2025/01/09, 2024-W Heap Study

Heap 소개

이 정 민(neutrinox4b1@gmail.com)

세종대학교 정보보호학과

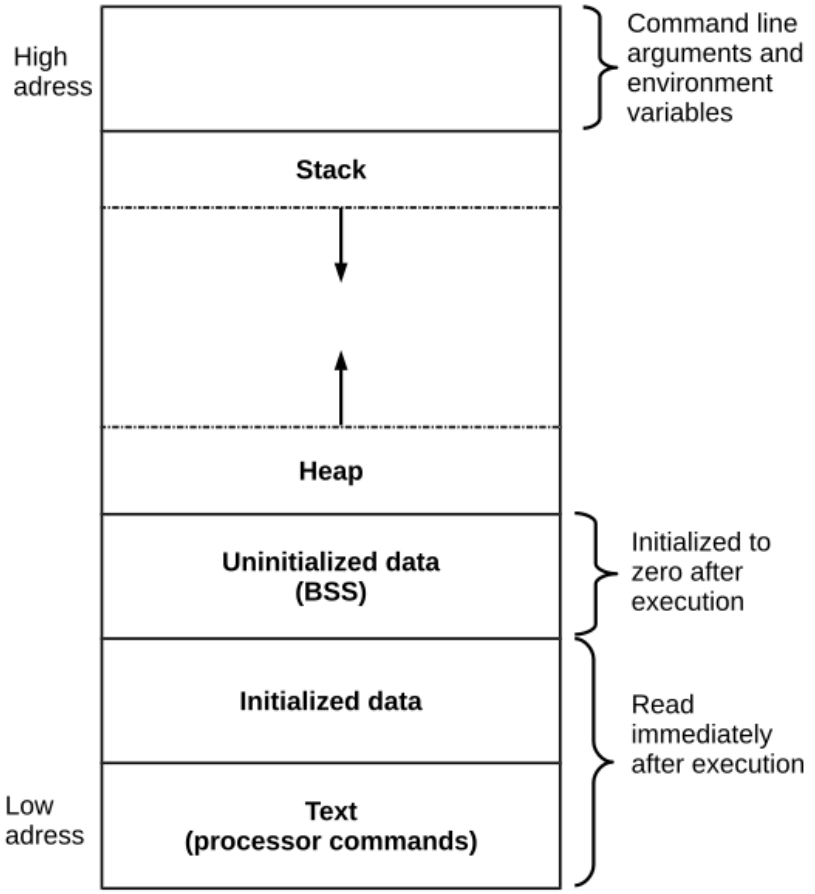
목 차

- Heap
- Memory Allocator
- Chunk & bin
- ~~Use After Free(UAF)~~

Heap

- 정의
  - 동적으로 할당된 메모리를 저장하는 전역 데이터 구조

- 특징
  - Stack보다 느림
  - 런타임 시간에 요청, 해제될 수 있음
  - 전역적 특성을 가짐
  - free()가 호출될 때까지 스코프를 가짐



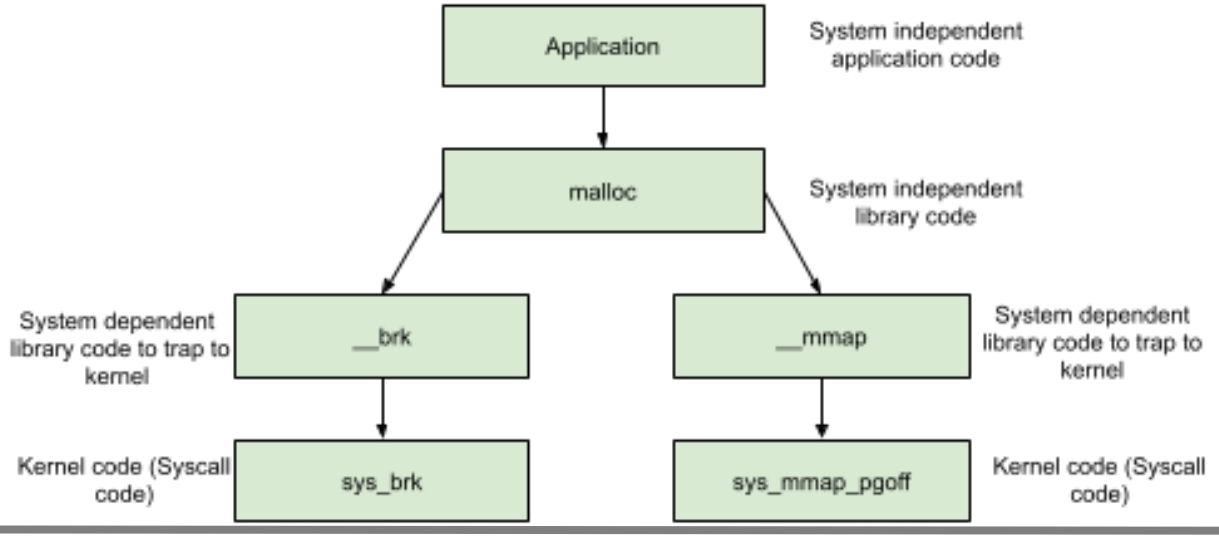
Heap

- 사용 예시
  - malloc(size_t n)
    - n바이트의 새로 할당된 chunk에 대한 포인터를 반환함
  - free(void *p)
    - p가 가리키는 메모리 chunk를 해제함

```c
// Dynamically allocate 10 bytes
char *buffer = (char *)malloc(10);

strcpy(buffer, "hello");
printf("%s\n", buffer); // prints "hello"

// Frees/unallocates the dynamic memory allocated earlier
free(buffer);
```

Heap

- malloc()의 system call
  - malloc은 내부적으로 brk() 또는 mmap()을 호출하여 메모리를 할당함
    - brk()는 비교적 작은 크기를 할당하는데에 사용됨
    - mmap()은 비교적 큰 크기를 할당하는데에 사용됨
    - MMAP_THRESHOLD에 따라 바뀜 (Default: 128KB)
    - 이렇듯, 동적으로 할당된 메모리를 관리하기 위해 Memory Allocator을 사용함



Memory Allocator

- 정의
  - 시스템에 동적으로 메모리 할당을 요구할 때, 해당 작업을 처리하며 힙 영역을 관리하는 모듈

- 분류
  - Explicit Allocator
    - 개발자가 공간 할당/해제를 관리함
    - e.g., C언어의 malloc과 free
  - Implicit Allocator
    - 개발자가 할당만 담당, free는 내부적으로 처리함
    - e.g., Java의 Garbage Collection(GC)

Memory Allocator

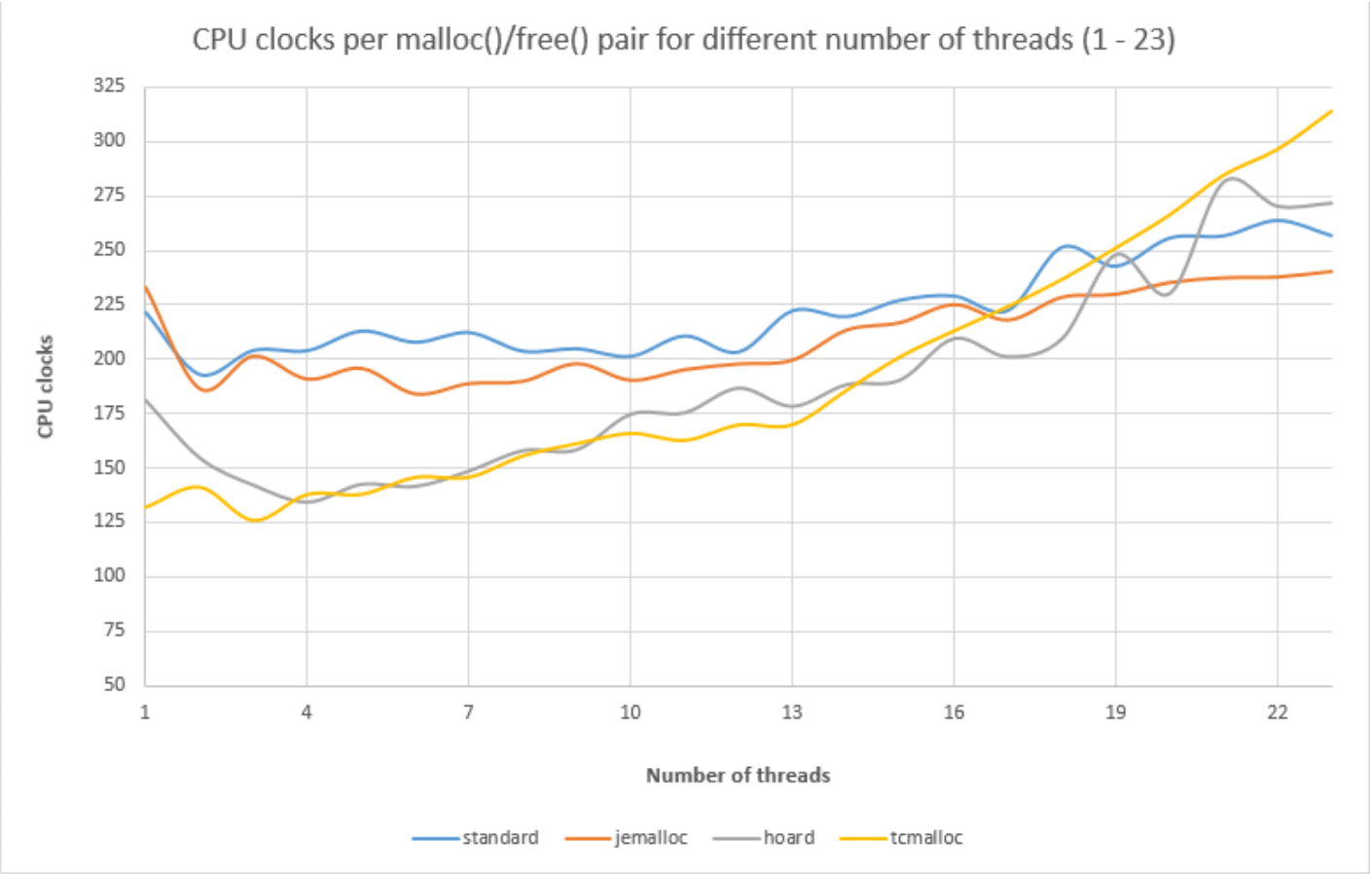
- Explicit Allocator
  - dlmalloc
    - 리눅스 초창기 사용된 기본 메모리 할당자
    - 동일 시간에 2개 스레드가 malloc 호출 시, 하나의 스레드만 임계영역에 들어갈 수 있어 다중 스레드에서 성능저하 발생
  - ptmalloc2
    - dlmalloc에서 스레딩 지원 기능을 추가함
  - jemalloc
    - 페이스북, 파이어폭스에서 주로 사용됨
    - 단편화 방지 및 동시 확장성 강조
  - tcmalloc
    - Google에서 개발하여, 크롬 및 다수 프로젝트에서 사용됨
    - 멀티스레드 환경에서 메모리 풀 사용 속도를 개선하기 위한 목적으로 설계됨

Memory Allocator

- dlmalloc과 ptmalloc2
  - ptmalloc2는 dlmalloc에서 fork됨
  - Doug Lea에 의해 dlmalloc이 개발되었음
    - https://gee.cs.oswego.edu/dl/html/malloc.html (Doug Lea의 글 참고)
    - https://neutrinox4b1.tistory.com/52 (번역본)
  - dlmalloc에서는 모든 스레드가 freelist data structure을 공유하여, 병목현상이 발생
  - ptmalloc2에서는 각 스레드가 별도의 힙 세그먼트를 유지하고, freelist data structure도 별도로 가지므로, 메모리가 즉시 할당됨
    - 이러한 동작을 **per thread arena**라고 함
  - glibc에서 freelist data structure는 bin으로 불림

Memory Allocator

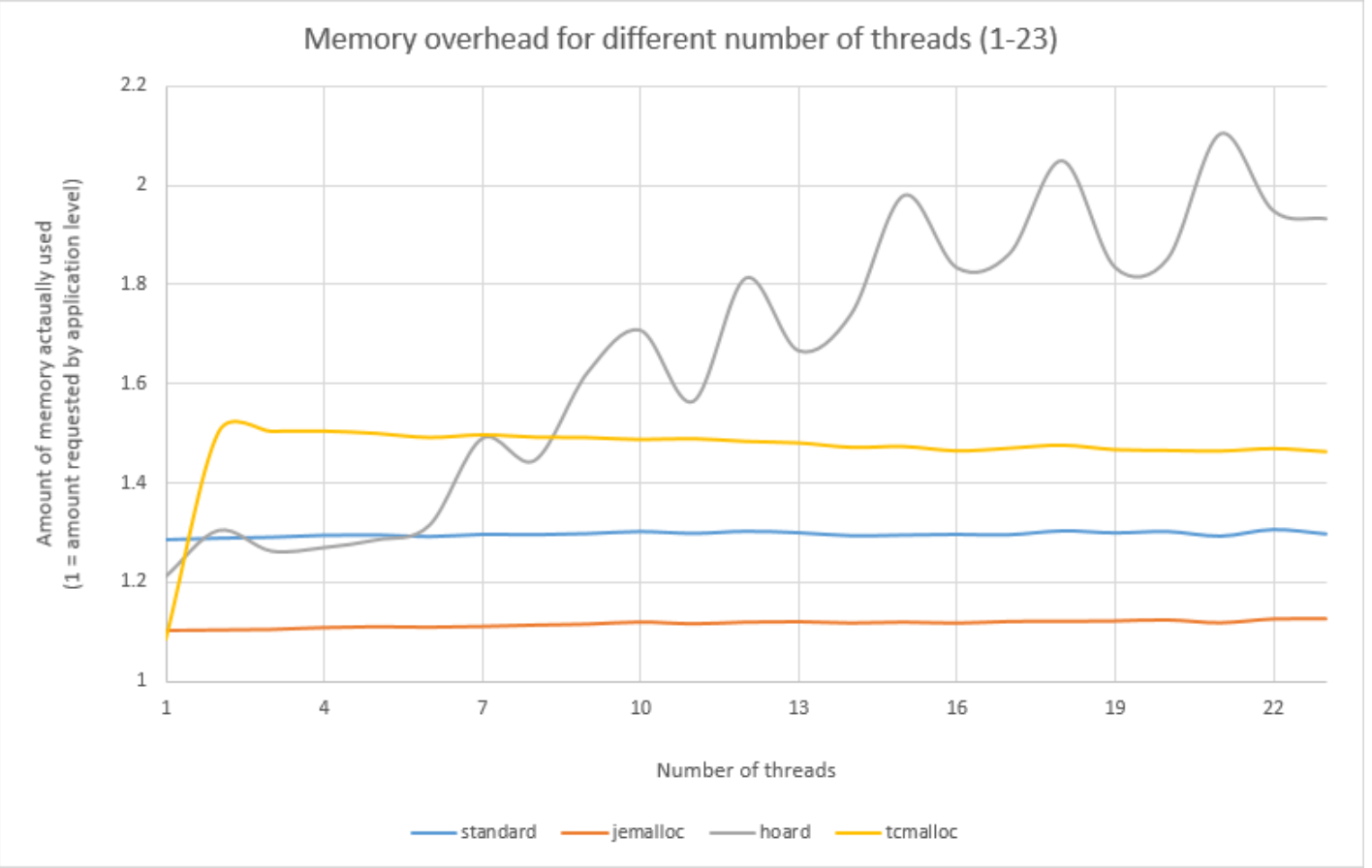
- malloc 벤치마킹 – 성능
  - 단일 스레드에서는 tcmalloc이 ptmalloc2보다 우세, 더 많은 수의 스레드에서는 ptmalloc2가 우세함



http://ithare.com/testing-memc
tmalloc2-tcmalloc-hoard-jemall
-to-simulate-real-world-loads/

Memory Allocator

- malloc 벤치마킹 – 메모리 오버헤드
  - jemalloc이 가장 효율적인 메모리 오버헤드를 가짐
  - ptmalloc2는 두 번째로 우수함
  - tcmalloc은 스레드 수 증가에 따라 뒤쳐졌음



http://ithare.com/testing-memc
tmalloc2-tcmalloc-hoard-jemall
-to-simulate-real-world-loads/

Chunk & bin

- ptmalloc의 객체
  - ptmalloc2에서는 chunk, bin, tcache, arena를 주요 객체로 사용함
    - glibc의 malloc은 청크 지향적(chunk-oriented)임
  - ptmalloc2, glibc 2.27(18.04 LTS bionic), x86-64을 기준으로 설명할 것임

Chunk & bin

- 정의
  - malloc()이 호출됨으로써 할당 받는 영역

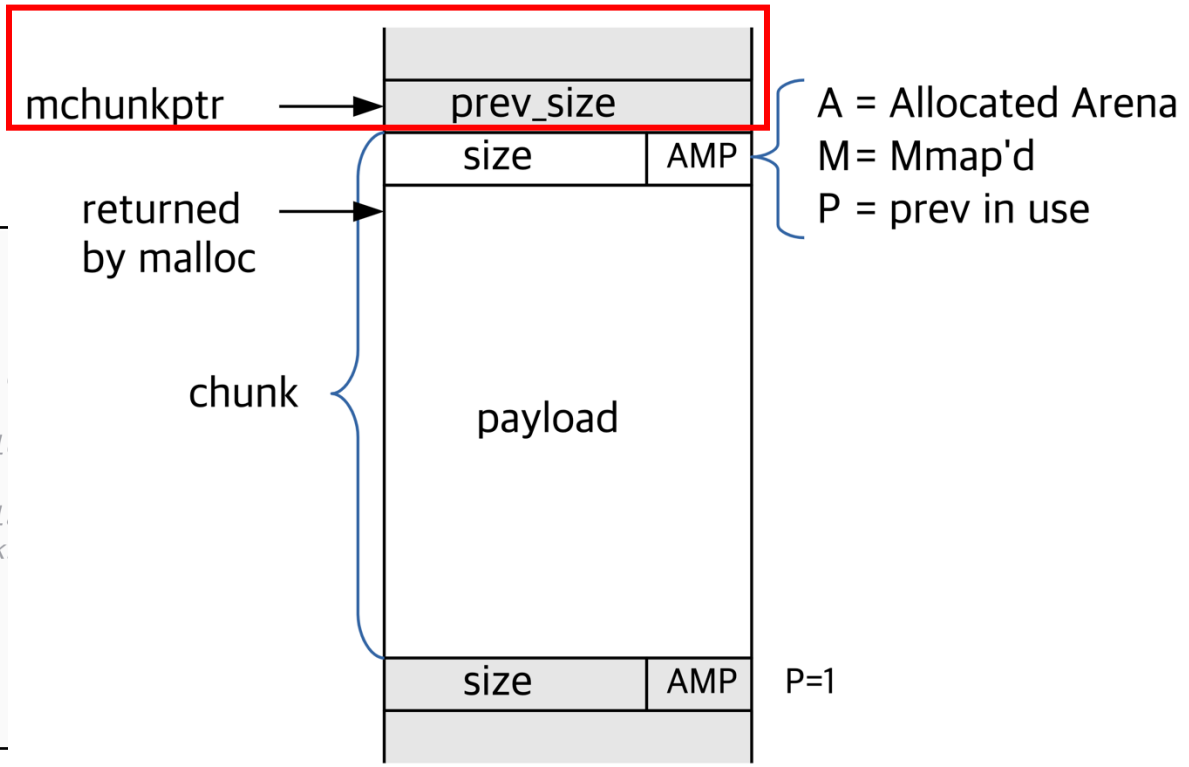
```
 1 struct malloc_chunk {
 2   INTERNAL_SIZE_T      mchunk_prev_size;  /* Size of previous chunk, if it is free. */
 3   INTERNAL_SIZE_T      mchunk_size;       /* Size in bytes, including overhead. */
 4   struct malloc_chunk* fd;                /* double links -- used only if this chunk is free.
 5 */struct malloc_chunk* bk;
 6   /* Only used for large blocks: pointer to next larger size.  */
 7   struct malloc_chunk* fd_nextsize; /* double links -- used only if this chunk is free. */
 8   struct malloc_chunk* bk_nextsize;
 9 };
10
11 typedef struct malloc_chunk* mchunkptr;
```

- 분류
  - 할당된 청크(Allocated(In-use) chunk)
  - 해제된 청크(Free chunk)

Chunk & bin

- Allocated Chunk
  - mchunkptr
    - 이전 chunk의 마지막 워드를 가리킴
  - prev_size
    - 이전 chunk가 free인 경우, 이전 chunk의 크기를 가짐
    - 이전 chunk가 allocated인 경우, 이전 청크의 사용자 데이터가 들어있음

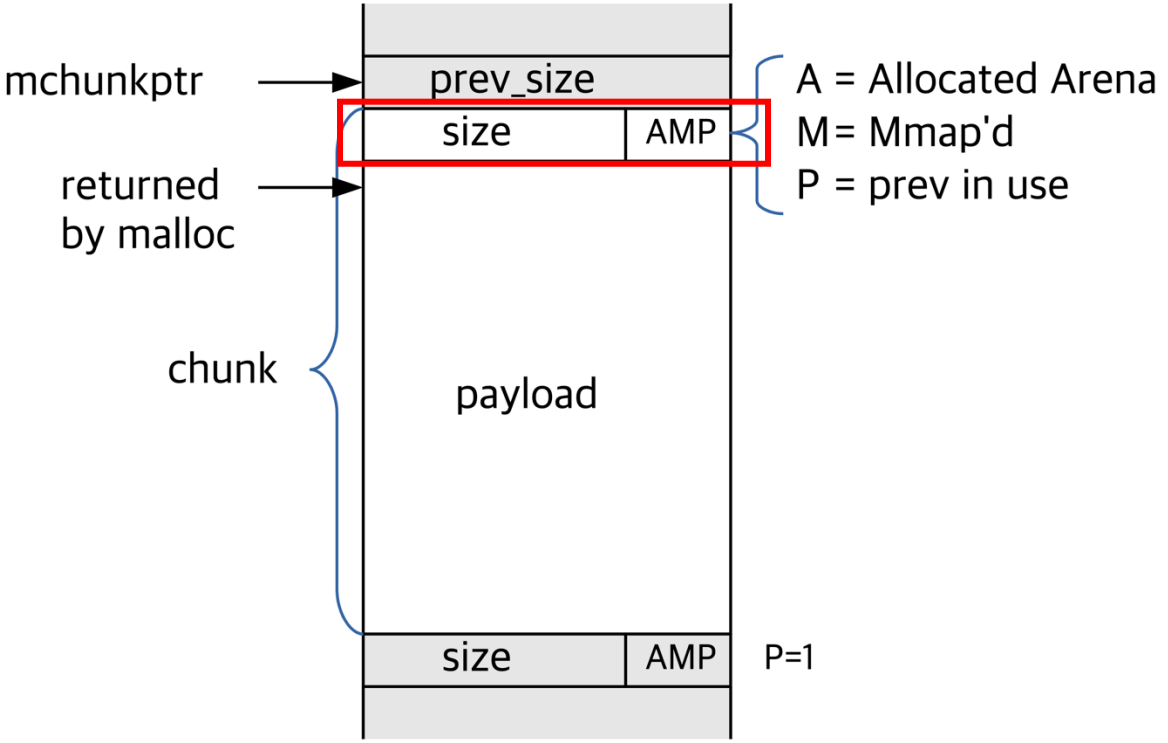
```
1 struct malloc_chunk {
2   INTERNAL_SIZE_T      mchunk_prev_size;  /* Size
3   INTERNAL_SIZE_T      mchunk_size;       /* Size
4   struct malloc_chunk* fd;                /* doubl
5 */struct malloc_chunk* bk;
6   /* Only used for large blocks: pointer to next l
7   struct malloc_chunk* fd_nextsize; /* double link
8   struct malloc_chunk* bk_nextsize;
9 };
10
11 typedef struct malloc_chunk* mchunkptr;
```



Chunk & bin

- Allocated Chunk
  - size
    - 현재 chunk의 크기를 가짐
    - 하위 3비트에 플래그 정보를 포함함
      - 외부 단편화를 없애기 위해 8바이트 단위로 정렬되기 때문에, 하위 비트를 사용 가능함

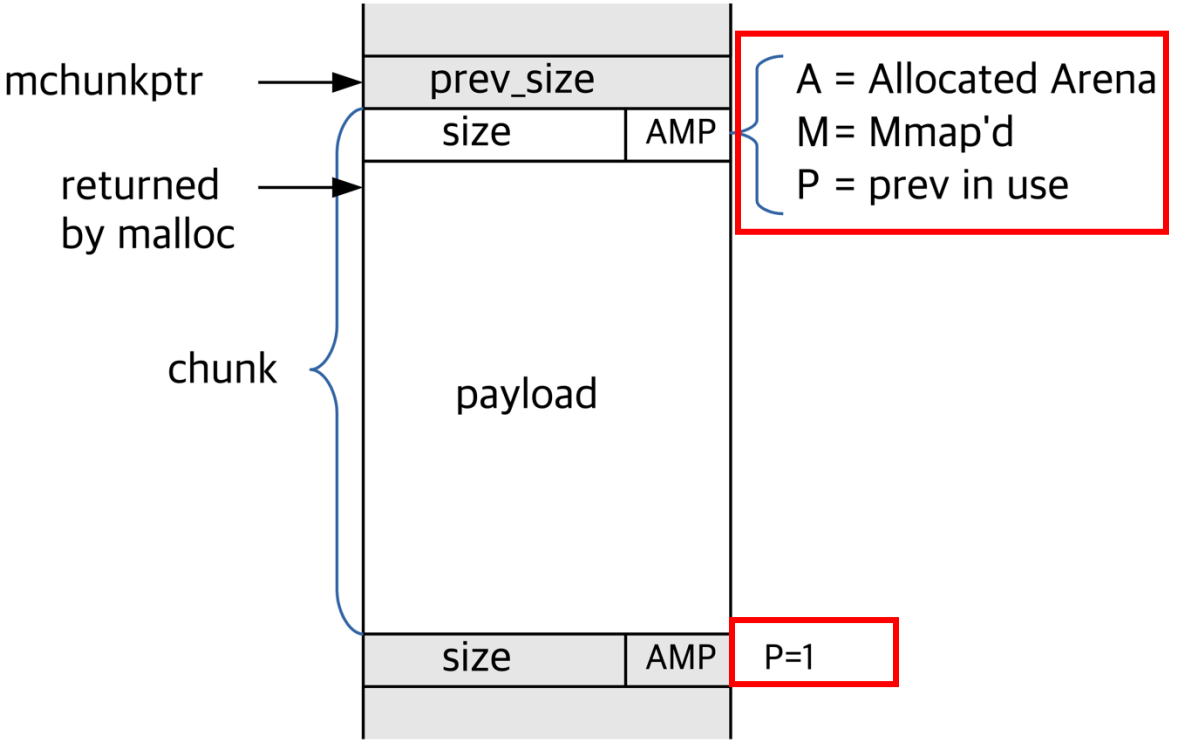
```
1 struct malloc_chunk {
2   INTERNAL_SIZE_T      mchunk_prev_size;  /* Size
3   INTERNAL_SIZE_T      mchunk_size;       /* Size
4   struct malloc_chunk* fd;                /* doubl
5 */struct malloc_chunk* bk;
6   /* Only used for large blocks: pointer to next l
7   struct malloc_chunk* fd_nextsize; /* double link
8   struct malloc_chunk* bk_nextsize;
9 };
10
11 typedef struct malloc_chunk* mchunkptr;
```



Chunk & bin

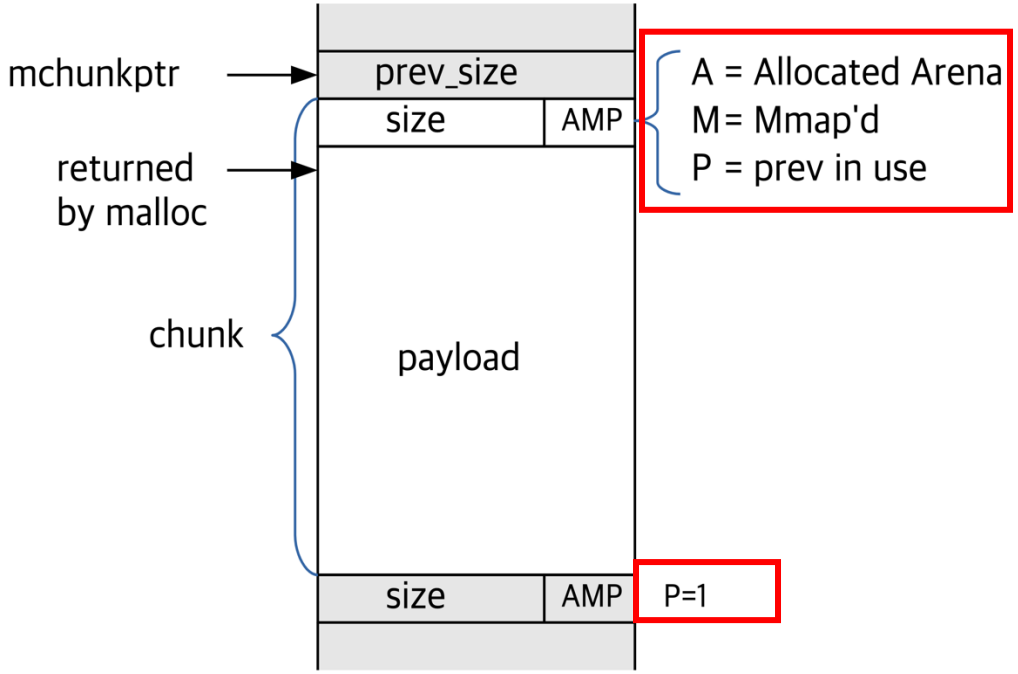
- Allocated Chunk
  - A(Allocated Arena, NON_MAIN_ARENA)
    - 0: chunk는 main아레나, main heap에서 나온 것임
    - 1: chunk는 mmap된 메모리에서 나온 것임
  - M(Mmap'd)
    - mmap을 통해 할당된 chunk인 경우 세팅됨

```
1 struct malloc_chunk {
2   INTERNAL_SIZE_T      mchunk_prev_size;  /* Size
3   INTERNAL_SIZE_T      mchunk_size;       /* Size
4   struct malloc_chunk* fd;                /* doubl
5 */struct malloc_chunk* bk;
6   /* Only used for large blocks: pointer to next l
7   struct malloc_chunk* fd_nextsize; /* double link
8   struct malloc_chunk* bk_nextsize;
9 };
10
11 typedef struct malloc_chunk* mchunkptr;
```



Chunk & bin

- Allocated Chunk
  - P(prev in use)
    - 0: prev chunk가 free된 상태임
    - 1: prev chunk가 allocated 상태임
  - “이전 chunk가 free인데, P가 1이에요” -> fastbin, tcache…
    - prev chunk가 병합 후보로 간주되지 않음을 나타내는 비트임
    - 애플리케이션이나 malloc 코드에 구현된 최적화 계층에 의해 사용 중임을 의미함



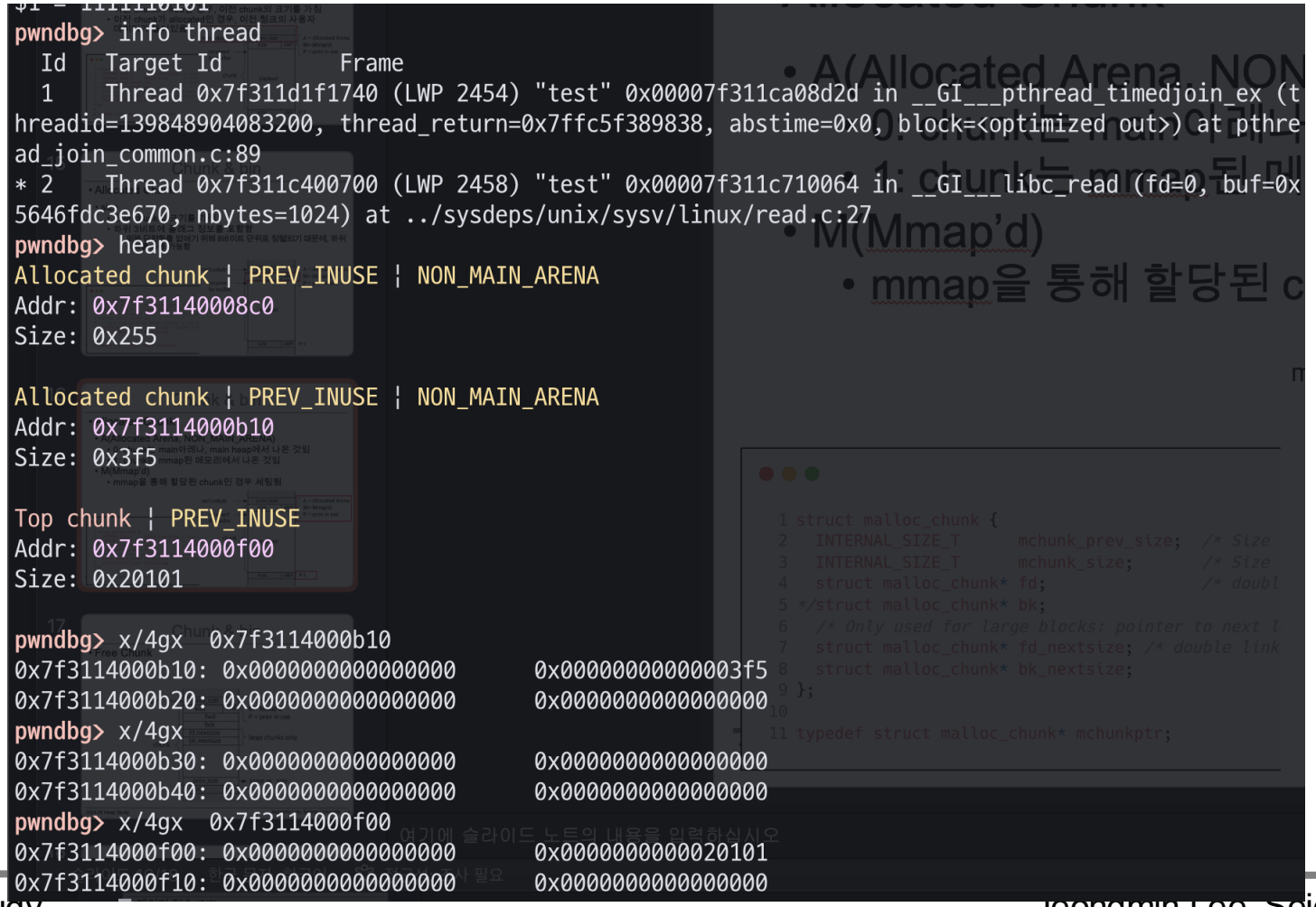
Chunk & bin

- 예시
  - main thread인 경우

```
pwndbg> info threads
  Id   Target Id         Frame
* 1    Thread 0x7f311d1f1740 (LWP 2454) "test" 0x00007f311ca08d2d in __GI___pthread_timedjoin_ex (t
hreadid=139848904083200, thread_return=0x7ffc5f389838, abstime=0x0, block=<optimized out>) at pthre
ad_join_common.c:89
  2    Thread 0x7f311c400700 (LWP 2458) "test" 0x00007f311c710064 in __GI___libc_read (fd=0, buf=0x
5646fdc3e670, nbytes=1024) at ../sysdeps/unix/sysv/linux/read.c:27
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0x5646fdc3e000
Size: 0x251

Allocated chunk | PREV_INUSE
Addr: 0x5646fdc3e250
Size: 0x411

Allocated chunk | PREV_INUSE
Addr: 0x5646fdc3e660
Size: 0x411

Free chunk (tcachebins) | PREV_INUSE
Addr: 0x5646fdc3ea70
Size: 0x3f1
fd: 0x00

Allocated chunk | PREV_INUSE
Addr: 0x5646fdc3ee60
Size: 0x121

Top chunk | PREV_INUSE
Addr: 0x5646fdc3ef80
Size: 0x20081
```

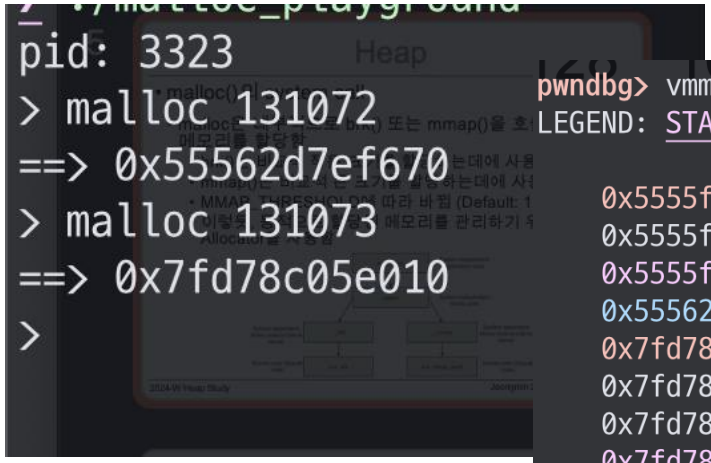
Chunk & bin

- 예시
  - main thread가 아닌 경우



Chunk & bin

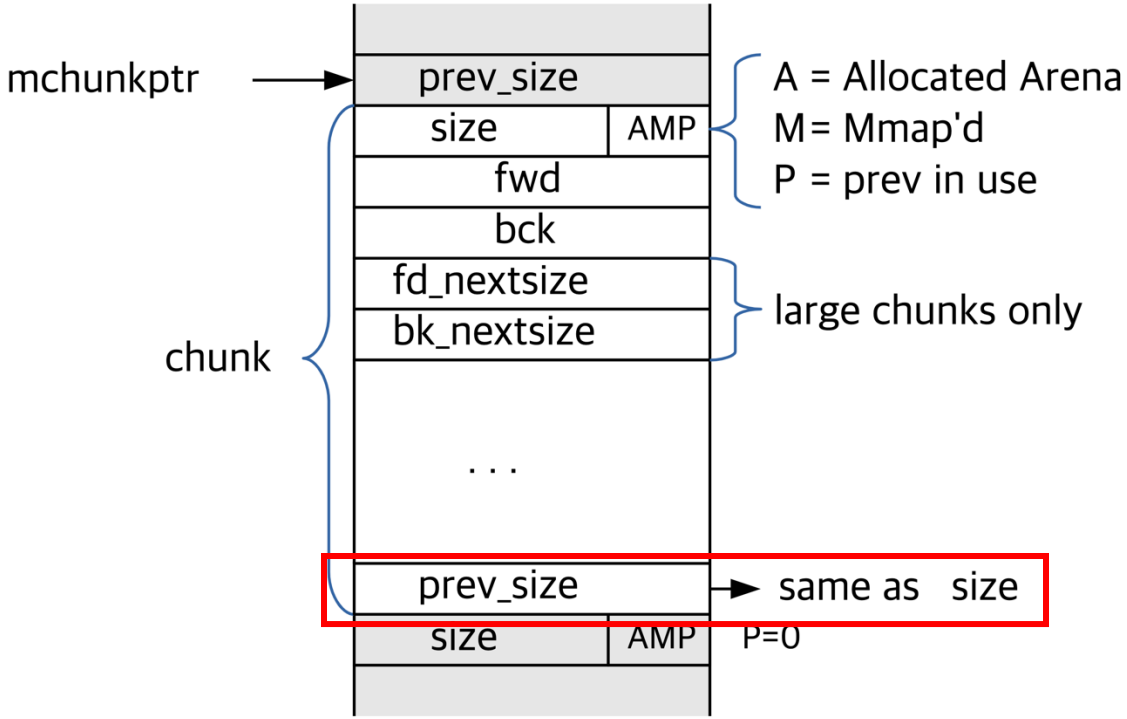
- 예시
  - MMAP_THREASHOLD(128 * 1024 bytes) 보다 큰 경우




```
pwndbg> vmmap
LEGEND: STACK ¦ HEAP ¦ CODE ¦ DATA ¦ RWX ¦ RODATA
              Start                End Perm     Size Offset File
    0x5555fc000000     0x5555fc002000 r-xp     2000      0 /study/how2heap/malloc_playground
    0x5555fc201000     0x5555fc202000 r--p     1000   1000 /study/how2heap/malloc_playground
    0x5555fc202000     0x5555fc203000 rw-p     1000   2000 /study/how2heap/malloc_playground
    0x55562d7ef000     0x55562d810000 rw-p    21000      0 [heap]
    0x7fd78ba00000     0x7fd78bbe7000 r-xp   1e7000      0 /lib/x86_64-linux-gnu/libc-2.27.so
    0x7fd78bbe7000     0x7fd78bde7000 ---p   200000 1e7000 /lib/x86_64-linux-gnu/libc-2.27.so
    0x7fd78bde7000     0x7fd78bdeb000 r--p     4000 1e7000 /lib/x86_64-linux-gnu/libc-2.27.so
    0x7fd78bdeb000     0x7fd78bded000 rw-p     2000 1eb000 /lib/x86_64-linux-gnu/libc-2.27.so
    0x7fd78bded000     0x7fd78bdf1000 rw-p     4000      0 [anon_7fd78bded]
    0x7fd78be00000     0x7fd78be29000 r-xp    29000      0 /lib/x86_64-linux-gnu/ld-2.27.so
    0x7fd78c029000     0x7fd78c02a000 r--p     1000  29000 /lib/x86_64-linux-gnu/ld-2.27.so
    0x7fd78c02a000     0x7fd78c02b000 rw-p     1000  2a000 /lib/x86_64-linux-gnu/ld-2.27.so
    0x7fd78c02b000     0x7fd78c02c000 rw-p     1000      0 [anon_7fd78c02b]
    0x7fd78c05e000     0x7fd78c081000 rw-p    23000      0 [anon_7fd78c05e]
    0x7fd78c087000     0x7fd78c08b000 r--p     4000      0 [vvar]
    0x7fd78c08b000     0x7fd78c08d000 r-xp     2000      0 [vdso]
    0x7ffe71ff4000     0x7ffe72015000 rw-p    21000      0 [stack]
0xffffffffff600000 0xffffffffff601000 r-xp     1000      0 [vsyscall]
pwndbg> x/4gx  0x55562d7ef670 - 0x10
0x55562d7ef660: 0x0000000000000000      0x0000000000020011
0x55562d7ef670: 0x0000000000000000      0x0000000000000000
pwndbg> x/4gx  0x7fd78c05e010 - 0x10
0x7fd78c05e000: 0x0000000000000000      0x0000000000021002
0x7fd78c05e010: 0x0000000000000000      0x0000000000000000
```

Chunk & bin

- Free Chunk
  - prev_size
    - prev_size는 size와 동일함
    - 이를 boundary tags 기법이라고 함
      - 병합 후보로 간주되지 않는 chunk는 마찬가지로 boundary tags를 세팅하지 않음



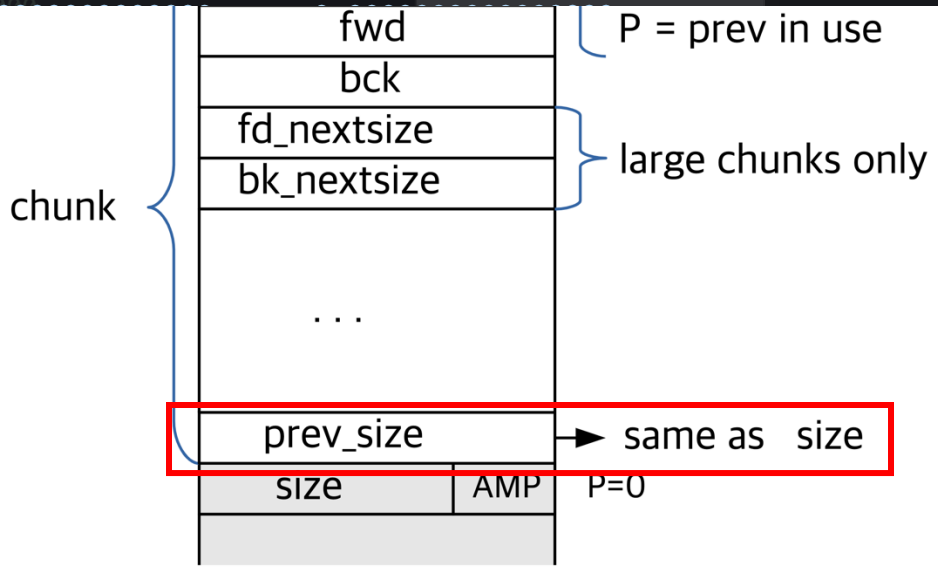
Chunk & bin

- Free Chunk
  - prev_size
    - prev_size는 size와 동일함

```
pid: 3867
> malloc 1080
==> 0x560b02657670
> malloc 1040
==> 0x560b02657ab0
> malloc 32
==> 0x560b02657ed0
```

```
0x560b02657650  0x0000000000000000  0x0000000000000000  ................
0x560b02657660  0x0000000000000000  0x0000000000000441  ........A.......
0x560b02657670  0x0000000000000000  0x0000000000000000  ................
```

```
0x560b02657a80  0x0000000000000000  0x0000000000000000  ................
0x560b02657a90  0x0000000000000000  0x0000000000000000  ................
0x560b02657aa0  0x0000000000000000  0x0000000000000421  ........!.......
0x560b02657ab0  0x0000000000000000  0x0000000000000000  ................
```



Chunk & bin

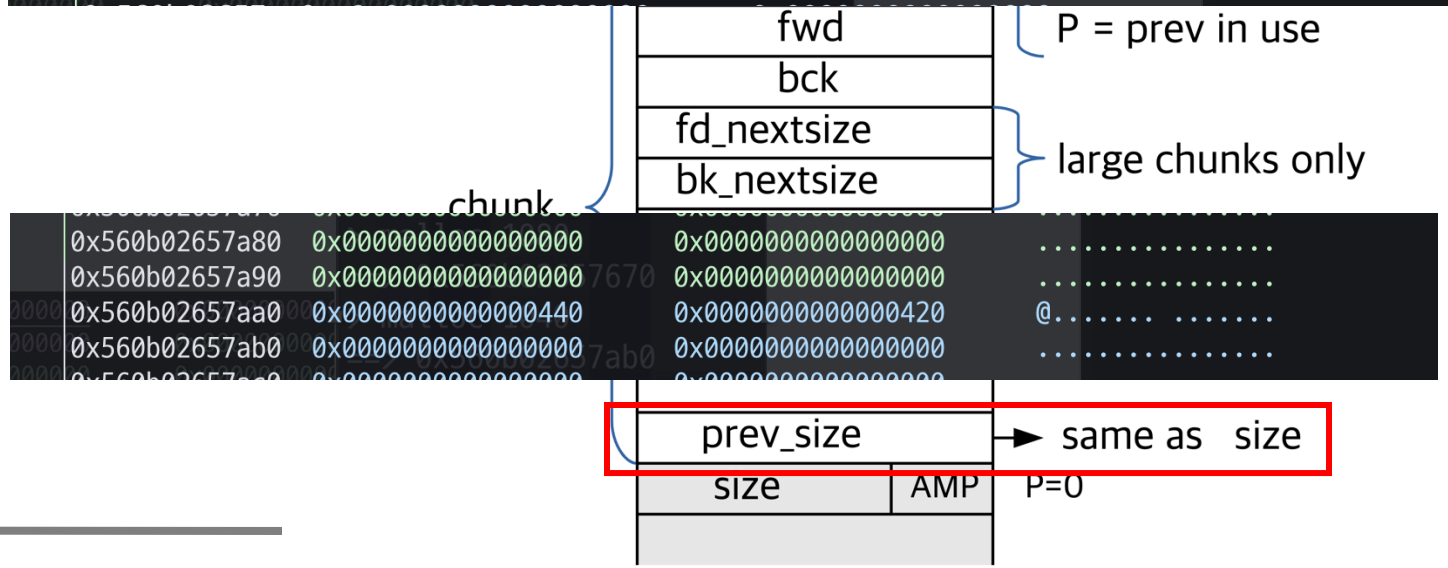
- Free Chunk
  - prev_size
    - prev_size는 size와 동일함

```
pid: 3867
> malloc 1080
==> 0x560b02657670
> malloc 1040
==> 0x560b02657ab0
> malloc 32
==> 0x560b02657ed0
```

```
pid: 3867
> malloc 1080
==> 0x560b02657670
> malloc 1040
==> 0x560b02657ab0
> malloc 32
==> 0x560b02657ed0
> free 0x560b02657670
==> ok
```

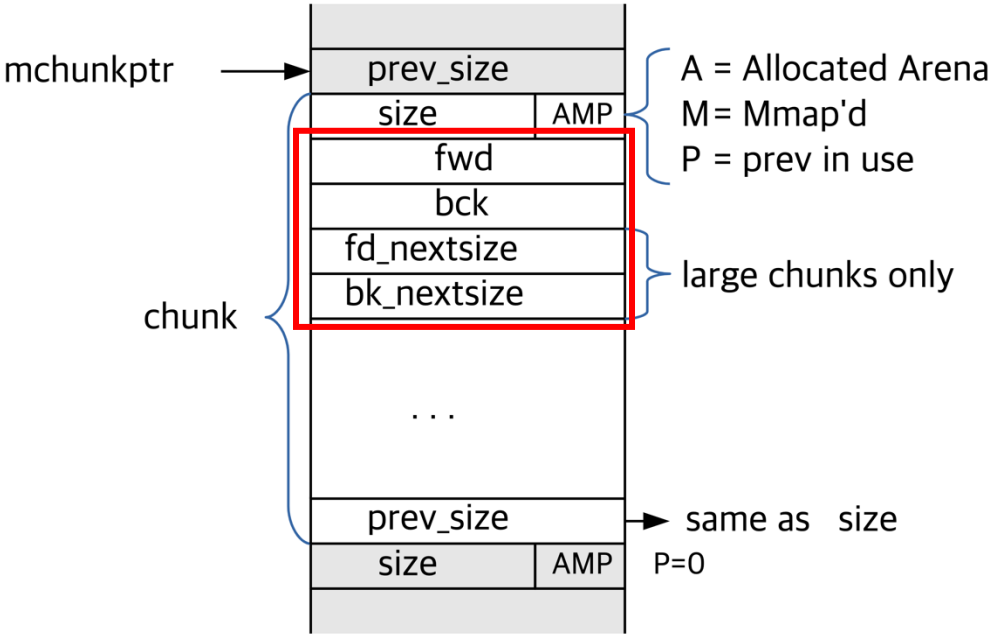
```
0x560b02657650  0x0000000000000000  0x0000000000000000  ................
0x560b02657660  0x0000000000000000  0x0000000000000441  ........A.......
0x560b02657670  0x0000000000000000  0x0000000000000000  ................
```

```
0x560b02657a80  0x0000000000000000  0x0000000000000000  ................
0x560b02657a90  0x0000000000000000  0x0000000000000000  ................
0x560b02657aa0  0x0000000000000000  0x0000000000000421  ........!.......
0x560b02657ab0  0x0000000000000000  0x0000000000000000  ................
```



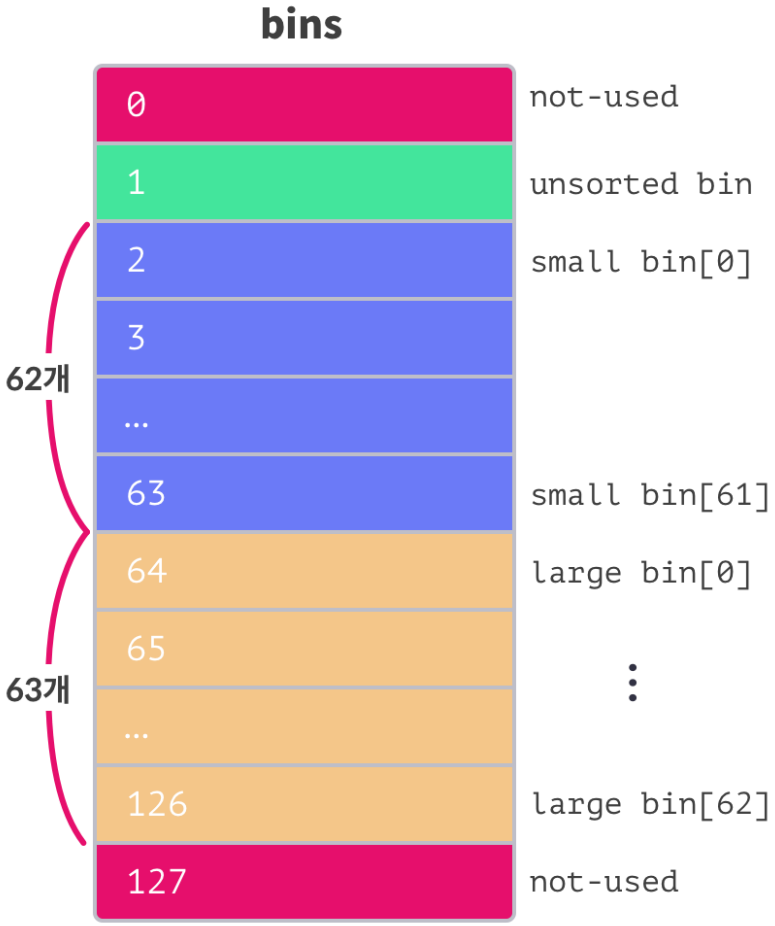
Chunk & bin

- Free Chunk
  - fd
    - 동일 bin의 다음 chunk를 가리킴(물리적 메모리의 다음 chunk 아님)
  - bk
    - 동일 bin의 이전 chunk를 가리킴(물리적 메모리의 이전 chunk 아님)
  - fd_nextsize, bk_nextsize
    - largebin에서 사용되며, fd, bk chunk의 크기를 가지고 있음



Chunk & bin

- bin
  - 정의
    - free chunk를 관리하는 freelist data structure
  - 종류
    - fastbin
    - smallbin
    - largebin
    - unsorted bin



Chunk & bin

- fastbin
  - 크기
    - 0x20(32) <= chunk_size <= 0xb0(176)
      - 디폴트로 7개의 fastbin만을 사용(global_max_fast에 의해 설정됨)

      ∴0x20 <= chunk_size <= 0x80(128) 까지임
  - 구조
    - LIFO(Last In, First Out) 구조를 가짐 (e.g., Stack)
    - 단일 연결 리스트 (fd만 존재, unlink 없음)
    - fastbin에 저장되는 chunk간에 병합 없음
      - 외부 단편화를 초래할 수 있으나, free 속도 향상

Chunk & bin

- smallbin
  - 크기
    - 0x20 <= chunk_size < 0x400
    - 62개의 bin이 존재, 16바이트 단위로 공간을 차지
      - e.g., 0x20, 0x30, 0x40 … 0x3f0
  - 구조
    - FIFO(Fist In, First Out) 구조 (e.g., Queue)
    - 원형 이중 연결 리스트 (fd, bk 존재, unlink 존재)
    - 메모리 상에서 인접한 두 청크가 해제되어 있고 이들이 smallbin에 있으면 병합(consolidation)을 수행함

Chunk & bin

- largebin
  - 크기
    - 0x400 <= chunk_size
    - 63개의 bin이 존재, 하나의 largebin에서 일정 범위의 청크를 모두 보관함
      - 크기 범위는 인덱스에 대해 로그적으로 증가함
      - e.g., largebin[0] => 1024 <= chunk_size <= 1088
        largebin[32] => 3072 <= chunk_size <= 3584
    - 가장 크기가 비슷한 best-fit 청크를 꺼내 할당함
  - 구조
    - 이중 연결 리스트, unlink 존재
    - 연속된 largebin 청크들은 병합 대상임

Chunk & bin

- unsorted bin
  - 특징
    - fastbin에 들어가지 않는 청크들은 해제 시, 크기를 구분하지 않고 unsorted bin에 저장됨
    - 원형 이중 연결 리스트 구조를 가짐
  - 예시
    - smallbin(0x20 ~ 0x400)에 해당하는 chunk 할당 요청 시
      - fastbin 또는 smallbin을 탐색한 뒤, unsorted bin을 탐색함
    - largebin 크기(>= 0x400)에 해당하는 chunk 할당 요청 시
      - unsorted bin을 먼저 탐색한 이후, largebin 탐색을 수행함

Chunk & bin

- tcache (thread local cache)
  - 정의
    - 각 스레드에 할당되는 캐시 저장소
  - 특징
    - >= glibc 2.26
    - 각 스레드는 64개의 tcache를 가짐
      - 각 스레드가 고유하게 가지고 있으므로, race condition을 고려하지 않고, 캐시에 접근 가능함
      - arena의 bin(unsorted, fast, small, large)에 접근하기 전에 tcache를 먼저 사용하므로 arena lock에 의한 병목현상 완화

Chunk & bin

- tcache (thread local cache)
  - 크기
    - 0x20 <= chunk_size <= 0x410 (1040)
      - 이 범위의 chunk들은 할당 및 해제할 때, tcache를 가장 먼저 조회함
    - 하나의 tcache는 같은 크기의 chunk 만을 보관함
      - 보관 가능한 갯수는 7개로 제한됨(메모리 낭비를 방지하기 위해)
      - tcache가 가득차면 적절한 bin으로 분류됨
      - e.g., 0x20사이즈 chunk를 9개 할당하고 해제한 상태

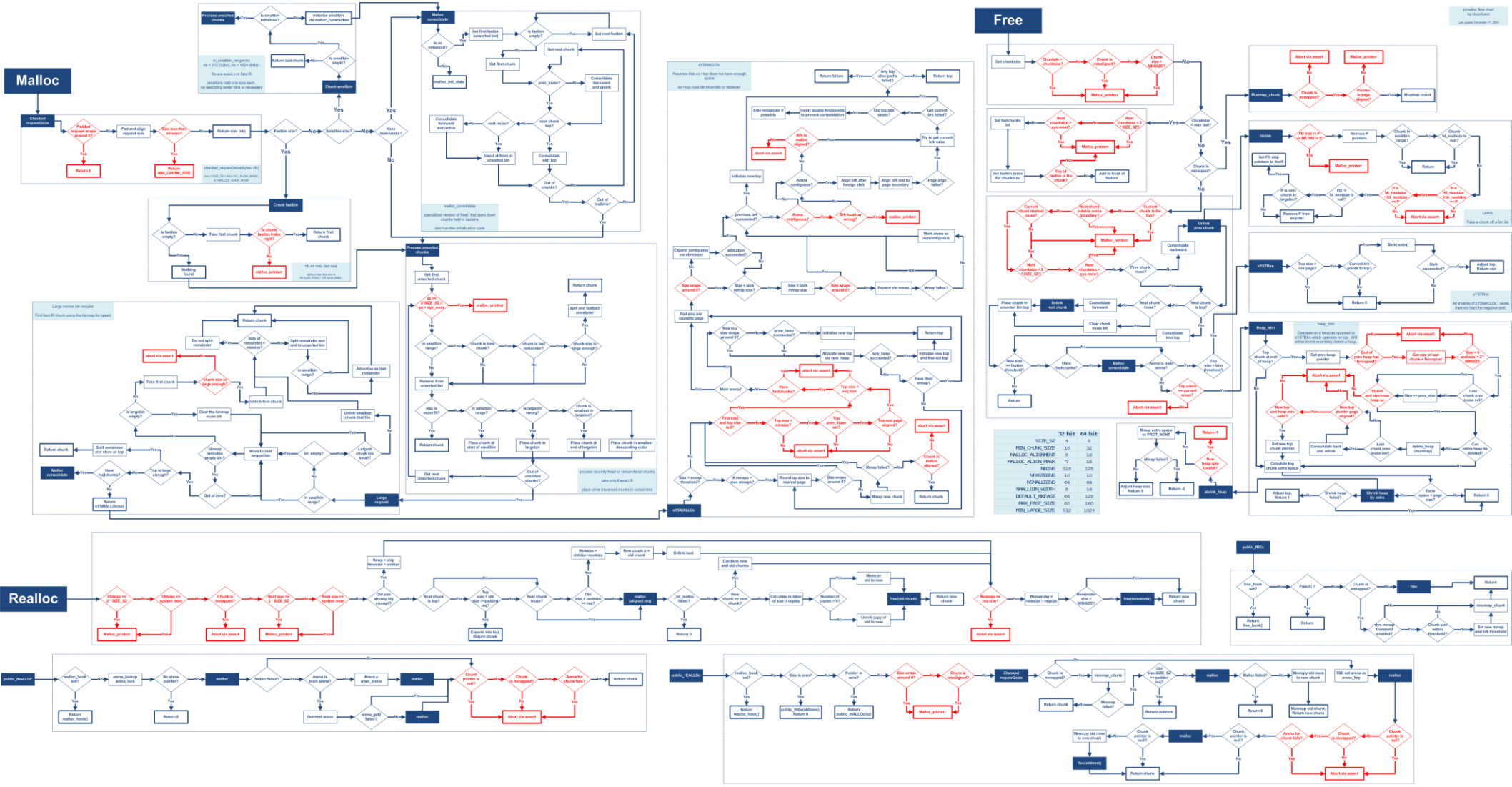
```
pwndbg> bins
tcachebins
0x30 [  7]: 0x62db819edbe0 —▸ 0x62db819edbb0 —▸ 0x62db819edb80 —▸ 0x62db819edb50 —▸ 0x62db819ed710 —▸ 0x62db819e
d6e0 —▸ 0x62db819ed6b0 ◂— 0
fastbins
0x30: 0x62db819edc30 —▸ 0x62db819edc00 ◂— 0
unsortedbin
empty
smallbins
empty
largebins
empty
```

Chunk & bin

- tcache (thread local cache)
  - 구조
    - LIFO 방식
    - 단일 연결 리스트
    - tcache에 들어간 청크들은 병합되지 않음

Chunk & bin

- 접근 순서...는 다음 기회에



Thanks!

이 정 민(neutrinox4b1@gmail.com)