Understanding glibc malloc

Posted on February 10, 2015 by sploitfun

I always got fascinated by heap memory. Questions such as

- How heap memory is obtained from kernel?
- How efficiently memory is managed?
- Is it managed by kernel or by library or by application itself?
- Can heap memory be exploited?

were in my mind for quite some time. But only recently I got time to understand about it. So here I would like to share my fascination turned knowledge!! Out there in the wild, many memory allocators are available:

- dlmalloc - General purpose allocator
- ptmalloc2 - glibc
- jemalloc - FreeBSD and Firefox
- tcmalloc - Google
- libumem - Solaris
- …

Every memory allocator claims they are fast, scalable and memory efficient!! But not all allocators can be suited well for our application. Memory hungry application's performance largely depends on memory allocator performance. In this post, I will only talk about '**glibc malloc**' memory allocator. In future, I am hoping to cover up other memory allocators. Throughout this post, for better understanding of 'glibc malloc', I will link its recent source code. So buckle up, lets get started with glibc malloc!!

History: ptmalloc2 was forked from dlmalloc. After fork, threading support was added to it and got released in 2006. After its official release, ptmalloc2 got integrated into glibc source code. Once its integration, code changes were made directly to glibc malloc source code itself. Hence there could be lot of changes between ptmalloc2 and glibc's malloc implementation.

System Calls: As seen in this post malloc internally invokes either brk or mmap syscall.

Threading: During early days of linux, dlmalloc was used as the default memory allocator. But later due to ptmalloc2's threading support, it became the default memory allocator for linux. Threading support helps in improving memory allocator performance and hence application performance. In dlmalloc when two threads call malloc at the same time ONLY one thread can enter the critical section, since freelist data structure is shared among all the available threads. Hence memory allocation takes time in multi threaded applications, resulting in performance degradation. While in ptmalloc2, when two threads call malloc at the same time memory is allocated immediately since each thread maintains a separate heap segment and hence freelist data structures maintaining those heaps are also separate. This act of maintaining separate heap and freelist data structures for each thread is called **per thread arena**.

Example:

```c
/* Per thread arena example. */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>

void* threadFunc(void* arg) {
        printf("Before malloc in thread 1\n");
        getchar();
        char* addr = (char*) malloc(1000);
        printf("After malloc and before free in thread 1\n");
        getchar();
        free(addr);
        printf("After free in thread 1\n");
        getchar();
}

int main() {
        pthread_t t1;
        void* s;
        int ret;
        char* addr;

        printf("Welcome to per thread arena example::%d\n",getpid());
        printf("Before malloc in main thread\n");
        getchar();
        addr = (char*) malloc(1000);
        printf("After malloc and before free in main thread\n");
        getchar();
        free(addr);
        printf("After free in main thread\n");
        getchar();
        ret = pthread_create(&t1, NULL, threadFunc, NULL);
        if(ret)
        {
                printf("Thread creation error\n");
                return -1;
        }
        ret = pthread_join(t1, &s);
        if(ret)
        {
                printf("Thread join error\n");
                return -1;
        }
        return 0;
}
```

Output Analysis:

Before malloc in main thread: In the below output we can see that there is NO heap segment yet and no per thread stack too since thread1 is not yet created.

```
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ ./mthread
Welcome to per thread arena example::6501
Before malloc in main thread
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ cat /proc/6501/maps
08048000-08049000 r-xp 00000000 08:01 539625
/home/sploitfun/ptmalloc.ppt/mthread/mthread
08049000-0804a000 r--p 00000000 08:01 539625
/home/sploitfun/ptmalloc.ppt/mthread/mthread
0804a000-0804b000 rw-p 00001000 08:01 539625
/home/sploitfun/ptmalloc.ppt/mthread/mthread
b7e05000-b7e07000 rw-p 00000000 00:00 0
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$
```

After malloc in main thread: In the below output we can see that heap segment is created and its lies just above the data segment (0804b000-0806c000), this shows heap memory is created by increasing program break location (ie) using **brk** syscall). Also do note that eventhough user requested only 1000 bytes, heap memory of size 132 KB is created. This contiguous region of heap memory is called **arena**. Since this arena is created by main thread its called **main arena**. Further allocation requests keeps using this arena until it runs out of free space. When arena runs out of free space, it can grow by increasing program break location (After growing top chunk's size is adjusted to include the extra space). Similarly arena can also shrink when there is lot of free space on top chunk.

NOTE: Top chunk is the top most chunk of an arena. For further details about it, see "Top Chunk" section below.

```
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ ./mthread
Welcome to per thread arena example::6501
Before malloc in main thread
After malloc and before free in main thread
...
sploitfun@sploitfun-VirtualBox:~/lsploits/hof/ptmalloc.ppt/mthread$ cat /proc/6501/maps
08048000-08049000 r-xp 00000000 08:01 539625
/home/sploitfun/ptmalloc.ppt/mthread/mthread
08049000-0804a000 r--p 00000000 08:01 539625
/home/sploitfun/ptmalloc.ppt/mthread/mthread
0804a000-0804b000 rw-p 00001000 08:01 539625
/home/sploitfun/ptmalloc.ppt/mthread/mthread
0804b000-0806c000 rw-p 00000000 00:00 0          [heap]
b7e05000-b7e07000 rw-p 00000000 00:00 0
...
```

```
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$
```

After free in main thread: In the below output we can see that when allocated memory region is freed, memory behind it doesnt get released to the operating system immediately. Allocated memory region (of size 1000 bytes) is released only to 'glibc malloc' library, which adds this freed block to main arenas bin (In glibc malloc, freelist datastructures are referred as bins). Later when user requests memory, 'glibc malloc' doesnt get new heap memory from kernel, instead it will try to find a free block in bin. And only when no free block exists, it obtains memory from kernel.

```
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ ./mthread
Welcome to per thread arena example::6501
Before malloc in main thread
After malloc and before free in main thread
After free in main thread
...
sploitfun@sploitfun-VirtualBox:~/lsploits/hof/ptmalloc.ppt/mthread$ cat /proc/6501/maps
08048000-08049000 r-xp 00000000 08:01 539625
/home/sploitfun/ptmalloc.ppt/mthread/mthread
08049000-0804a000 r--p 00000000 08:01 539625
/home/sploitfun/ptmalloc.ppt/mthread/mthread
0804a000-0804b000 rw-p 00001000 08:01 539625
/home/sploitfun/ptmalloc.ppt/mthread/mthread
0804b000-0806c000 rw-p 00000000 00:00 0          [heap]
b7e05000-b7e07000 rw-p 00000000 00:00 0
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$
```

Before malloc in thread1: In the below output we can see that there is NO thread1 heap segment but now thread1's per thread stack is created.

```
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ ./mthread
Welcome to per thread arena example::6501
Before malloc in main thread
After malloc and before free in main thread
After free in main thread
Before malloc in thread 1
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ cat /proc/6501/maps
08048000-08049000 r-xp 00000000 08:01 539625
/home/sploitfun/ptmalloc.ppt/mthread/mthread
08049000-0804a000 r--p 00000000 08:01 539625
/home/sploitfun/ptmalloc.ppt/mthread/mthread
0804a000-0804b000 rw-p 00001000 08:01 539625
/home/sploitfun/ptmalloc.ppt/mthread/mthread
0804b000-0806c000 rw-p 00000000 00:00 0          [heap]
```

```
b7604000-b7605000 ---p 00000000 00:00 0
b7605000-b7e07000 rw-p 00000000 00:00 0          [stack:6594]
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$
```

After malloc in thread1: In the below output we can see that thread1's heap segment is created. And its lies in memory mapping segment region (b7500000-b7521000 whose size is 132 KB) and hence this shows heap memory is created using **mmap** syscall unlike main thread (which uses sbrk). Again here, eventhough user requested only 1000 bytes, heap memory of size 1 MB is mapped to process address space. Out of these 1 MB, only for 132KB read-write permission is set and this becomes the heap memory for this thread. This contiguous region of memory (132 KB) is called **thread arena**.

NOTE: When user request size is more than 128 KB (lets say malloc(132*1024)) and when there is not enough space in an arena to satisfy user request, memory is allocated using mmap syscall (and NOT using sbrk) irrespective of whether a request is made from main arena or thread arena.

```
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ ./mthread
Welcome to per thread arena example::6501
Before malloc in main thread
After malloc and before free in main thread
After free in main thread
Before malloc in thread 1
After malloc and before free in thread 1
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ cat /proc/6501/maps
08048000-08049000 r-xp 00000000 08:01 539625
/home/sploitfun/ptmalloc.ppt/mthread/mthread
08049000-0804a000 r--p 00000000 08:01 539625
/home/sploitfun/ptmalloc.ppt/mthread/mthread
0804a000-0804b000 rw-p 00001000 08:01 539625
/home/sploitfun/ptmalloc.ppt/mthread/mthread
0804b000-0806c000 rw-p 00000000 00:00 0          [heap]
b7500000-b7521000 rw-p 00000000 00:00 0
b7521000-b7600000 ---p 00000000 00:00 0
b7604000-b7605000 ---p 00000000 00:00 0
b7605000-b7e07000 rw-p 00000000 00:00 0          [stack:6594]
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$
```

After free in thread1: In the below output we can see that freeing allocated memory region doesnt release heap memory to the operating system. Instead allocated memory region (of size 1000 bytes) is released to 'glibc malloc', which adds this freed block to its thread arenas bin.

```
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ ./mthread
Welcome to per thread arena example::6501
Before malloc in main thread
After malloc and before free in main thread
After free in main thread
Before malloc in thread 1
After malloc and before free in thread 1
After free in thread 1
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ cat /proc/6501/maps
08048000-08049000 r-xp 00000000 08:01 539625
/home/sploitfun/ptmalloc.ppt/mthread/mthread
08049000-0804a000 r--p 00000000 08:01 539625
/home/sploitfun/ptmalloc.ppt/mthread/mthread
0804a000-0804b000 rw-p 00001000 08:01 539625
/home/sploitfun/ptmalloc.ppt/mthread/mthread
0804b000-0806c000 rw-p 00000000 00:00 0          [heap]
b7500000-b7521000 rw-p 00000000 00:00 0
b7521000-b7600000 ---p 00000000 00:00 0
b7604000-b7605000 ---p 00000000 00:00 0
b7605000-b7e07000 rw-p 00000000 00:00 0          [stack:6594]
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$
```

Arena:

Number of arena's: In above example, we saw main thread contains main arena and thread 1 contains its own thread arena. So can there be a one to one mapping between threads and arena, irrespective of number of threads? Certainly not. An insane application can contain more number of threads (than number of cores), in such a case, having one arena per thread becomes bit expensive and useless. Hence for this reason, application's arena limit is based on number of cores present in the system.

```
For 32 bit systems:
     Number of arena = 2 * number of cores.
For 64 bit systems:
     Number of arena = 8 * number of cores.
```

Multiple Arena:

Example: Lets say a multithreaded application (4 threads - Main thread + 3 user threads) runs on a 32 bit system which contains 1 core. Here no of threads (4) > 2*no of cores (2). Hence in such a case, 'glibc malloc' makes sure that multiple arenas are shared among all available threads. But how its shared?

- When main thread, calls malloc for the first time already created main arena is used without any contention.

- When thread 1 and thread 2 calls malloc for the first time, a new arena is created for them and its used without any contention. Until this point threads and arena have one-to-one mapping.
- When thread 3 calls malloc for the first time, number of arena limit is calculated. Here arena limit is crossed, hence try reusing existing arena's (Main arena or Arena 1 or Arena 2)
- Reuse:
  - Once loop over the available arenas, while looping try to lock that arena.
  - If locked successfully (lets say main arena is locked successfully), return that arena to the user.
  - If no arena is found free, block for the arena next in line.
- Now when thread 3 calls malloc (second time), malloc will try to use last accessed arena (main arena). If main arena is free its used else thread3 is blocked until main arena gets freed. Thus now main arena is shared among main thread and thread 3.

Multiple Heaps:

Primarily below three datastructures are found in 'glibc malloc' source code:

heap_info - Heap Header - A single thread arena can have multiple heaps. Each heap has its own header. Why multiple heaps needed? To begin with every thread arena contains ONLY one heap, but when this heap segment runs out of space, new heap (non contiguous region) gets mmap'd to this arena.

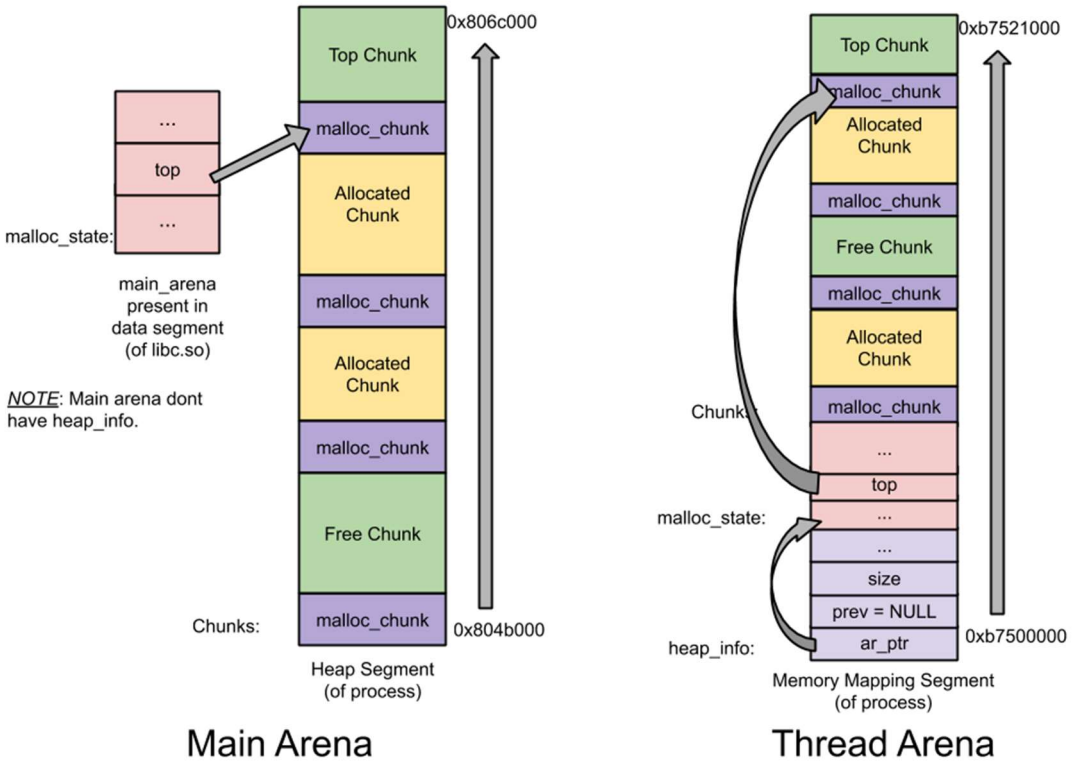
malloc_state - Arena Header - A single thread arena can have multiple heaps, but for all those heaps only a single arena header exists. Arena header contains information about bins, top chunk, last remainder chunk…

malloc_chunk - Chunk Header - A heap is divided into many chunks based on user requests. Each of those chunks has its own chunk header.

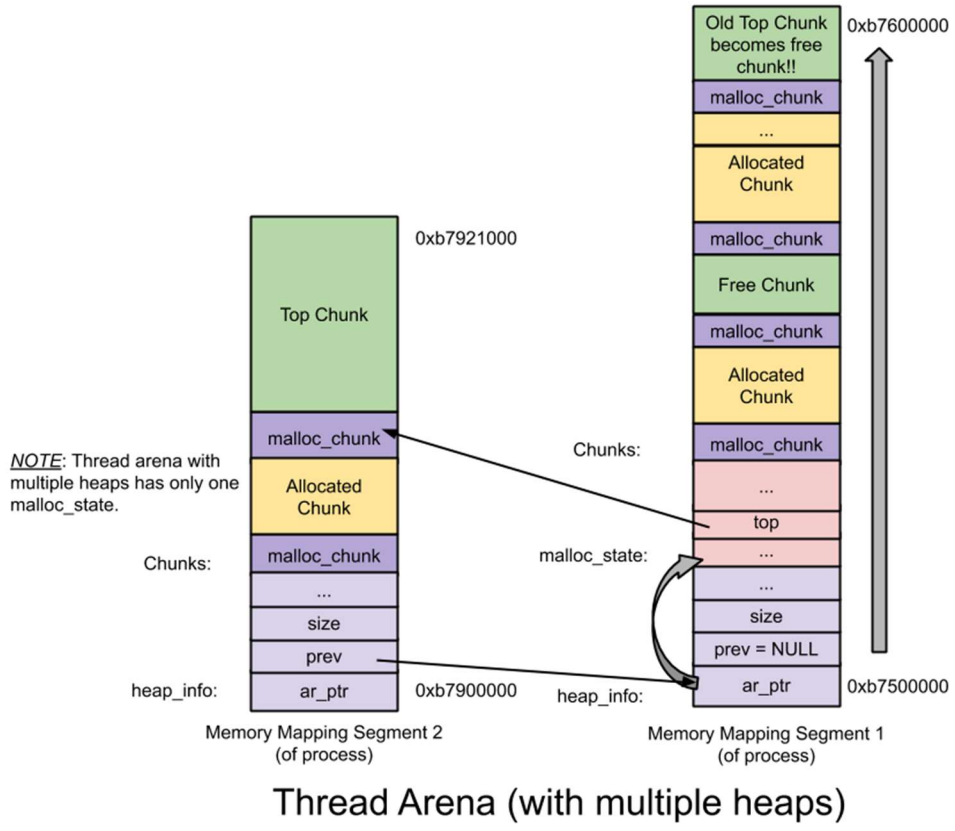
NOTE:

- Main arena dont have multiple heaps and hence no heap_info structure. When main arena runs out of space, sbrk'd heap segment is extended (contiguous region) until it bumps into memory mapping segment.
- Unlike thread arena, main arena's arena header isnt part of sbrk'd heap segment. Its a global variable and hence its found in libc.so's data segment.

Pictorial view of main arena and thread arena (single heap segment):



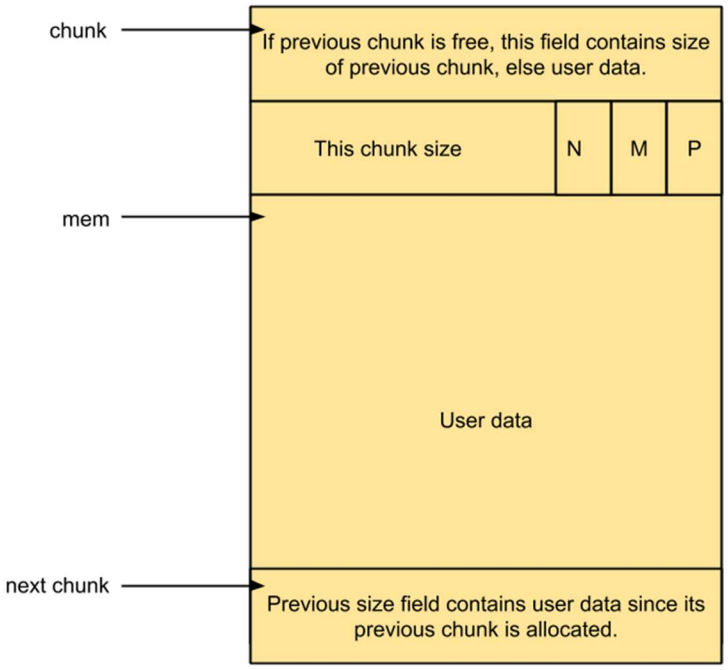
Pictorial view of thread arena (multiple heap segment's):



Chunk: A chunk found inside a heap segment can be one of the below types:

- Allocated chunk
- Free chunk
- Top chunk
- Last Remainder chunk

Allocated chunk:



Allocated Chunk

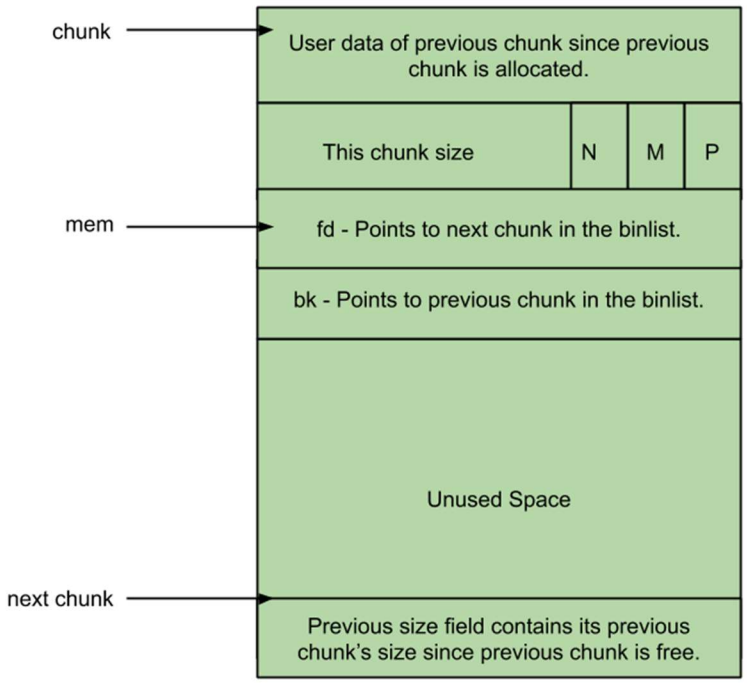
prev_size: If the previous chunk is free, this field contains the size of previous chunk. Else if previous chunk is allocated, this field contains previous chunk's user data.
size: This field contains the size of this allocated chunk. Last 3 bits of this field contains flag information.

- PREV_INUSE (P) - This bit is set when previous chunk is allocated.
- IS_MMAPPED (M) - This bit is set when chunk is mmap'd.
- NON_MAIN_ARENA (N) - This bit is set when this chunk belongs to a thread arena.

NOTE:

- Other fields of malloc_chunk (like fd, bk) is NOT used for allocated chunk. Hence in place of these fields user data is stored.
- User requested size is converted into usable size (internal representation size) since some extra space is needed for storing malloc_chunk and also for alignment purposes. Conversion takes place in such a way that last 3 bits of usable size is never set and hence its used for storing flag information.

Free Chunk:



Free Chunk

prev_size: No two free chunks can be adjacent together. When both the chunks are free, its gets combined into one single free chunk. Hence always previous chunk to this freed chunk would be allocated and therefore prev_size contains previous chunk's user data.
size: This field contains the size of this free chunk.
fd: Forward pointer - Points to next chunk in the same bin (and NOT to the next chunk present in physical memory).
bk: Backward pointer - Points to previous chunk in the same bin (and NOT to the previous chunk present in physical memory).

Bins: Bins are the freelist datastructures. They are used to hold free chunks. Based on chunk sizes, different bins are available:

- Fast bin
- Unsorted bin
- Small bin
- Large bin

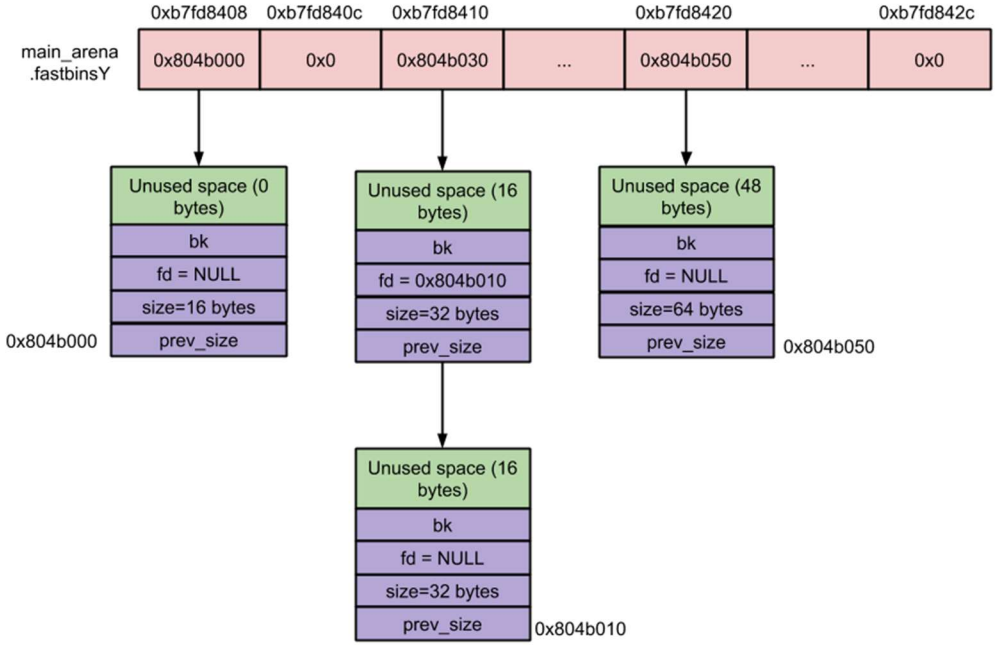
Datastructures used to hold these bins are:

fastbinsY: This array hold fast bins.
bins: This array hold unsorted, small and large bins. Totally there are 126 bins and its divided as follows:

- Bin 1 - Unsorted bin
- Bin 2 to Bin 63 - Small bin
- Bin 64 to Bin 126 - Large bin

Fast Bin: Chunks of size 16 to 80 bytes is called a fast chunk. Bins holding fast chunks are called fast bins. Among all the bins, fast bins are faster in memory allocation and deallocation.

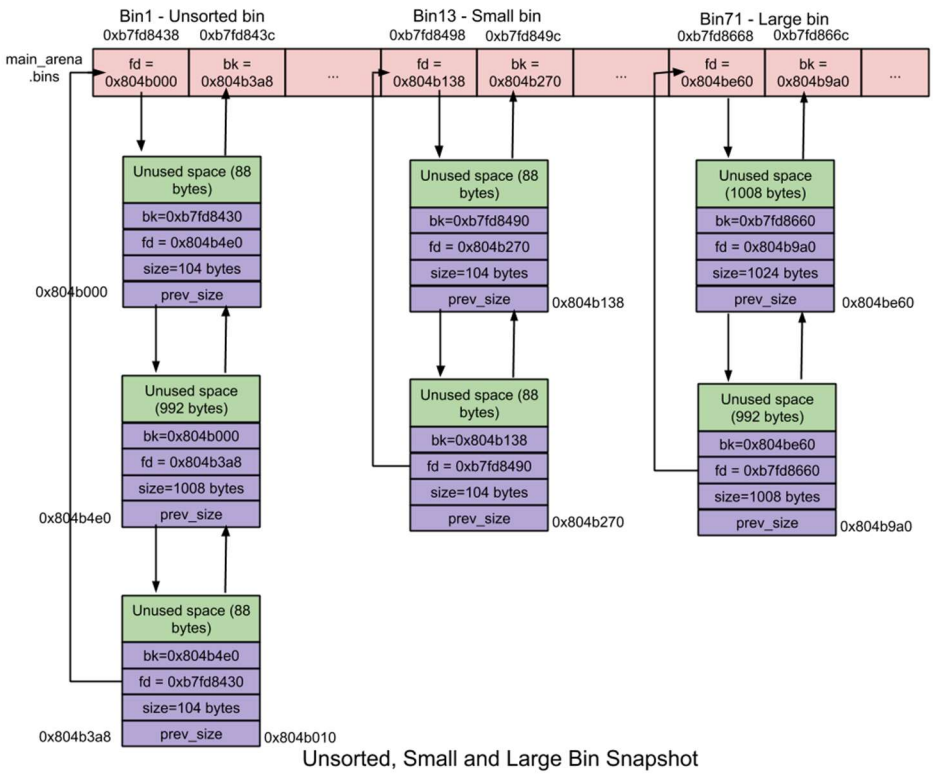
- Number of bins - 10
  - Each fast bin contains a single linked list (a.k.a binlist) of free chunks. Single linked list is used since in fast bins chunks are not removed from the middle of the list. Both addition and deletion happens at the front end of the list - LIFO.
- Chunk size - 8 bytes apart
  - Fast bins contain a binlist of chunks whose sizes are 8 bytes apart. ie) First fast bin (index 0) contains binlist of chunks of size 16 bytes, second fast bin (index 1) contains binlist of chunks of size 24 bytes and so on…
  - Chunks inside a particular fast bin are of same sizes.
- During malloc initialization, maximum fast bin size is set to 64 (!80) bytes. Hence by default chunks of size 16 to 64 is categorized as fast chunks.
- No Coalescing - Two chunks which are free can be adjacent to each other, it doesnt get combined into single free chunk. No coalescing could result in external fragmentation but it speeds up free!!
- malloc(fast chunk) -
  - Initially fast bin max size and fast bin indices would be empty and hence eventhough user requested a fast chunk, instead of fast bin code, small bin code tries to service it.
  - Later when its not empty, fast bin index is calculated to retrieve its corresponding binlist.
  - First chunk from the above retrieved binlist is removed and returned to the user.
- free(fast chunk) -
  - Fast bin index is calculated to retrieve its corresponding binlist.
  - This free chunk gets added at the front position of the above retrieved binlist.



Fast Bin Snapshot

Unsorted Bin: When small or large chunk gets freed instead of adding them in to their respective bins, its gets added into unsorted bin. This approach gives 'glibc malloc' a second chance to reuse the recently freed chunks. Hence memory allocation and deallocation speeds up a bit (because of unsorted bin) since time taken to look for appropriate bin is eliminated.

- Number of bins - 1
  - Unsorted bin contains a circular double linked list (a.k.a binlist) of free chunks.
- Chunk size - There is no size restriction, chunks of *any* size belongs to this bin.



Unsorted, Small and Large Bin Snapshot

Small Bin: Chunks of size less than 512 bytes is called as small chunk. Bins holding small chunks are called small bins. Small bins are faster than large bins (but slower than fast bins) in memory allocation and deallocation.

- Number of bins - 62
  - Each small bin contains a circular double linked list (a.k.a binlist) of free chunks. Double linked list is used since in small bins chunks are unlinked from the middle of the list. Here addition happens at the front end and deletion happens at the rear end of the list - FIFO.
- Chunk Size - 8 bytes apart
  - Small bin contains a binlist of chunks whose sizes are 8 bytes apart. ie) First Small bin (Bin 2) contains binlist of chunks of size 16 bytes, second small bin (Bin 3) contains binlist of chunks of size 24 bytes and so on…
  - Chunks inside a small bin are of same sizes and hence it doesnt need to be sorted.
- Coalescing - Two chunks which are free cant be adjacent to each other, it gets combined into single free chunk. Coalescing eliminates external fragmentation but it slows up free!!
- malloc(small chunk) -
  - Initially all small bins would be NULL and hence eventhough user requested a small chunk, instead of small bin code, unsorted bin code tries to service it.
  - Also during the first call to malloc, small bin and large bin datastructures (bins) found in malloc_state is initialized ie) bins would point to itself signifying they are empty.

    - Later when small bin is non empty, last chunk from its corresponding binlist is removed and returned to the user.
- free(small chunk) -
    - While freeing this chunk, check if its previous or next chunk is free, if so coalesce ie) unlink those chunks from their respective linked lists and then add the new consolidated chunk into the beginning of unsorted bin's linked list.

Large Bin: Chunks of size greater than equal to 512 is called a large chunk. Bins holding large chunks are called large bins. Large bins are slower than small bins in memory allocation and deallocation.

- Number of bins - 63
    - Each large bin contains a circular double linked list (a.k.a binlist) of free chunks. Double linked list is used since in large bins chunks are added and removed at any position (front or middle or rear).
    - Out of these 63 bins:
        - 32 bins contain binlist of chunks of size which are 64 bytes apart. ie) First large bin (Bin 65) contains binlist of chunks of size 512 bytes to 568 bytes, second large bin (Bin 66) contains binlist of chunks of size 576 bytes to 632 bytes and so on…
        - 16 bins contain binlist of chunks of size which are 512 bytes apart.
        - 8 bins contain binlist of chunks of size which are 4096 bytes apart.
        - 4 bins contain binlist of chunks of size which are 32768 bytes apart.
        - 2 bins contain binlist of chunks of size which are 262144 bytes apart.
        - 1 bin contains a chunk of remaining size.
    - Unlike small bin, chunks inside a large bin are NOT of same size. Hence they are stored in decreasing order. Largest chunk is stored in the front end while the smallest chunk is stored in the rear end of its binlist.
- Coalescing - Two chunks which are free cant be adjacent to each other, it gets combined into single free chunk.
- malloc(large chunk) -
    - Initially all large bins would be NULL and hence eventhough user requested a large chunk, instead of large bin code, next largest bin code tries to service it.
    - Also during the first call to malloc, small bin and large bin datastructures (bins) found in malloc_state is initialized ie) bins would point to itself signifying they are empty.
    - Later when large bin is non empty, if the largest chunk size (in its binlist) is greater than user requested size, binlist is walked from rear end to front end, to find a suitable chunk whose size is near/equal to user requested size. Once found, that chunk is split into two chunks
        - User chunk (of user requested size) - returned to user.
        - Remainder chunk (of remaining size) - added to unsorted bin.
    - If largest chunk size (in its binlist) is lesser than user requested size, try to service user request by using the next largest (non empty) bin. Next largest bin code scans the binmaps to find the next largest bin which is non empty, if any

such bin found, a suitable chunk from that binlist is retrieved, split and returned to the user. If not found, try serving user request using top chunk.

- free(large chunk) - Its procedure is similar to free(small chunk).

Top Chunk: Chunk which is at the top border of an arena is called top chunk. It doesnt belong to any bin. Top chunk is used to service user request when there is NO free blocks, in any of the bins. If top chunk size is greater than user requested size top chunk is split into two:

- User chunk (of user requested size)
- Remainder chunk (of remaining size)

The remainder chunk becomes the new top. If top chunk size is lesser than user requested size, top chunk is extended using sbrk (main arena) or mmap (thread arena) syscall.

Last Remainder Chunk: Remainder from the most recent split of a small request. Last remainder chunk helps to improve locality of reference ie) consecutive malloc request of small chunks might end up being allocated close to each other.

But out of the many chunks available in an arena, which chunk qualifies to be last remainder chunk?

When a user request of small chunk, cannot be served by a small bin and unsorted bin, binmaps are scanned to find next largest (non empty) bin. As said earlier, on finding the next largest (non empty) bin, its split into two, user chunk gets returned to the user and remainder chunk gets added to the unsorted bin. In addition to it, it becomes the new last remainder chunk.

How locality of reference is achieved?

Now when user subsequently request's a small chunk and if the last remainder chunk is the only chunk in unsorted bin, last remainder chunk is split into two, user chunk gets returned to the user and remainder chunk gets added to the unsorted bin. In addition to it, it becomes the new last remainder chunk. Thus subsequent memory allocations end up being next to each other.