Understanding glibc malloc 번역

Posted on October 8, 2018

원문 링크 (https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/)

복무 시작하고 공부를 잘 안하게 되다보니 무언가라도 해야 될 것 같아서 포너블 문제를 다시 풀어보려했습니다. 그 전에도 heap에 대한 이해가 부족함을 느꼈는데 안하다가 다시 하려니 너무 헷갈려서 malloc부터 다시 heap을 공부하기로 다짐했습니다. 그래도 예전에 한 번 훑어봤던 부분이기에 처음 공부했을 때 보다는 이해가 빠르네요

본문

나는 항상 heap 메모리에 매료되었다. 다음과 같은 질문

> 커널에서 힙 메모리를 얻는 방법은 무엇일까?
>
> 얼마나 효율적으로 메모리를 관리할까?
>
> 커널이나 라이브러리 또는 응용 프로그램 자체에서 관리할까?
>
> heap 메모리가 악용 될 수 있을까?

꽤 많은 시간이 걸리는 질문이다. 그러나 나는 최근에 그것을 이해하기 위한 시간을 가졌다. 그래서 여기에 내가 매료된 지식에 대해서 공유하려한다!! 거기에는 많은 메모리 할당자들이 사용 가능하다:

- dlmalloc - 일반적 목적의 할당자

- ptmalloc2 - glibc
- jemalloc - FreeBSD와 Firefox
- tcmalloc - Google
- libumem - Solaris
- …

모든 메모리 할당자는 그들이 빠르고, 규모가변적이고 메모리 효율적이라고 주장한다!! 그러나 모든 할당자들이 우리의 애플리케이션에 적합하다고 할 수 없다. 메모리 사용량이 많은 응용 프로그램의 성능은 메모리 할당자의 성능에 크게 좌우된다. 이 글에서는 'glibc malloc' 메모리 할당자에 대해서만 다룰 것이다. 앞으로 다른 메모리 할당자 또한 다루길 바라고 있다. 이 글에서 'glibc malloc'에 대해 더 잘 이해하기 위해 최신 소스코드를 링크(https://sourceware.org/ml/libc-alpha/2014-09/msg00088.html)할 것이다. 이제 glibc malloc을 시작하자!!

History: ptmalloc2 (http://www.malloc.de/en/)는 dlmalloc (http://g.oswego.edu/dl/html/malloc.html)로부터 나누어졌다. 나누어진 후, 스레딩 지원 기능이 추가되어 2006년에 릴리즈 되었다. 공식적인 릴리즈 후에 ptmalloc2는 glibc 소스 코드에 통합되었다. 통합이 완료된 후 코드 변경은 glibc malloc 소스 코드를 직접 수정하는 것이 되었다. 따라서 ptmalloc2와 glibc malloc 구현 간에 많은 변화가 있을 수 있다.

System Calls: 이 글 (https://sploitfun.wordpress.com/2015/02/11/syscalls-used-by-malloc/)에서 볼 수 있듯이 malloc은 내부적으로 brk (http://man7.org/linux/man-pages/man2/sbrk.2.html) 혹은 mmap (http://man7.org/linux/man-pages/man2/mmap.2.html)을 호출한다.

Threading: 리눅스 초기에는 dlmalloc이 기본 메모리 할당자로 사용되었다. 그러나 ptmalloc2의 스레딩 지원으로 인해 리눅스 기본 메모리 할당자가 되었다. 스레딩 지원은 메모리 할당자 및 응용 프로그램 성능 향상에 도움이 된다. dlmalloc에서 두 개의 스레드가 동시에 malloc을 호출할 때 freelist 데이터 구조는 사용 가능한 모든 스레드 간에 공유되기 때문에 오직 한 개의 스레드만 critical section(임계 영역)에 들어갈 수 있다. 따라서 멀티 스레드 응용 프로그램에서 메모리 할당에 시간을 소요하므로 성능이 저하된다. ptmalloc2에서 두 스레드가 동시에 malloc을 호출할 때 각각의 스레드는 분리된 힙 세그

먼트를 유지하고 힙을 유지하기 위한 freelist 데이터 구조 또한 분리되어 있기 때문에 즉각적으로 할당된다. 이러한 각 스레드의 유지를 위해 heap과 freelist 데이터 구조를 분리하는 행위를 **per thread arena**라고 부른다.

[*Example](*Example*):**

```
/* 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",getp
        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*:**

메인 스레드의 malloc 전: 아래의 출력에서 heap 세그먼트가 아직 없는 것과 thread1이 생성되지 않아 스레드 스택이 없는 것을 볼 수 있다.

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

메인 스레드의 malloc 이후: 아래의 출력에서 heap 세그먼트가 생성되었고 데이터 세그먼트(0804b000-0806c000) 바로 위에 있는 것을 볼 수 있다. 이것은 heap 메모리가 프로그램의 break location을 증가시키면서 생성됨을 보여준다 (ie) **brk** syscall을 사용). 또한 사용자가 1000 bytes만 요청했지만, heap 메모리의 사이즈는 132 KB가 생성되었다. 이러한 heap 메모리의 인접한 영역을 **arena**라고 부른다. 이 arena는 메인 스레드에 의해 생성되었기 때문에 **main arena**라고 부른다. 이후 할당 요청은 비어있는 공간(free space)이 없어질 때 까지 이 arena를 사용하며 유지된다. arena의 비어있는 공간이 없어졌을 때, 프로그램의 break location을 증가시켜서 자라날 수 있다.(top chunk의 크기를 늘려 여분의 공간을 포함할 수 있도록 조정한 후). 비슷하게 top chunk의 free space가 많으면 줄어들 수도 있다.

NOTE: Top chunk는 arena의 최상단 chunk이다. 이것에 대한 자세한 내용은 아래 “Top Chunk” section에서 보여준다.

```
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ ./mthrea
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/mthr
08048000-08049000 r-xp 00000000 08:01 539625     /home/sploitfu
08049000-0804a000 r--p 00000000 08:01 539625     /home/sploitfu
0804a000-0804b000 rw-p 00001000 08:01 539625     /home/sploitfu
0804b000-0806c000 rw-p 00000000 00:00 0          [heap]
b7e05000-b7e07000 rw-p 00000000 00:00 0
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$
```

메인 스레드에서 free 이후: 아래 출력에서 할당된 메모리 영역이 해제됐을 때, 운영체제에 즉각적으로 메모리가 반환되지 않음을 볼 수 있다. 할당된 메모리 영역(of size 1000 bytes)는 main arenas bin (glibc 말록에서 freelist 데이터구조는 bins으로써 참조)에 해제된 블록을 추가하고 ‘glibc malloc’ 라이브러리에 반환된다. 후에 사용자가 메모리를

요청하면 ‘glibc malloc’은 커널로부터 새로운 heap 메모리를 가져오는 대신에 bin의 free block을 찾는다. 오직 free block이 존재하지 않을 때, 커널로부터 메모리를 가져온다.

```
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ ./mthrea
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/mthr
08048000-08049000 r-xp 00000000 08:01 539625     /home/sploitfu
08049000-0804a000 r--p 00000000 08:01 539625     /home/sploitfu
0804a000-0804b000 rw-p 00001000 08:01 539625     /home/sploitfu
0804b000-0806c000 rw-p 00000000 00:00 0          [heap]
b7e05000-b7e07000 rw-p 00000000 00:00 0
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$
```

thread1에서 malloc 전: 아래의 출력에서 thread1의 heap 영역이 없으나 thread1의 스레드 스택이 생성됐음을 볼 수 있다.

```
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ ./mthrea
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 /pro
08048000-08049000 r-xp 00000000 08:01 539625     /home/sploitfu
08049000-0804a000 r--p 00000000 08:01 539625     /home/sploitfu
0804a000-0804b000 rw-p 00001000 08:01 539625     /home/sploitfu
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$
```

thread1의 malloc 이후 : 아래 출력에서 thread1의 heap 영역이 생성되었음을 볼 수 있다. 그리고 메모리mapping된 세그먼트 일부 (13 KB의 크기의 b7500000-b7521000)에 위치하고 sbrk를 사용하는 main thread와 다르게 **mmap** syscall을 사용하여 heap memory를 생성함을 보여준다. 다시 여기서, 사용자가 오직 1000 bytes를 요청했음에도 불구하고, 1 MB 크기의 heap 메모리가 프로세스 주소 공간에 mapped 되어 있다. 이 1 MB 중에서, 오직 132KB가 read-write 권한이 설정되었고, 이 thread를 위한 heap 메모리가 되었다. 이 인접한 memory 영역(132 KB)을 **thread arena**라고 부른다.

NOTE: 사용자가 128 KB(malloc(132*1024))보다 큰 사이즈를 요청할 경우와 arena에 사용자 요청을 만족할만한 충분한 공간이 없을 경우, main area 혹은 thread arena로부터의 요청 여부에 관계 없이 메모리는 sbrk를 사용하는 것이 아니라 mmap 시스템 콜을 사용해서 할당한다.

```
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ ./mthrea
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 /pro
08048000-08049000 r-xp 00000000 08:01 539625     /home/sploitfu
08049000-0804a000 r--p 00000000 08:01 539625     /home/sploitfu
0804a000-0804b000 rw-p 00001000 08:01 539625     /home/sploitfu
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$
```

thread1 free 호출 이후 : 해제한 할당된 메모리 영억은 운영 체제에게 heap 메모리를 반환하지 않는다. 대신에 할당된 메모리 영역(1000 bytes)은 thread arenas bin의 freed block에 추가하고 ‘glibc malloc’에 반환된다.

```
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ ./mthrea
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 /pro
08048000-08049000 r-xp 00000000 08:01 539625     /home/sploitfu
08049000-0804a000 r--p 00000000 08:01 539625     /home/sploitfu
0804a000-0804b000 rw-p 00001000 08:01 539625     /home/sploitfu
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:

arena의 수: 위 예시에서, main thread는 main arena를 포함하고 thread 1은 자신의 thread arena를 포함하는 것을 보았다. 그러면 스레드의 수와 관계 없이 threads와 arena 사이의 일대일 매핑이 가능할까? 확실하게 아니다. 미친 응용 프로그램은 코어 수보다 많은 스레드를 포함할 수 있지만, 이러한 경우, 하나의 arena 당 thread를 가져서 bit가 크게 낭비된다. 이러한 이유로, 응용 프로그램의 arena는 현재 시스템의 코어 수에 기반하여 제한된다.

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

Multiple Arena:

예시: 1개의 코어를 가진 32비트 시스템의 멀티스레드 응용 프로그램(4 스레드 - 메인 스레드 + 3 유저 스레드)을 실행시켜보자. 여기서 스레드가 없는 경우(4) > 2*코어가 없는 경우(2)를 볼 수 있다. 이러한 경우, 'glibc malloc'은 multiple arenas가 사용 가능한 모든 스레드에서 공유되도록 한다. 그런데 어떻게 공유할까?

- 메인 스레드의 경우, 처음 malloc을 호출하면 이미 생성된 main arena는 어떠한 충돌 없이 사용된다.
- thread 1과 thread2가 처음 malloc을 호출하면, 이 스레드들을 위해 새로운 arena가 만들어지고 어떠한 충돌없이 사용된다. 여기까지 스레드와 arena는 일대일 매핑이 된다.
- thread 3이 처음으로 malloc을 호출하면, arena 수의 제한이 계산된다. 여기서 arena 제한을 넘어서, 존재하는 arena(main arena, Arena 1, Arena 2)를 재사용(reuse)한다.
- Reuse:
    - 사용가능한 arena들이 한 번의 루프를 도는 동안, arena에 lock을 시도한다.
    - 만약 lock이 성공적으로 된 경우(main arena가 성공적으로 lock된 경우), 유저에게 arena를 반환한다.
    - 만약 어떠한 arena도 해제되지 않은 경우, 다음 라인에 arena가 block된다.
- thread3이 malloc을 두 번째 호출한 경우, malloc은 마지막으로 접근한 아레나를 사용하도록 시도한다(main arena). 만약 main arena가 사용 가능하지 않은 경우, 메인 아레나가 free될 때 까지 thread 3이 차단된다. 따라서 메인 아레나는 메인 스레드와 스레드 3 사이에서 공유된다.

Multiple Heaps:

'glibc malloc' 소스 코드에서 아래의 주요 세 가지 데이터 구조체를 확인할 수 있다:

heap_info - Heap Header - 단일 스레드 arena는 여러 개의 힙을 가질 수 있다. 각각의 힙은 자신의 헤더를 가진다. 왜 여러 개의 힙이 필요한가? 모든 스레드 arena는 오직 한 개의 힙만을 가지고 시작하지만, 힙 영역의 공간이 부족해지면 새로운 힙(인접하지 않은 구역)이 이 arena에서 할당되어야 한다.

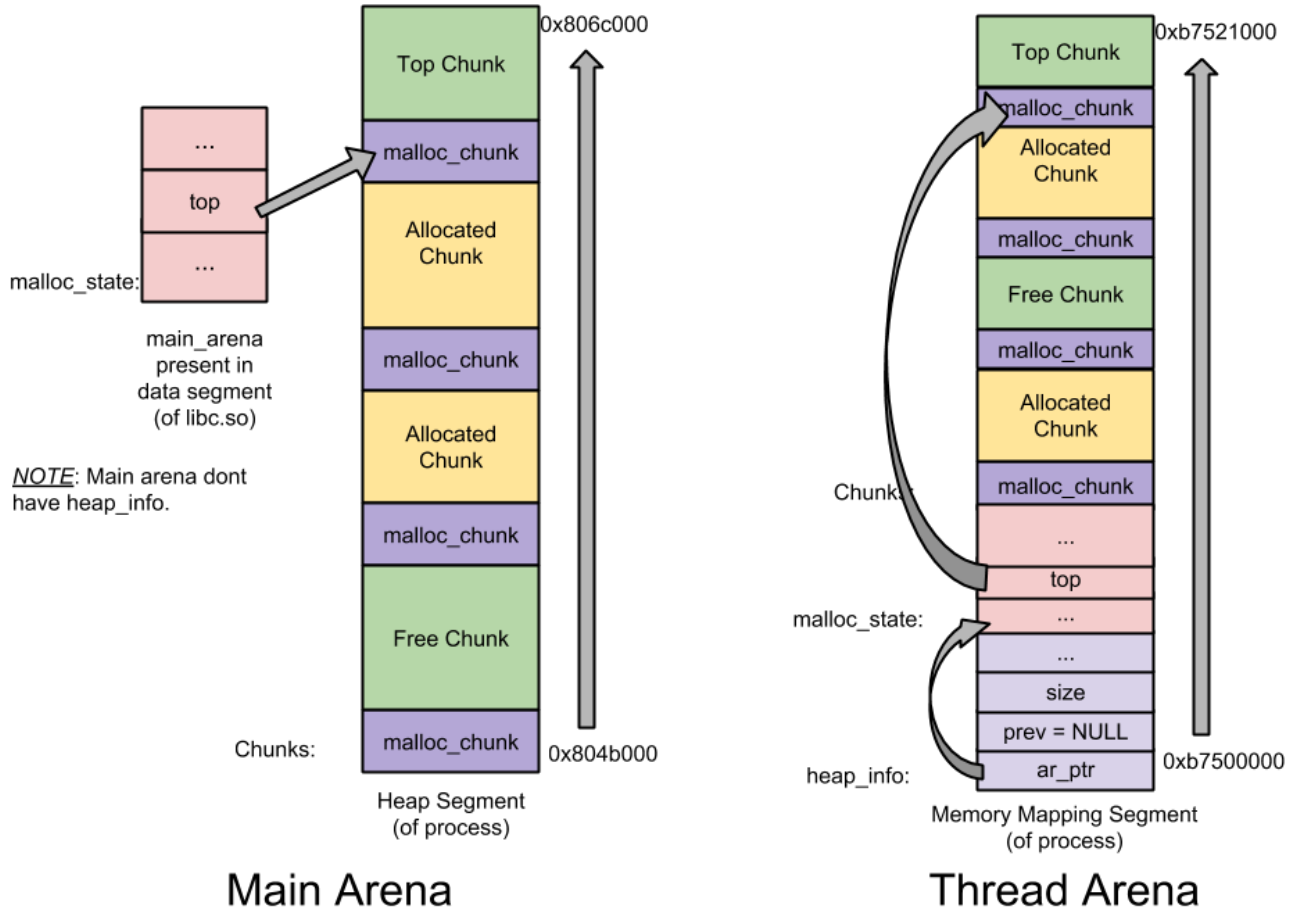
malloc_state - Arena Header - 단일 스레드 arena는 여러 개의 힙을 가질 수 있지만 모든 힙에 대해서 오직 한 개의 arena 헤더가 존재한다. Arena 헤더는 bins, top chunk, last remainder chunk 등에 관한 정보를 가진다.

malloc_chunk - Chunk Header - 힙은 사용자의 요청에 기반하여 많은 chunk로 분리된다. 이러한 chunk들 각각은 자신의 chunk 헤더를 가진다.

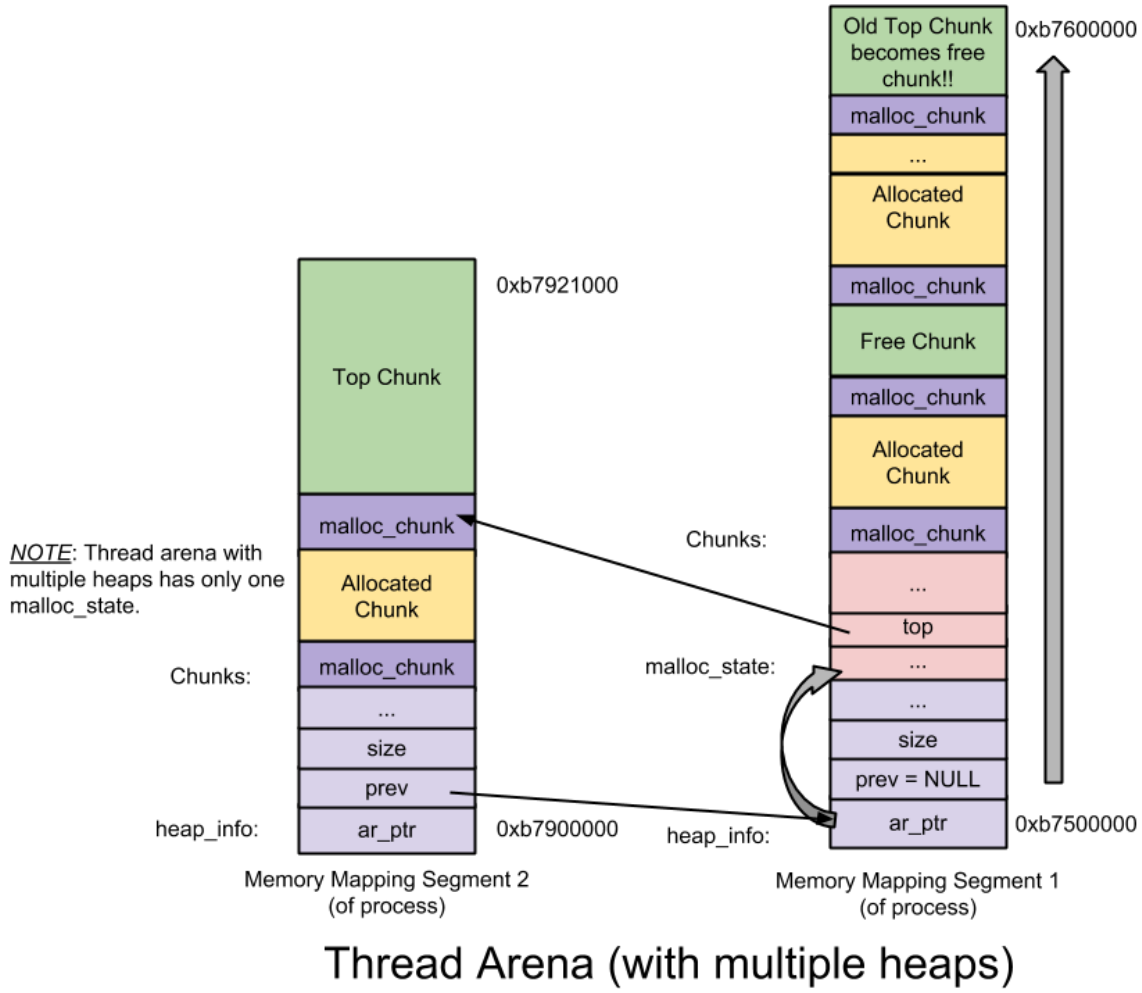
NOTE:

- 메인 arena는 여러 개의 힙을 가질 수 없고 heap_info 구조체를 가질 수 없다. 메인 arena의 공간이 부족할 때, sbrk 힙 영역은 메모리가 매핑된 영역까지 확장(인접한 영역)된다.
- 스레드 arena와 다르게, 메인 arena의 arena 헤더는 sbrk 힙 영역의 일부가 아니다. 그것은 전역 변수이고 libc.so의 데이터 영역에서 찾을 수 있다.

main arena와 thread arena를 그림으로 나타낸 경우(단일 heap 세그먼트):



thread arena를 그림으로 나타낸 경우(여러 heap 세그먼트):

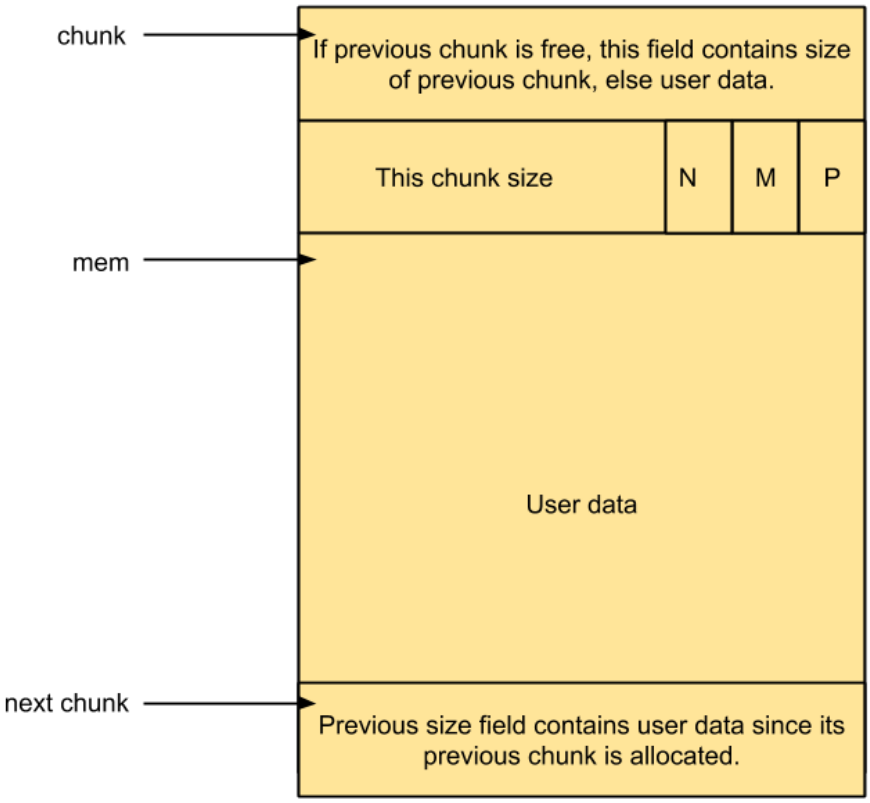


Thread Arena (with multiple heaps)

Chunk: 청크(Chunk)는 heap 세그먼트 안에서 찾을 수 있고 아래의 type 중 하나이다:

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

Allocated chunk(할당된 청크):



Allocated Chunk

prev_size: 이전 chunk가 해제되었을 경우, 이 필드는 이전 chunk의 크기를 가진다. 이전 chunk가 할당되어있으면, 이 필드는 이전 chunk의 user data를 가진다.

size: 이 필드는 할당된 chunk의 크기를 가진다. 마지막 3비트는 flag 정보를 가지고 있다.

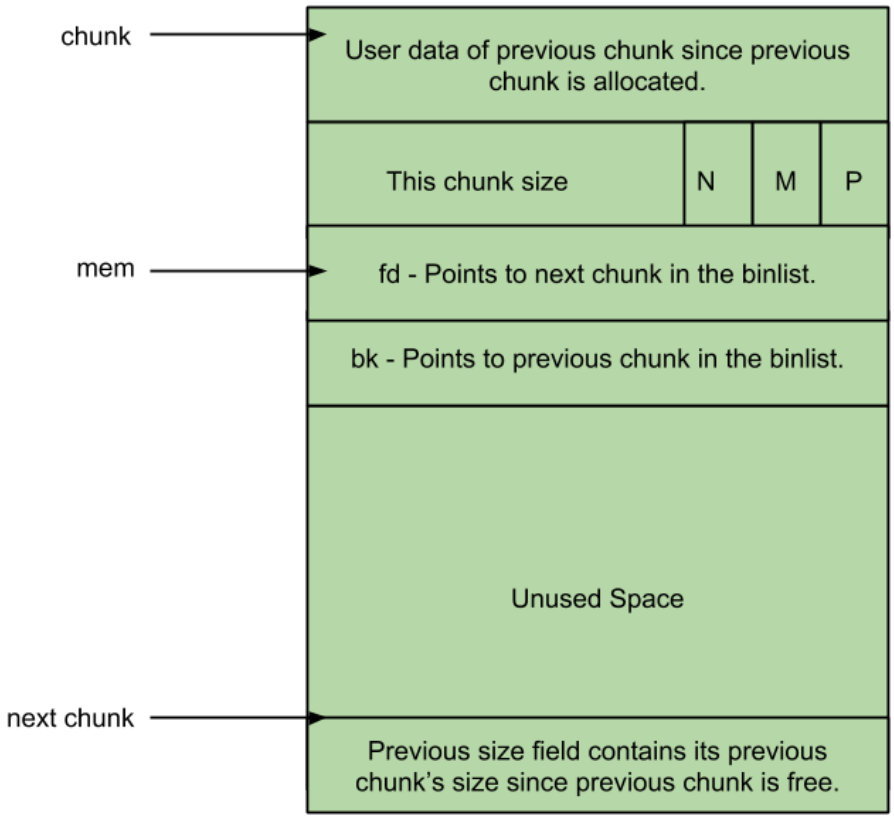
- PREV_INUSE (P) - 이 비트는 이전 chunk가 할당되었을 때 세팅된다.
- IS_MMAPPED (M) - 이 비트는 chunk가 mmap을 통해 할당되었을 때 세팅된다.
- NON_MAIN_ARENA (N) - 이 비트는 현재 chunk가 thread arena에 속하는 경우 세팅된다.

NOTE:

- malloc_chunk의 다른 필드(fd,bk 같은)는 할당된 청크에 사용되지 않는다. 따라서 할당된 청크에는 유저 데이터가 저장되어 있다.

- 사용자가 요청한 크기는 malloc_chunk 저장과 정렬을 위해 약간의 공간이 필요하기 때문에 사용 가능한 크기(내부를 나타내는 크기)로 변환된다. 사용 가능한 크기의 마지막 3비트의 변환은 플래그 정보를 저장하는데 사용하기 때문에 세트되지 않는다.

Free Chunk:



Free Chunk

prev_size: 두 개의 프리 청크는 함께 인접해있을 수 없다. 두 청크가 모두 비어 있으면 하나의 단일 청크로 합쳐진다. 따라서 해제된 청크에 항상 이전 청크가 할당되므로 prev_size에는 이전 청크의 사용자 데이터가 포함된다.

size: 이 필드는 free 청크의 크기를 포함한다.

fd: Forward pointer - 같은 bin에서 다음 청크를 가리킨다.(실제 메모리에 있는 다음 청크가 아니다.)

bk: Backward pointer - 같은 bin에서 이전 청크를 가리킨다.(실제 메모리에 있는 이전 청크가 아니다.)

Bins: Bins은 freelist 데이터 구조이다. 그들은 free 청크를 잡아두는데 사용된다. chunk 크기에 따라 다른 bins들이 사용된다:

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

데이터구조는 이러한 bins을 보관하는데 사용된다:

fastbinsY: 이 배열은 fast bins을 가진다.

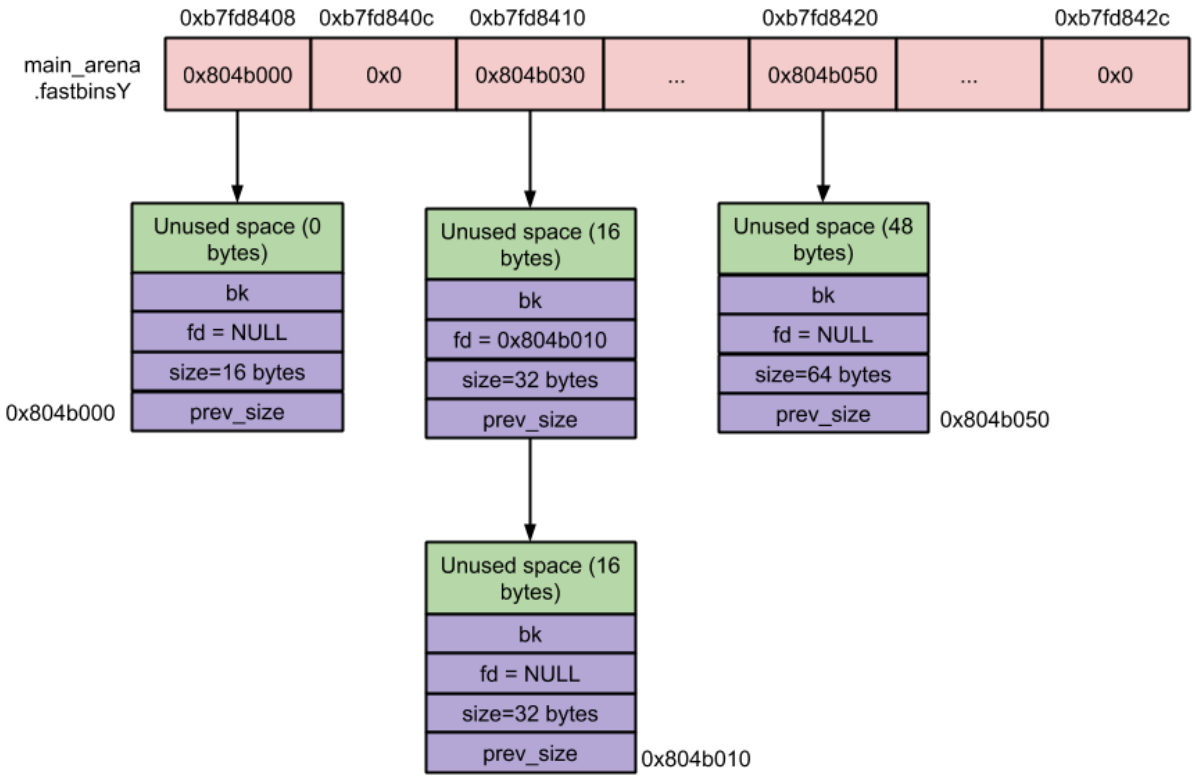
bins: 이 배열은 unsorted, small 그리고 large bins을 가진다. 총 126개의 bin이 있고 다음과 같이 나뉜다.

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

Fast Bin: 청크의 크기가 16~80 바이트인 경우 fast chunk라고 부른다. fast chunk를 가진 bins은 fast bins이라고 불린다. 모든 빈 중에서 fast bins은 메모리 할당과 반납이 빠르다.

- bins의 갯수 - 10
    - 각 fast bin은 free chunk로 구성된 단일 연결 리스트(bin list)를 가진다. 단일 연결 리스트는 리스트의 중간에서 fast bins chunk를 삭제할 수 없어서 사용된다. 추가와 제거는 리스트의 앞과 끝에서 발생한다. - LIFO.
- Chunk 크기 - 8 bytes 씩
    - Fast bins은 8 바이트의 크기를 가지는 chunk의 bin list를 가진다. ie) 첫 fast bin (index 0)은 16 bytes 크기 청크의 bin list를 가지고, 두 번째 fast

bin (index 1)은 24 byte 크기 청크의 bin list를 가지고..
    - fast bin 내부의 청크는 동일한 크기이다.
- malloc 초기화 동안, fast bin의 최대 크기는 64 (!80) bytes로 세트된다. 따라서 기본 청크의 크기가 16~64인 경우 fast chunk에 분류된다.
- 병합 없음 - 해제된 두 청크는 각각 인접할 수 있고, 단일 free chunk로 결합되지 않는다. 병합이 없으면 외부 단편화를 가져오지만 해제 속도는 빨라진다
- malloc(fast chunk) -
    - 처음에는 fast bin 최대 크기 및 fast bin indices가 비어있어서 사용자가 fast chunk를 요청하면 fast bin code 대신에 small bin code가 서비스할 수 있다.
    - 나중에 비어있지 않게 되면, fast bin index는 해당 binslit를 검색 하기 위해 계산된다.
    - 위에서 검색된 binlist의 첫 번째 청크는 삭제되고 사용자에게 반환된다.
- free(fast chunk) -
    - fast bin index는 해당 binlist를 검색 하기 위해 계산한다.
    - free chunk는 위에서 검색된 binlist의 앞 쪽에 추가된다.

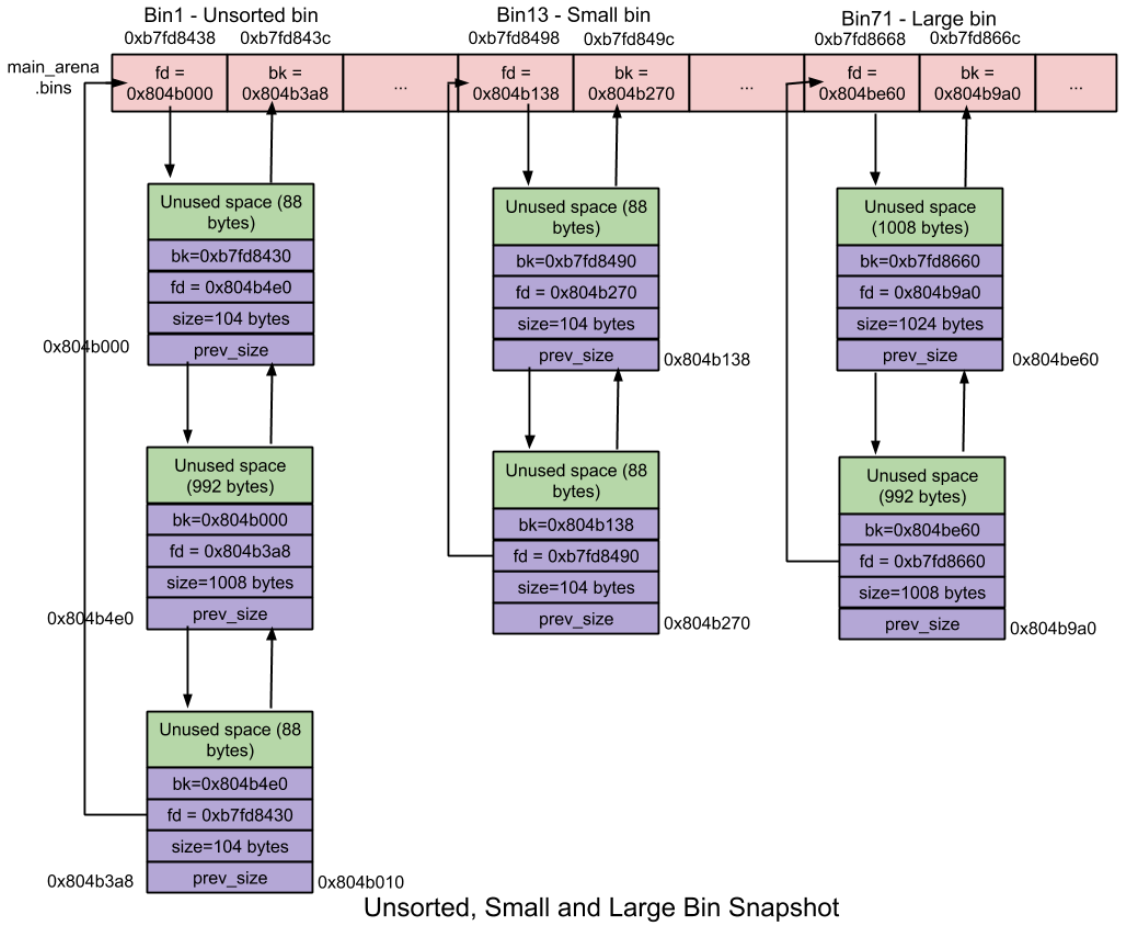


Fast Bin Snapshot

Unsorted Bin: small 혹은 large chunk가 각각에 bins에서 추가되지 않고 해제되면 unsorted bin에 추가된다.

이 방법은 'glibc malloc'에게 최근 해제된 청크를 재사용할 수 있는 기회를 제공한다. 따라서 적절한 bin을 찾는데 걸리는 시간이 적어지기 때문에 메모리 할당 및 해제 속도가 약간 빨라진다(unsorted bin이기 때문).

- Bins의 갯수 - 1
  - unsorted bin은 free chunk의 circular double linked list (binlist)를 가진다.
- Chunk 크기 - 크기의 제한이 없으며 모든 사이즈의 청크가 이 bin에 속한다.



Unsorted, Small and Large Bin Snapshot

Small Bin: 512 bytes 아래의 크기를 가진 청크를 small chunk라고 한다. samll chunk를 가진 bins을 small bins이라고 한다. Small bins은 메모리 할당과 해제에서 large bins보다 빠르다.(fast bins보다는 느리다.)

- Bins의 갯수 - 62개
    - 각각의 small bin은 free chunk의 circular double linked list(binlist라 불린다)를 가진다. Double linked list는 list의 중간에서 small bins 청크를 제거할 수 있어서 사용된다. 노드의 추가는 앞에서 발생하고 제거는 뒤에서 발생한다 - FIFO.
- Chunk 크기 - 각 8 bytes
    - Small bin은 8 바이트의 크기를 가지는 chunk의 bin list를 가진다. ie) 첫 번째 Small bin (Bin 2)는 16 바이트 크기 청크의 binlist를 가지고, 두 번째 Small bin (Bin 3)는 24 바이트 크기의 청크의 bin list를 가지고..
    - Small bin 내부의 청크는 동일한 크기여서 따로 정렬할 필요가 없다.
- 병합 - 해제된 두 청크는 각각 인접할 수 없어서 단일 free chunk로 결합된다. 병합은 외부 단편화를 줄여주지만 해제 속도를 느리게한다.
- malloc(small chunk) -
    - 처음에 small bins은 NULL이어서 사용자가 small bin을 요청하면 small bin code 대신에 unsorted bin code가 서비스할 수 있다.
    - 또한 malloc에 대한 첫 호출 중 malloc_state에 있는 small bin과 large bin이 초기화 된다. ie) bins은 비어있는 자기 자신을 가리킨다.
    - 나중에 small bin이 비어있지 않을 때, 해당 binlist의 마지막 청크는 제거된 후 사용자에게 반환된다.
- free(small chunk) -
    - 이 청크를 해제하는 동안 이전 혹은 다음 청크가 해제되어있는지 체크하여 만약 해제되었을 경우 병합한다. ie) 해당 청크를 각각의 연결 리스트에서 연결 해제한 다음 unsorted bins의 시작 부분에 새롭게 합병된 청크를 추가한다.

Large Bin: 512보다 크거나 같은 크기의 청크들을 large chunk라고 한다. large chunk를 가지는 Bins를 large bins이라고 부른다. Large bin은 메모리 할당과 해제가 small bins보다 느리다.

- Bins의 갯수 - 63
    - 각각의 large bin은 해제된 청크의 circular double linked list를 가진다. 이중 연결 리스트는 어느 위치(앞,뒤,중간)에서든 large bins 청크를 추가하거나 삭제할 수 있어서 사용된다.
    - 63개의 bins:
        - 32개의 bins는 각 64바이트의 크기를 가진 청크의 binlist를 가진다. ie) First large bin (Bin 65)는 512 바이트에서 568바이트 크기의 청크의

binlist를 가진다, 두 번째 large bin (Bin 66)은 576 바이트에서 632 바이트 크기 청크의 binlist를 가진다…
        - 16개의 bins는 각 512 바이트 크기를 가진 청크의 binlist를 가진다.
        - 8개의 bins는 각 4,096 바이트 크기를 가진 청크의 binlist를 가진다.
        - 4개의 bins는 각 32,768 바이트 크기를 가진 청크의 binlist를 가진다.
        - 2개의 bins는 각 262,144 바이트 크기를 가진 청크의 binlist를 가진다.
        - 1개의 bin은 남은 크기를 가지는 청크를 가진다.
    - Small bin과 다르게 large bin 내부의 청크는 같은 크기를 가지지 않는다. 따라서 그들은 내림차순으로 정렬된다. 가장 큰 청크는 binlist의 가장 앞 쪽에 저장되고 가장 작은 청크는 가장 뒷 쪽에 저장된다.
- 병합 - 해제된 두 청크는 각각 인접해 있을 수 없어서 단일 free 청크로 결합된다.
- malloc(large chunk) -
    - 처음에 모든 large bins은 NULL이기 때문에 사용자가 large chunk를 요청하더라도 large bin code 대신에 next largest bin code가 서비스 할 수 있다.
    - 또한 malloc에 대한 첫 호출 중 malloc_state에 있는 small bin과 large bin이 초기화 된다. ie) bins은 비어있는 자기 자신을 가리킨다.
    - 나중에 large bin이 비어있지 않을 때, binlist의 가장 큰 크기의 청크가 사용자가 요청한 사이즈보다 크면 binlist는 처음부터 끝까지 탐색하여 사용자가 요청한 사이즈에 가깝거나 동일한 크기의 적합한 청크를 찾는다. 찾게되면 해당 청크는 두 개의 청크로 분리된다.
        - User chunk (사용자 요청한 크기) - 사용자에게 반환된다.
        - Remainder chunk (남은 크기) - unsorted bin에 추가된다.
    - 만약 binlist의 가장 큰 크기의 청크가 사용자가 요청한 크기보다 작을 때, 비어있지 않은 다음 largest bin을 사용하여 사용자 요청에 응답한다. 다음 largest bin code는 비어있지 않은 다음 largest bin을 찾기 위해 binmaps을 스캔하고 만약 찾을 경우 binlist에서 적합한 청크가 탐색되고, 분리되어 유저에게 반환된다. 만약 찾지 못할 경우 top chunk를 사용하여 사용자의 요청에 응한다.
- free(large chunk) - 그것의 절차는 small chunk의 free와 같다.

Top Chunk: arena의 가장 꼭대기에 위치해있는 청크를 top chunk라 한다. 어떠한 bin에도 속해있찌 않는다. Top chunk는 bins 중에 해제된 블록이 없는 경우 사용자 요청에 응하기 위해 사용된다. 만약 top chunk의 크기가 유저가 요청한 사이즈보다 클 경우 top chunk는 두 개로 분리된다:

- User chunk(사용자가 요청한 크기)
- Remainder chunk(남은 크기)

remainder chunk는 새로운 top이 된다. 만약 top chunk의 크기가 유저가 요청한 사이즈보다 작을 경우, top chunk는 sbrk(main arena) 혹은 mmap(thread arena) syscall을 사용하여 확장된다.

Last Remainder Chunk: Remainder는 가장 최근 요청에 의해 분리된 작은 조각들이다. Last remainder chunk는 지역 참조성을 증가시키는데 도움이 된다. ie)

그러나 arena에서 많은 사용 가능한 청크들 중 어떠한 청크가 last remainder chunk가 되기에 적합할까?

사용자가 small chunk에 요청했을 때, small bin과 unsorted bin으로 제공할 수 없는 경우, binmaps는 비어있지 않은 다음 largest bin을 찾기 위해 스캔된다. 앞서 말한대로, 다음 largest bin을 찾고 그것은 2개로 분리되고 user chunk는 사용자에게 반환되고 remainder chunk는 unsorted bin에 추가된다. 그 외에도 새로운 last remainder chunk가 된다..

어떻게 지역 참조성을 달성하는가?

이제 사용자가 작은 청크를 요청하고 last remainder chunk가 unsorted bins의 유일한 청크인 경우 last remainder chunk는 두 개로 분리되고, user chunk는 사용자에게 반환되고 remainder chunk는 unsorted bin에 추가된다. 그 외에도 그것은 새로운 last remainder chunk가 된다. 따라서 후속 메모리 할당은 결국 옆에 할당되는 것으로 끝난다.

Tags: system (/tags#system) heap (/tags#heap) translate (/tags#translate)

(ht.. (ht..

← PREVIOUS POST (/2017/08/19/PWNABLE-TW-START/)

NEXT POST → (/2019/04/06/CPP-CIN-ERROR/)

ALSO ON IDIOTH

 (https://github.com/idioth)  (mailto:dbsghdrhks78@gmail.com)