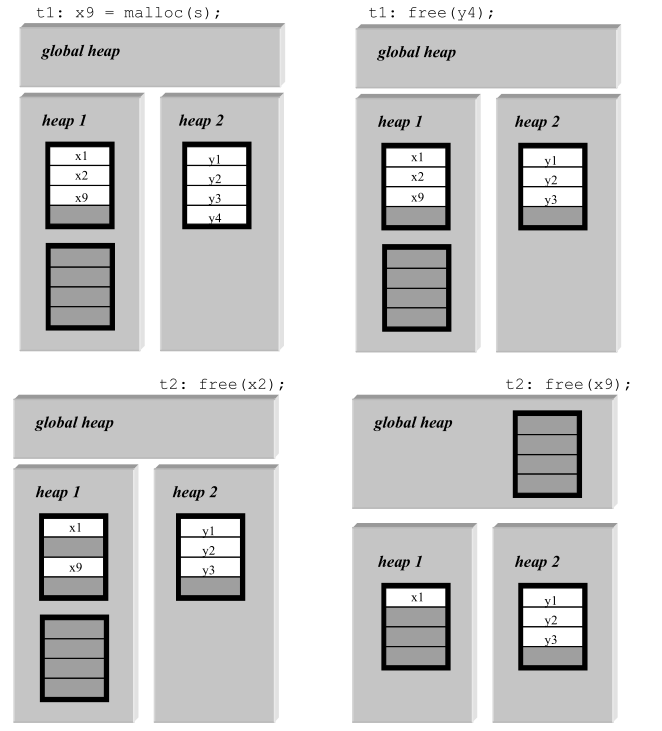
Hoard moves superblocks from a per-processor heap to the global heap when the per-processor heap crosses the emptiness threshold: more than ***f***, the empty fraction, of its blocks are not in use ( u(i) < (1 ***− f)a(i) ), and there are more than some number K of superblocks’ worth of free memory on the heap ( u(i) < a(i) − K ∗ S ).***

As long as a heap is not more than f empty, and has K or fewer superblocks, Hoard will not move superblocks from a per-processor heap to the global heap. Whenever a per-processor heap does cross the emptiness threshold, Hoard transfers one of its superblocks that is at least f empty to the global heap. Always removing such a superblock whenever we cross the emptiness threshold maintains the following invariant on the per-processor heaps: ( u(i) ≥ a(i) − K ∗ S) ∨ (u(i) ≥ (1 − f)a(i) ). When we remove a superblock, we reduce u(i) by at most (1 − f)S but reduce a(i) by S, thus restoring the invariant.

Hoard ﬁnds f-empty superblocks in constant time by dividing superblocks into a number of bins that are calld “fullness groups”. Each bin contains a doubly-linked list of superblocks that are in a given fullness range (e.g., all superblocks that are between 3/4 and completely empty are in the same bin). Hoard moves superblocks from one group to another when appropriate, and always allocates from nearly-full superblocks.

**In our assignment, we advise to implement fullness ranges as following: all superblocks that are between 3/4 and completely empty are in the same bin. Others are in another bin. But, we will also accept the solutions that all superblocks from the same size class are in the same doubly-connected list.**

Let's have a look at the example of how the allocator manages superblocks? For simplicity, we assume there are two threads and heaps (thread i maps to heap i). In this example (which reads from top left to top right, then bottom left to bottom right), the empty fraction f is 1/4 and K is 0. Thread 1 executes code written on the left-hand side of each diagram (preﬁxed by “t1:”) and thread 2 executes code on the right-hand side (preﬁxed by “t2:”). Initially, the global heap is empty, heap 1 has two superblocks (one partially full, one empty), and heap 2 has a completely-full superblock. The top left diagram shows the heaps after thread 1 allocates x9 from heap 1. Hoard selects the fullest superblock in heap 1 for allocation. Next, in the top right diagram, thread 1 frees y4, which is in a superblock that heap 2 owns. Because heap 2 is still more than 1/4 full, Hoard does not remove a superblock from it. In the bottom left diagram, thread 2 frees x2, which is in a superblock owned by heap 1. This free does not cause heap 1 to cross the emptiness threshold, but the next free (of x9) does. Hoard then moves the completely-free superblock from heap 1 to the global heap.



Finally let's present the pseudocode of the malloc and free

|  |
| --- |
|  |
|  |

Explanation of the malloc

Hoard directly allocates “large” objects (size > S/2) via the virtual memory system. When a thread on processor i calls malloc for small objects, Hoard locks heap i and gets a block of a superblock with free space, if there is one on that heap (line 4). If there is not, Hoard checks the global heap (heap 0) for a superblock. If there is one, Hoard transfers it to heap i, adding the number of bytes in use in the superblock s.u to u(i), and the total number of bytes in the superblock S to a(i) (lines 10–14). If there are no superblocks in either heap i or heap 0, Hoard allocates a new superblock and inserts it into heap i (line 8). Hoard then chooses a single block from a superblock with free space, marks it as allocated, and returns a pointer to that block.

Explanation of the free

Each superblock has an “owner” (the processor whose heap it’s in). When a processor frees a block, Hoard ﬁnds its superblock (through a pointer in the block’s header). (If this block is “large”, Hoard immediately frees the superblock to the operating system.) It ﬁrst locks the superblock and then locks the owner’s heap. Hoard then returns the block to the superblock and decrements u i. If the heap is too empty (u(i) < a(i) − K ∗ S and u(i) < (1 − f)a(i)), Hoard transfers a superblock that is at least f empty to the global heap (lines 10-12). Finally, Hoard unlocks heap i and the superblock.

**Task 3 – Run the school solution and compare with regular malloc**

We have provided a *libmtmm.a* shared library that re-implements the regular malloc, free and realloc supplied by glibc and defined in stdlib.h. We also have proved a driver called *linux-scalability.c* that creates multiple threads each allocating/freeing memory chunks. Makefile that we have supplied compiles to *linux-scalability* executable file.

You can choose one of the following options:

1. uncomment the line *"MYLIBS = libmtmm.a"* in our Makefile to link to *linux-scalability* with hoard memory allocator (and comment other " *MYLIBS =XYZ"* lines)
2. uncomment the line *"MYLIBS = libSimpleMTMM.a"* in our Makefile to link to *linux-scalability* with a simple memory allocator we have supplied (and comment other " *MYLIBS =XYZ"* lines)
3. uncomment the line *"MYLIBS ="* in our Makefile to link to *linux-scalability* with a standard memory allocator (and comment other " *MYLIBS =XYZ"* lines)

After the *Makefile* modification run *"make clean; make"* and launch the driver. E.g. by

The paramaters of the driver deserve a short explanation. 512 is the size of chunks allocated via malloc. 10000 is the number of malloc calls and 10 is the number of threads each one performing 10000 mallocs of 512 byte.

Recompile the *linux-scalability* with different libraries and compare results. The following table demonstrates the comparison of results obtained with the hoard memory allocator and the standard memory allocator:

|  |  |
| --- | --- |
| **Hoard memory allocator**  **./linux-scalability 512 1000000 50** | **Standard memory allocator**  **./linux-scalability 512 1000000 50** |
| Average execution time = 6.831516 seconds, standard deviation = 2.918605. | Average execution time = 2.474888 seconds, standard deviation = 0.344976. |
| Average execution time = 6.513190 seconds, standard deviation = 2.799645. | Average execution time = 2.471997 seconds, standard deviation = 0.447987. |
| Average execution time = 5.668700 seconds, standard deviation = 2.440067. | Average execution time = 2.671545 seconds, standard deviation = 0.457854. |
| Average execution time = 2.900224 seconds, standard deviation = 1.203150. | Average execution time = 5.700411 seconds, standard deviation = 2.652924. |

|  |  |
| --- | --- |
| Average execution time = 6.641511 seconds, standard deviation = 2.966338. | Average execution time = 10.472085 seconds, standard deviation = 2.947556. |
| Average execution time = 4.995677 seconds, standard deviation = 2.002350. | Average execution time = 5.595637 seconds, standard deviation = 2.601380. |
| Average execution time = 7.666752 seconds, standard deviation = 2.882301. | Average execution time = 7.960329 seconds, standard deviation = 3.305003 |

We compute the standard deviation on the average execution time of the hoard memory allocator we will see that it is smaller then the standard deviation of the average execution time of the standard memory allocator. The same applies to the standard deviation of the standard deviation computed by the *linux-scalability.*

**Task 4 – libSimpleMTMM.a – a simple memory allocator**

We have also supplied a simple memory allocator *mtmm.c* that allocates/frees memory using *mmap/munmap*. This simple allocator is an example of how a memory chunk that was requested from the operating systems is returned back to the OS. It doesn't implements locks and doesn't checks for thread ID as described in the Hoard memory allocator pseudo code. You can use *mtmm.c* as a kickoff (starting point).

**Task 5 – Implement heap lock mechanism**

We will use pthread mutexes to implement the heap lock.

InitHeapLock(int i){

**int** r = pthread\_mutex\_init (&mutex[i], NULL);

**if** (r) {

**…**

}

…

}

**void** **lock** (**int i**) {

pthread\_mutex\_lock (&mutex[i]);

}

**void** **unlock** (**int i**) {

pthread\_mutex\_unlock (&mutex[i]);

}

**Task 6 –using hash function to map between thread ID and heap**

Since per-processor heap implementation requires OS support we will use hash function to map between thread ID and heap:

Heap number to be used by thread is thread ID modulo number of heaps.

To obtain thread ID one can use the following code snippet.

pthread\_t self;

self = pthread\_self();

**Task 7 – Using various data structures implementations**

Consider using an existing code for coding to save time and debugging effort. For example doubly connected linked lists may be reused from [here](http://codingfreak.blogspot.com/2012/02/implementation-of-doubly-linked-list-in.html).

The site contains mode data structure implementations you may find useful.

Note: To allocate nodes for the linked list, first allocated some big pool of nodes using the following code snippet:

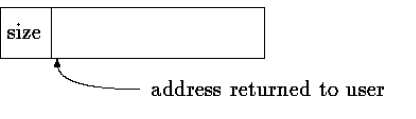
|  |
| --- |
| int fd = open("/dev/zero", O\_RDWR);  p = mmap(0, sizeof(ListNode) \* MAX\_NUMBER\_OF\_NODES, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE, fd, 0);  close(fd); |
| Note 1: You probably should add to the list node structure the indication whether this part of memory is allocated for the list mode or not. In this way you are actually implementing an auxiliary malloc for list nodes. For simplicity, this auxiliary malloc can take O(n) time. |
| Note 2: Define MAX\_NUMBER\_OF\_NODES = 2^16. After allocating MAX\_NUMBER\_OF\_NODES auxiliary malloc will simply fail causing to library malloc fail as well. |

**Task 8 – Using mmap to allocate memory for superblocks and large memory chunks**

Consider using the mmap to allocate virtual memory for superblocks and large (more then S/2) memory chunks.

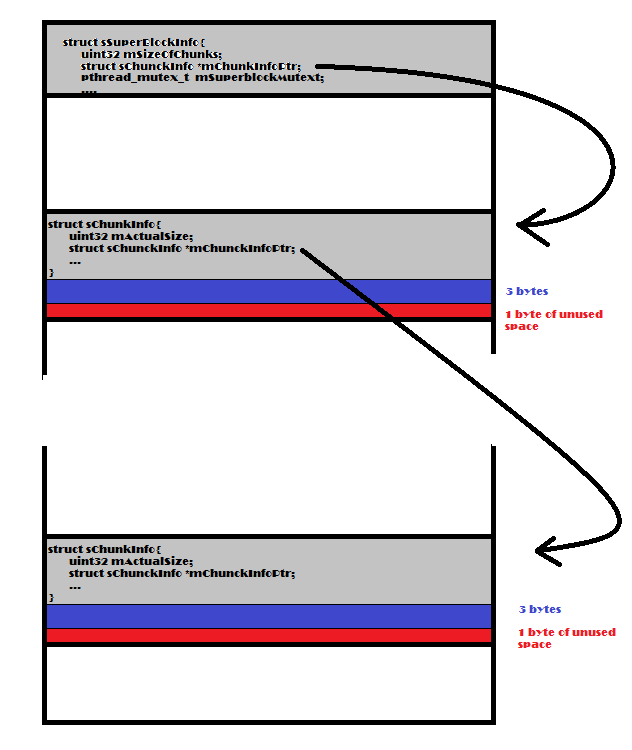
|  |
| --- |
| int fd = open("/dev/zero", O\_RDWR);  p = mmap(0, size, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE, fd, 0);  close(fd);  Note: don't forget to check return values! |

Since munmap requires size parameter, one has to save the size of the allocated memory. This size can be saved as depicted here:



We have also supplied a simple memory allocator mtmm.c that allocates/frees memory using mmap/munmap. This simple allocator contains an example of how size is saved before the address returned to user by the malloc and how the whole memory chunk that was requested from the operating systems is returned back to the OS.

Superblocks may be managed in the following manner:



**Task 9 – Implementing memory allocation for small chunks**

Small memory chunks (less then S/2 bytes) are allocated from superblocks. Heap i has an ordered list of superblocks from most full to least for each size class from 2 to S/2.

You may consider using the following structure to maintain the small memory chunks management (of course, you can add more members to the struct of you need):

struct sSizeClass

{

/\*

\* @breaf mSize – to hold the class size

\*/

unsigned int mSize;

/\*

\* @breaf – to hold ordered list of superblocks from most full to least.

\* All superblocks in this list are from size class mSize

\*/

OrderListType mOrderdListOfSuperblocks;

}

Then, you can have in your code a struct that describes per CPU heap. Of course you can add more members to the struct of you need.

typedef struct sHeap

{

/\*

\* @breaf mID – to hold the CPU ID

\*/

unsigned int mCPUID;

/\*

\* @breaf – to hold size classed for all sizes from 1 to log2(SUPERBLOCK\_SIZE)

\*/

struct sSizeClass sizeClasses[16];

} perCPUHeap\_t, globalHeap\_t;

And finally, we can define a class to hold 2 per CPU heaps that may be called sHoard (of course, you can add more members to the struct of you need):

struct sHoard

{

/\*

\* @breaf – for simplicity**,** our implementation of hoard allocator

\* will use only 2 "per CPU" heaps and one global heap

\*/

perCPUHeap\_t mCPUHeap[2];

globalHeap\_t mGlobalHeap;

}

**Task 10 – Defining parameters that govern the allocation**

The following parameters will govern the allocation policy:

S is defined in *mtmm.h* as SUPERBLOCK\_SIZE

***f*** = 1/4 – define it in your code (e.g. in mtmm.c)

K = 0 – define it in your code (e.g. in mtmm.c)

Fullness range: all superblocks that are between 3/4 and completely empty are in the same bin.

References

[1]

<http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/>

[2]

**“The C Programming Language”** by Kernighan and Ritchie (attached to maman12.zip as a pfd)

[3] <http://en.wikibooks.org/wiki/C_Programming/C_Reference/stdlib.h/malloc#Implementations>

[4]

<http://en.wikipedia.org/wiki/Hoard_memory_allocator>

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_