Heap

An application has two types of memory space in the memory: Stack, Heap. In a short definition, we can tell Stack is used for static memory allocation, while, Heap is used for Dynamic memory allocation. So what is the heap exactly? Every application has a region call Heap inside memory, Unlike Stack, the Application can dynamically use this region, which means, They can request memory and Free memory space on the run time dynamically. We should mention this region is global, as the result, heap-allocated memory does not belong to a specific function or scope like a local variable All global variables are defined in Heap by default.

Allocator

Every platform has got own way to interact with heap memory, there is lots of allocator like iOS allocator, Free BSD allocator, and Linux allocator. work specifically on Glibc memory allocator which drives from ptmalloc heap implementations, which is itself derived from dlmalloc, so, at first we want to discuss Glibc allocations algorithms and Methods.

Sample in c :

**typedef struct**

{

**int** field1;

**char\*** field2;

} SomeStruct;

**int** main()

{

SomeStruct\* myObject = (SomeStruct\*)malloc(**sizeof**(SomeStruct));

**if**(myObject != NULL)

{

myObject->field1 = 1234;

myObject->field2 = “Hello World!”;

do\_stuff(myObject);

free(myObject);

}

**return** 0;

}

Glibc has other allocators like calloc, realloc,memalign too, which, can be free after use via free().

Heap manager can’t simply allocate a part of memory and return the address to the user, Heap Manager needs a way to store metadata about the allocated memory alongside keep tracking of Free space, Also, Heap manager needs to align memory in 8 bytes for 32bit system, and 16Byte in 64bit system.

This allocation metadata and padding store alongside the allocated memory by malloc which returns to the user. For this reason, the Heap manager uses a concept called ‘Chunk’, each chunk usually bigger than request memory allocation because of metadata. When the user sends a request for memory allocation, the heap manager finds a proper space for user data and metadata, then, returns a pointer to the Userdata section of a chunk.

# malloc\_chunk :

struct malloc\_chunk {

INTERNAL\_SIZE\_T mchunk\_prev\_size; /\* Size of previous chunk (if free). \*/

INTERNAL\_SIZE\_T mchunk\_size; /\* Size in bytes, including overhead. \*/

struct malloc\_chunk\* fd; /\* double links -- used only if free. \*/

struct malloc\_chunk\* bk;

/\* Only used for large blocks: pointer to next larger size. \*/

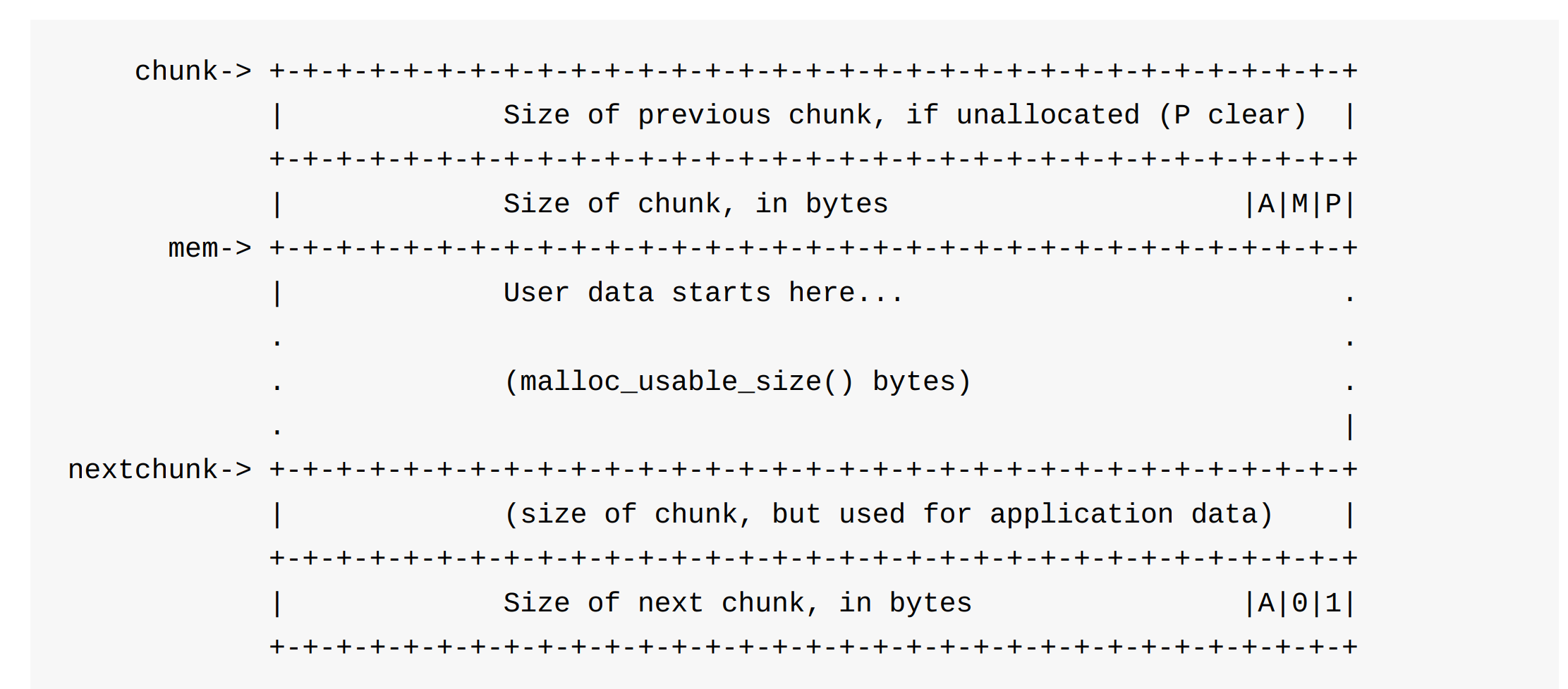
struct malloc\_chunk\* fd\_nextsize; /\* double links -- used only if free. \*/

struct malloc\_chunk\* bk\_nextsize;

};

typedef struct malloc\_chunk\* mchunkptr;

# Allocated chunk:



# Free chunk

# Chunk Allocation

First, we discuss the Heap manager in a simple algorithm, then, we discuss each part and improvement of the Heap allocation algorithm. Whats happens when the user requests a memory allocation :

1. Heap manager cheek previously freed chunk, if there is a chunk which big enough, then, return it.
2. If the Heap manager unable to find a previously freed chunk, then look at the top of the heap, If there is available free space, then allocated a new chunk and return it.
3. If there is not enough space in Heap anymore, then ask Kernel for new memory allocation to the end of the heap and allocate chunk from this new space.
4. If all the above space failed then malloc returns NULL, which means, we can’t allocate new space.

# Allocation from Freed Chunks

Heap manager track of Freed chunks via some linked list call ‘bins’. When a new allocation request arrives, the heap manager lock at this series of link list to find a big enough chunk. There are several different types of bins which we discuss later.

# Allocation from Top of Heap

So what’s happen if there were no freed space? In this situation, the Heap manager checks the remaining space at the end of the heap, if there is enough space to create a chunk then allocate a new chunk from the top of the heap space.

# Ask Kernel for more memory

If there is not enough space at the end of the heap, the heap manager will ask Linux Kernel for more memory. Now we have to discuss another system call ‘brk’. When a program has been started, the heap manager calls ‘sbrk’ which used the 'brk' system call. This system call allocates more memory just after where the program gets load which is the end of the heap.

After a while, this system call will break, Because, the newly allocated region at the end of the program space will collide with another thing in the process’s space. Once this situation happens heap manager will unable to allocate contiguous memory space, so, by calling mmap try to allocate non-contiguous memory space.

## MMAP

As we mentioned, the Heap manager uses MMAP for large memory allocation instead of sbrk. Chunks metadata contain a special flag for these reasons which indicate this chunk allocated off-heap via mmap. After the user releases the memory by free(), these chunks return to the heap manager.

# Arena

Concurrency is a challenge in multi-thread applications, so, Heap manager is not safe from this issue. There are many solutions to overcome this problem. In the early days' heap manager used a simple global lock before every heap operation to overcome this situation, however, this approach has a great cost. In multi-thread applications with heavy use of heap, this strategy leads to huge performance issues, because, heap needs to lock many times.

To improve the performance ptmalloc2 introduces a concept called ‘Arena’. In this approach different thread has their Arena, also, Each arena manages their chunks, as the result, threads can work at the same time on different Arena without altering each other data.

When the process create a new thread, the heap manager finds an unused Arena up to the maximum allowed number which is (2\* the number of CPU core) for 32-bit and (8\* the number of CPU core) for 64-bit. So, what happens when we reach max number? Thread has to share Arena.

As we saw before, the main Areas create and grow by the sbrk system call, however, this is not true for the second heap. The second heap creates via mmap.

# Sub Heap

As we mentioned subhead works as like the main heap, however, they have some differences. The main heap is located right after where the program starts in memory, they grow with ‘sbrk’ system call. Sub heap create by use of mmap, so, they are not located where the program starts contiguously, moreover, Heap manager expands them by ‘protect.

How heap manager create and manage the sub heap? In the first step, the heap manager asks the kernel to reserve an area of memory for the subheap by malloc. Reserving does not allocate memory to subheap, it just tells the kernel, Don’t use this area. Mmap by flags required page as PROT\_NON achieve this goal.

On the next step, when the main heap expands by ‘sbrk’, the heap manager allocates the previously reserved area to subheap by calling ‘protect. What mprotect do? It concerts PROT\_NON to PROT\_READ|PROT\_WRITE. In this way, at least the kernel allocates real physical memory to each subheap, subheap expand in mmap area until It fills all mmap reserve area.

# free()

Whats happen when user does not need allocated space anymore? Simply , by calling heap can free up the space . Now , the question is how free() works ? First , it needs to calculate the chunk address from user pointer . User has pointer to user data section of chunk , but , free needs the actual address of chunk , so , by subtracting the pointer address from metadata size we can find the chunks address . You may ask what happen if we send an invalid pointer? This may lead to memory corruption , as the result , heap manager do some security check before free up the space :

1. Allocation is align on 8-bit
2. Validation of chunk size
3. Chunk is in the Arena address space .
4. Not already free

# Bins

Bin is a mechanism that the Heap manager used to manage and organize recently freed chunks, so, they can be reused during the next allocations. The simplest solution to keep track of freed space is to store them in a giant bin, although this could work It is not a good solution. Malloc is very wildly used, so, this approach leads to a performance problem.

To improve performance, the heap manager uses a different types of bins. There is 5 different type of bins: small bin, large bin, fast bin, unsorted bin, tcache. All of the mentioned bins exist in the same area of heap manager source code. This is a list with 127 items, the index 0 is unused.

Now, We want to discuss how the heap manager recycles a chunk for a new allocation. First, let's consider a simple schema :

1. If the M-bit in metadata is set, then this chunk is allocated off-heap, so, It should be unmmap.
2. Otherwise, if the chunk right before this chunk is free, then they will be merge
3. If the chunk right after this chunk is free, then they will be merge
4. After the merge, if the larger chunk is located at the top of the heap, then it will be absorbed into the end of the heap instead of adding to bins
5. Otherwise, the freed chunk will add to the corresponding bin

## Small Bins

There are 62 small bins, Each of them store the same size of the chunk, as the result, each list is sorted by default, also, store and retrieve chunk from small bins is fast.

## Large Bins

For chunks over 512 bytes, the Heap manager uses another strategy call Large Bins. There are 63 Large Bins. The main difference between small and large bins is large bins use a size range instead of a fixed size for every bin. Every bin in the large bins store chunk with a unique size range, in a way, they don’t overlap with each other.

The other difference is large bins need to manually sort during the insertion of a chunk. As the result large bins are slower than small bins, however, Large bins used less than small bins in a real program, so, this is not a big problem.

## Unsorted Bins

Usually free() follow by an allocation of the same size in a real program. The heap manager introduced another optimization call Unsorted bins. In this optimization, the heap manager merges freed chunk with its neighbors and puts it inside a general bin call Unsorted bins instead of finding the corresponding bin. When a new allocation request arrives, the heap manager compares the request to each item in an unsorted bin, If an item is big enough for request, the heap manager returns it, otherwise move an item to the corresponding bin.

## Fast Bins

This is the second optimizations on heap manager. These bins keep recently freed small chunk in a list, moreover, chunk in this bin does not merge with neighbors, instead, fast bin keeps them alive, so, If program sends the same size request, then heap manager can use this chunks.

The main difference between fast bins and small bins is fast bins do not merge chunks with neighbors. How fast bin do that? The heap manager doesn’t free up this chunk by don’t set P-bit. Of course, this strategy has a downside too, the memory of the process becomes full or fragment after a while, to overcome this issue heap manager need to ‘flashing ‘ memory from time to time.

## Tcache bins

Tcache is the last optimizations in the heap manager. Usually, a process has multiple threats, and a resource-like heap is shared between threats. A shared resource between threats can lead to a problem called ‘race condition’. As we mentioned before, a simple strategy to overcome race conditions is to lock. The lock gave temporary ownership of resources to one threat. When the job of the first threat finishes, then the second one can proceed. The lock has a great cost for heap managers which is frequently used by a program. As we talk before, the Heap manager overcomes this problem by using a secondary Arena for each threat until hits the threshold, moreover, it uses tcache per-threat cache to reduce the cost of the lock. Tcache speeds up the allocation by having per-threat bins of a small chunk. When a threat sends an allocation request, the first tcache of that threat check for available chunk, If tcache have a big enough chunk, then threat use it and don’t need to wait for a heap lock.

## Final Strategy

malloc()

1. If the size corresponds with a *tcache* bin and there is a *tcache* chunk available, return that immediately
2. If the request is enormous allocate a chunk off-heap via *mmap*
3. *Otherwise we obtain the arena heap lock and then perform the following strategies, in order*
   1. Try the *fastbin/smallbin* recycling strategy
      * + If a corresponding *fast bin* exists, try and find a chunk from there (and also opportunistically prefill the *tcache* with entries from the fast bin).
        + Otherwise, if a corresponding *small bin* exists, allocate from there (opportunistically prefilling the *tcache* as we go).
   2. Resolve all the deferred frees
      * + Otherwise “truly free” the entries in the fast-bins and move their consolidated chunks to the *unsorted* bin
        + Go through each entry in the *unsorted* bin. If it is suitable, stop. Otherwise, put the unsorted entry on its corresponding small/large bin as we go (possibly promoting small entries to the *tcache* as we go).
   3. Default back to the basic recycling strategy
      * + If the chunk size corresponds with a large bin, search the corresponding large bin now
   4. Create a new chunk from scratch
   5. If all else fails, return NULL

Free()

1. If the pointer is NULL, the C standard defines the behavior as “do nothing”.
2. Otherwise, convert the pointer back to a chunk by subtracting the size of the chunk metadata.
3. Perform a few sanity checks on the chunk, and abort if the sanity checks fail.
4. If the chunk fits into a *tcache* bin, store it there.
5. If the chunk has the *M* bit set, give it back to the operating system via *munmap*.
6. Otherwise we obtain the arena heap lock and then:
   * If the chunk fits into a fastbin, put it on the corresponding fastbin, and we’re done.
   * If the chunk is > 64KB, consolidate the fastbins immediately and put the resulting merged chunks on the unsorted bin.
   * Merge the chunk backwards and forwards with neighboring freed chunks in the small, large, and unsorted bins.
   * If the resulting chunk lies at the top of the heap, merge it into the top of the heap rather than storing it in a bin.
   * Otherwise store it in the *unsorted bin*. (*Malloc* will later do the work to put entries from the unsorted bin into the small or large bins).