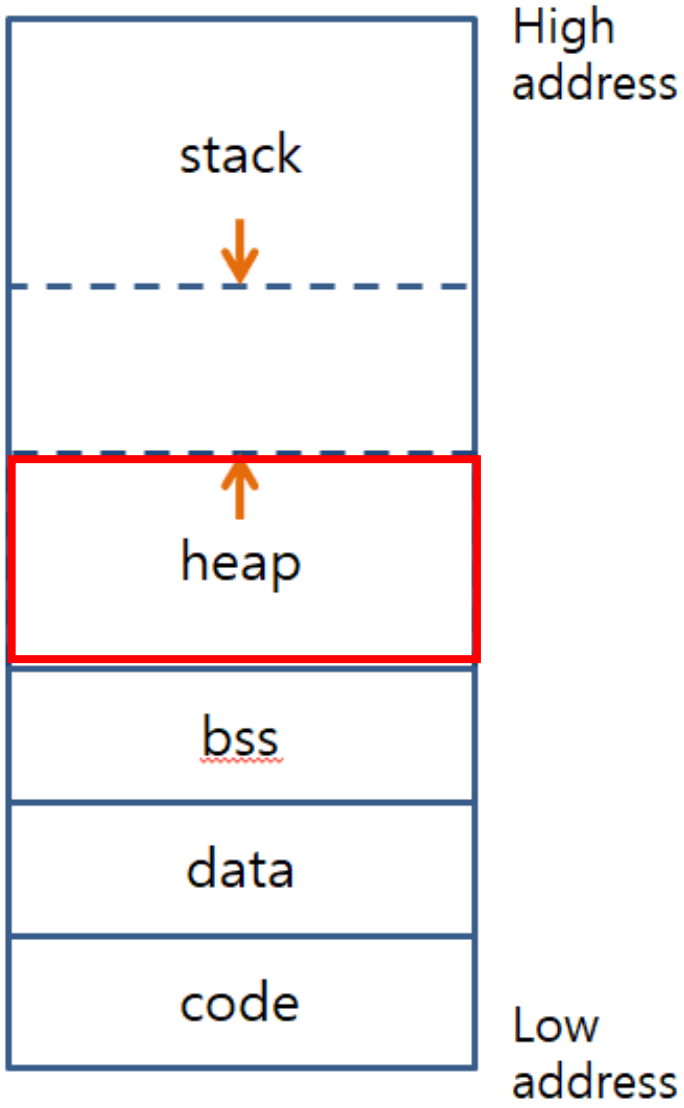
Heap

00

메모리 구조의 Heap

201716905 김강민

Index



- **메모리 구조의 Heap**

- FIFO(First In First Out) 형식의 메모리

- 동적 할당한 메모리 저장 (malloc, calloc, realloc 등)

- Stack 영역과 반대로 낮은 주소에서 높은 주소로 신장

- 리눅스에서 malloc_chunk 구조체 형식으로 데이터 저장

Heap의 할당

Heap
chunk 1 (0x10)
chunk 2 (0x120)
chunk 3 (0x70)
chunk 4 (0x400)
.
.

- **해제하지 않은 상태의 할당**

- 할당을 받으면 malloc_chunk 구조체 단위로 저장
(ptmalloc2 할당자 기준 - glibc)

- 크기에 상관 없이 순서대로 저장

- 물리적 메모리 근접

Heap
chunk 1 (0x10)
chunk 2 (0x120)
chunk 3 (0x70)
chunk 4 (0x400)
.
.

chunk 5 (0x70)

- **해제된 Heap의 재사용**

- 메모리 공간 사용의 효율을 증가 시키기 위해 해제된 공간을 재사용

- 할당될 chunk의 크기와 유사한 크기의 chunk가 free 상태라면, 그곳에 새로운 chunk 데이터를 저장

- 할당될 chunk의 크기에 맞는 곳이 없다면 순차적으로 저장

해제된 Heap의 관리


```
struct malloc_state
{
  /* Serialize access.  */
  mutex_t mutex;

  /* Flags (formerly in max_fast).  */
  int flags;

  /* Fastbins */
  mfastbinptr fastbinsY[NFASTBINS];

  /* topchunk의 base address */
  mchunkptr top;

  /* 가장 최근의 작은 요청으로부터 분리된 나머지 */
  mchunkptr last_remainder;

  /* 위에서 설명한대로 pack된 일반적인 bins */
  mchunkptr bins[NBINS * 2 - 2];

  /* Bitmap of bins */
  unsigned int binmap[BINMAPSIZE];

  /* 연결 리스트 */
  struct malloc_state *next;

  /* 해제된 아레나를 위한 연결 리스트 */
  struct malloc_state *next_free;

  /* 현재 Arena의 시스템으로부터 메모리 할당  */
  INTERNAL_SIZE_T system_mem;
  INTERNAL_SIZE_T max_system_mem;
};
```


- **malloc_state**

- brk 시스템 콜을 사용하여 할당된 힙을 효율적으로 관리하기 위해 존재하는 구조체

- 각 Thread에 대한 힙 영역 관리

- 해제된 chunk는 fastinsY와 bins에 크기에 맞춰 저장

- 새로 할당이 들어올 때, 크기에 맞는 bin에서 유사한 크기의 해제된 chunk를 찾아서 재사용

해제된 Heap의 관리

Bins	Fast bin	Small bin	Large bin	Unsorted bin
Chunk type	Fast chunk	Small chunk	Large chunk	Small, Large chunk
The size of chunk	0x20 ~ 0xb0	1024 바이트 미만	1024 바이트 이상	제한 없음 (Free chunk만 등록 가능)
Bin 개수	10	62	63	1
Free chunk 관리 특징	• Free chunk가 서로 인접해 있어도 하나의 단일 Free chunk로 병합되지 않습니다.	• 2개의 Free chunk는 서로 인접해 있을 수 없습니다. • Free chunk가 서로 인접해 있으면 하나의 단일 Free chunk로 병합됩니다.		-

- **bin**

- 재사용을 하기위해 free된 chunk의 주소를 크기별로 저장해둔 배열

- fastbinsY : fastbin을 관리 (32 bit - 0x10 ~ 0x40, 64 bit - 0x20 ~ 0x80)

- bins : small bin, large bin, unsorted bin을 관리

- **Fast bin**

- LIFO(Last In Fisrt Out) 구조를 사용

- 속도 향상을 위해 단일 연결리스트 구조 (다른 bin들은 이중 연결리스트)

- fast bin이 처리하는 메모리의 최대 크기는 global_max_fast 변수 에 의해서 결정

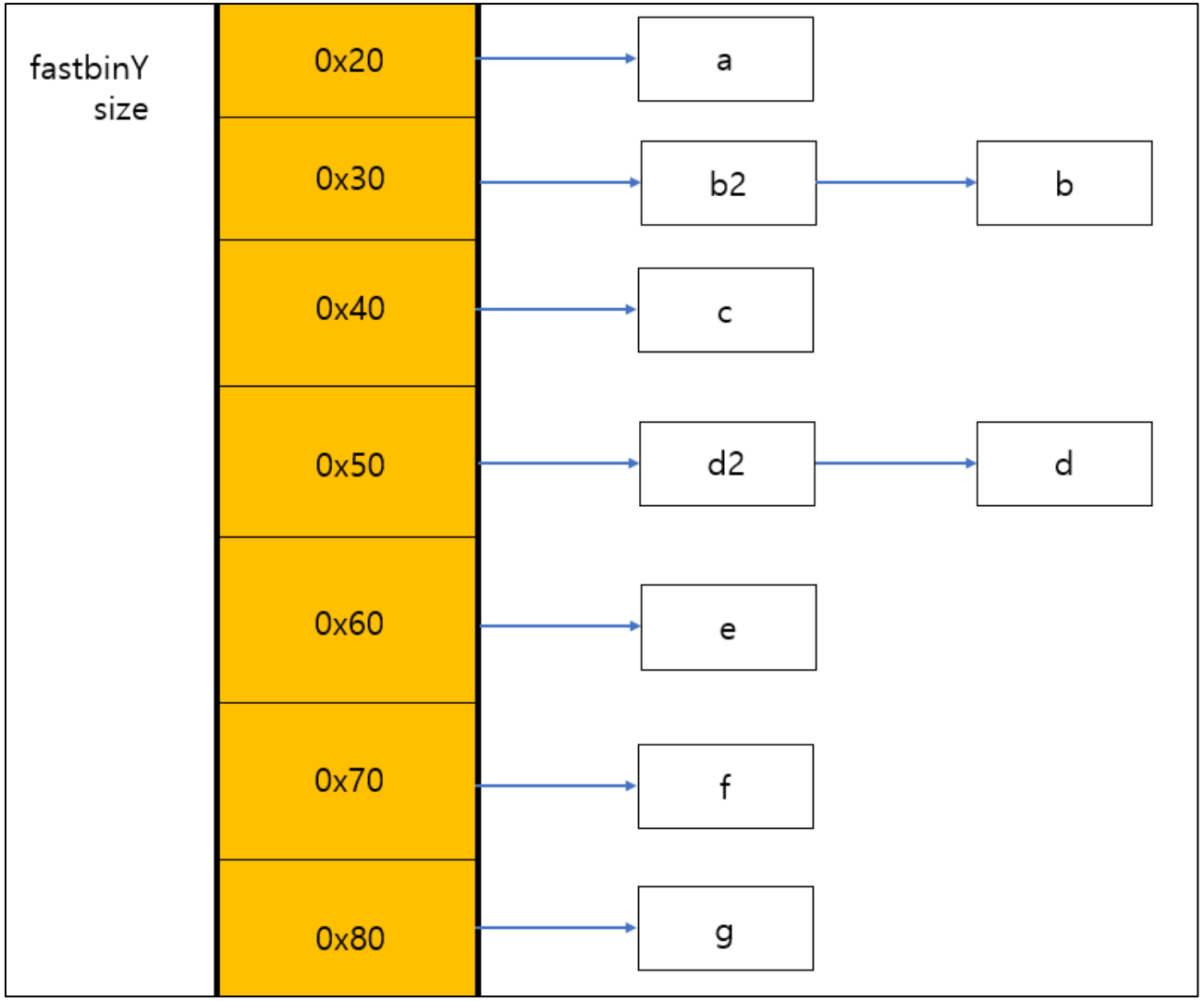
- 10개의 bin 사용 (64 bit 기준으로 10개의 bin이 존재하지만, 디폴트로 7개만 사용)

  32 bit - 16, 24, 32, 40, 48, 56, 64 byte

  64 bit - 32, 48, 64, 80, 96, 112, 128 byte

(free chunk 크기가 98과 110이 있으면 둘 다 112 bin에 저장)

Fast bin의 재사용



- **fastbin 재사용 과정**

1. 0x28 크기의 chunk 할당 요청

2. 해당 크기가 fastbinY에 포함되는지 확인

3. fastbinY 배열 중 할당 요청된 크기에 맞는 배열 확인 (여기서는 0x30)

4. 해당 크기 배열의 가장 첫번째인 b2에 해당하는 위치를 재사용

- **Unsorted bin**

- unsorted bin은 1개만 존재, FIFO 구조, 이중연결리스트

- small bin이나 large bin 크기의 chunk를 저장

- 해당 bin을 이용해 적절한 bin을 찾는 시간이 적기 때문에 할당과 해제의 처리속도가 빠름 (캐쉬와 유사)

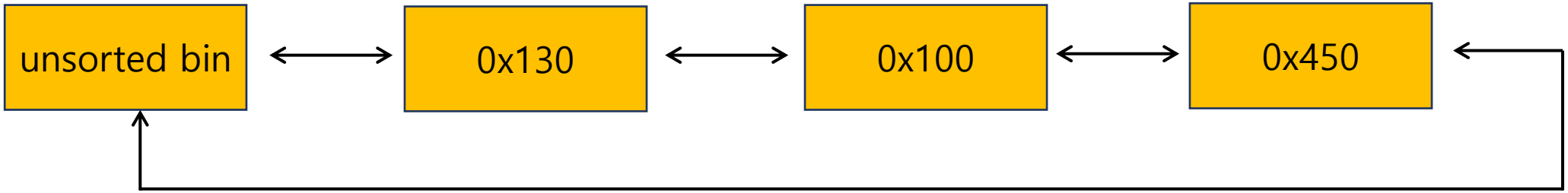
- 해당 bin은 Chunk 크기에 대한 제한이 없기 때문에 다양한 크기의 청크가 해당 Bin에 저장될 수 있다

- free()된 청크는 unsorted bin에 보관되며, 메모리 할당 시 동일한 크기의 영역을 다시 요청하는 경우 해당 영역을 재사용함. (fast chunk는 예외)

- 검색된 청크는 바로 재할당되거나 실패하면 원래의 Bin을 돌아감

- large bin이나 small bin은 해당 bin에 바로 들어가지 않고 unsorted bin에 우선 저장

Unsorted bin 동작 과정



- 다양한 크기의 chunk가 존재

- FIFO 구조

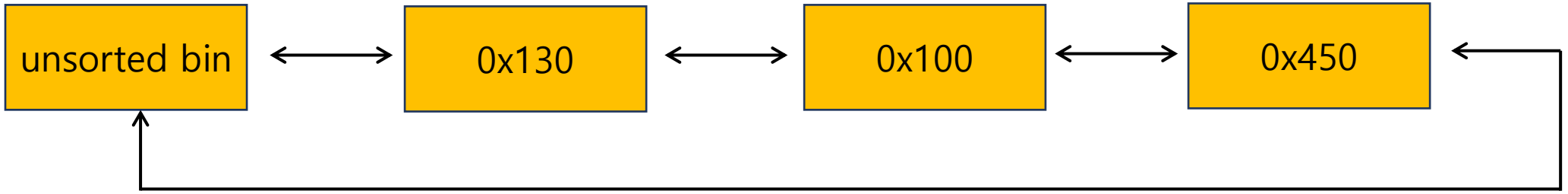
- 이중 연결리스트

- large bin, small bin에 들어가기 전에 unsorted bin에 저장 후, 재할당 실패시 해당 bin으로 저장

Unsorted bin 동작 과정

0x100 할당 요청

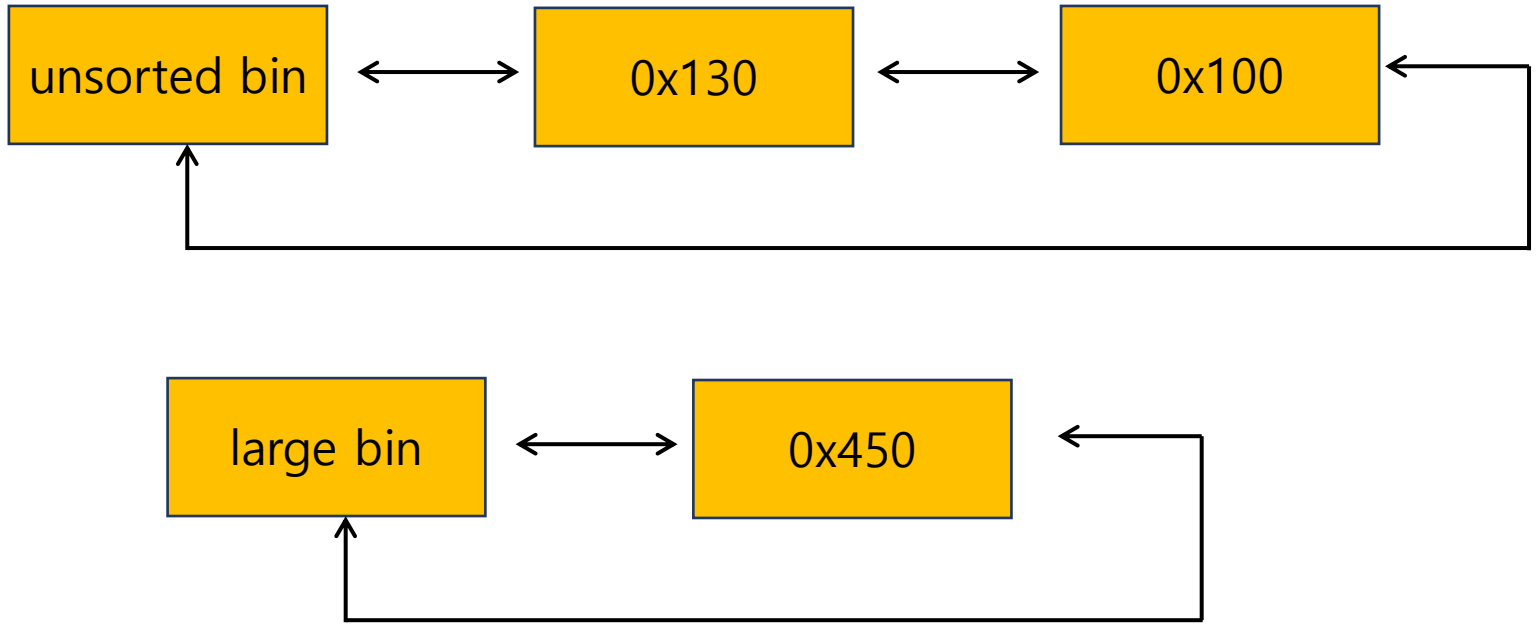
크기가 맞지 않음.
재할당 실패
(large bin에 저장)



Unsorted bin 동작 과정

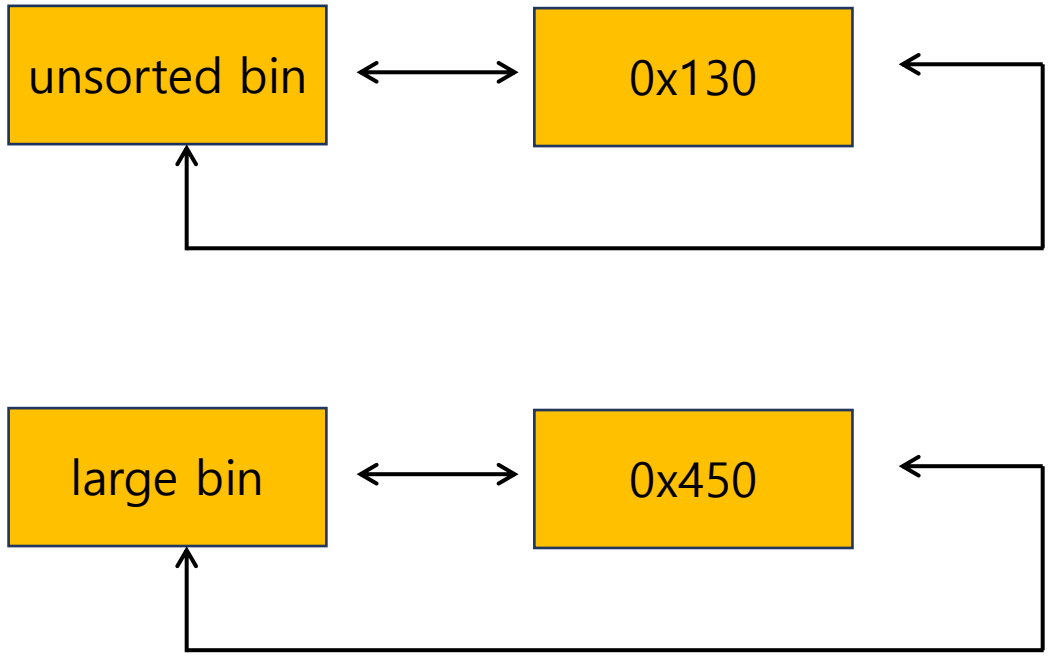
0x100 할당 요청

요청에 맞는 크기
재할당



Unsorted bin 동작 과정

0x100 할당 요청



- **Small bin**

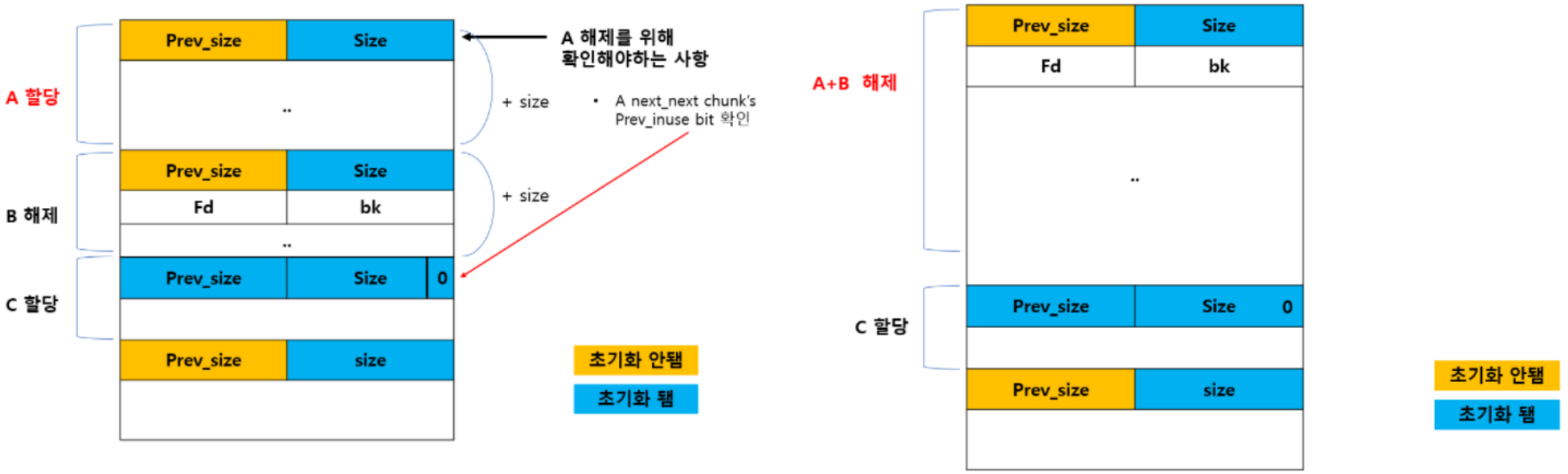
- 62개의 bin을 사용함 (small_bin[0]~small_bin[61]), 8 byte 씩 증가

- 0x20 ~ 0x400 미만의 크기를 가지는 chunk를 관리

- 해당 bin에는 2개의 Free chunk가 서로 물리적으로 인접 불가 (물리적으로 인접 시 병합)

- FIFO 구조, 이중연결리스트

- small bin에 저장하기 전, unsorted bin에 먼저 저장된 후 재할당 실패 시 small bin에 저장



- **chunk의 결합**

- 물리적으로 인접해 있는 free chunk는 서로 병합하여 더 큰 chunk를 만든다. (fastbin 제외)

- chunk의 free 여부는 다음 chunk의 Prev_inuse bit를 통해서 확인한다.

- **Large bin**

- 63개의 bin을 사용함 (small_bin[0]~small_bin[62])

- 0x400 이상의 크기를 가지는 chunk를 관리

- 128k byte 이상은 bin에서 관리하지 않고 mmap() 시스템 콜을 통해 별도 관리

- fd_nextsize와 bk_nextsize를 이용

- bin 내부에서 내림차순으로 정렬

- 이외에는 small bin과 동일

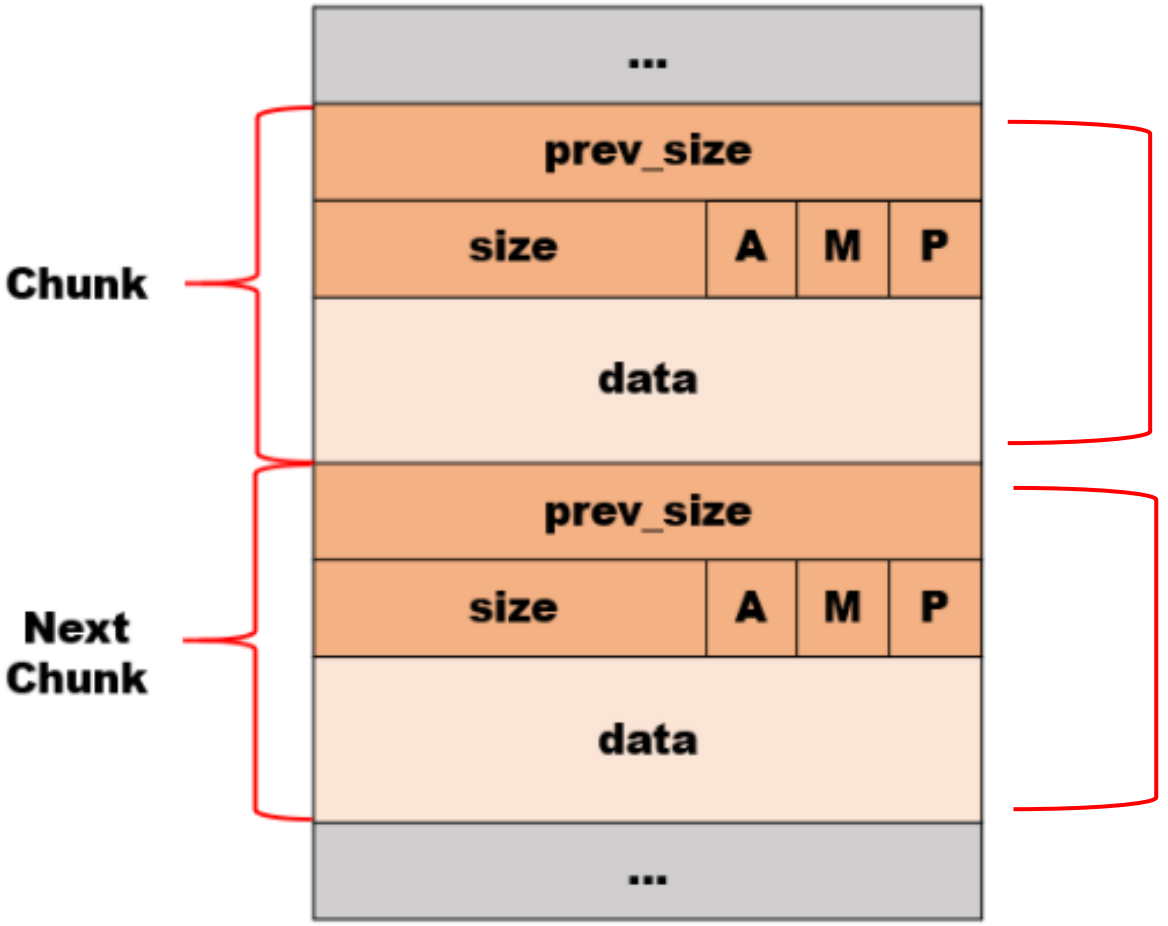
- **bin의 개수** : 63개, 대략적인 크기의 chunk를 저장
  - 32개의 bin : 64 Bytes씩 증가
  - 16개의 bin : 512 Bytes씩 증가
  - 8개의 bin : 4096 Bytes씩 증가
  - 4개의 bin : 32768 Bytes씩 증가
  - 2개의 bin : 262144 Bytes씩 증가
  - 1개의 bin : 이 외의 남은 크기의 chunk

Chunk의 구조


```
struct malloc_chunk
{
    INTERNAL_SIZE_T      prev_size;
    INTERNAL_SIZE_T      size;

    struct malloc_chunk* fd;
    struct malloc_chunk* bk;

    struct malloc_chunk* fd_nextsize;
    struct malloc_chunk* bk_nextsize;
};
```

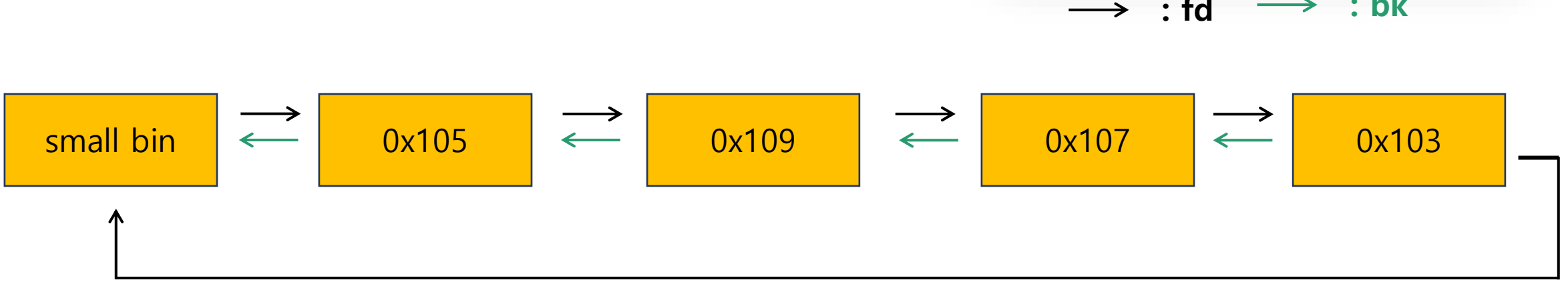
- **prev_size** : 물리적 메모리의 이전 chunk의 size
- **size** : 현재 청크의 size, 마지막 3bit는 Flags로 현재 chunk의 상태를 나타낸다.
- **fd** : 같은 bin의 다음 chunk를 가리키는 포인터
- **bk** : 같은 bin의 이전 chunk를 가리키는 포인터
- **fd_nextsize** : 크기가 작은 첫번째 chunk를 가리키는 포인터 (large bin)
- **bk_nextsize** : 크기가 큰 첫번째 chunk를 가리키는 포인터 (large bin)

Chunk의 구조



- size나 prev_size는 할당 요청한 크기에 포함되어 있다.
  (32bit - 0x8, 64bit - 0x10)
- size의 끝 3bit는 Flags로 chunk의 상태를 나타낸다.
- A : main_arena에서 관리여부
  M : mmap 시스템 콜 사용 할당 여부
  P : 이전 chunk의 할당 여부

Chunk의 구조



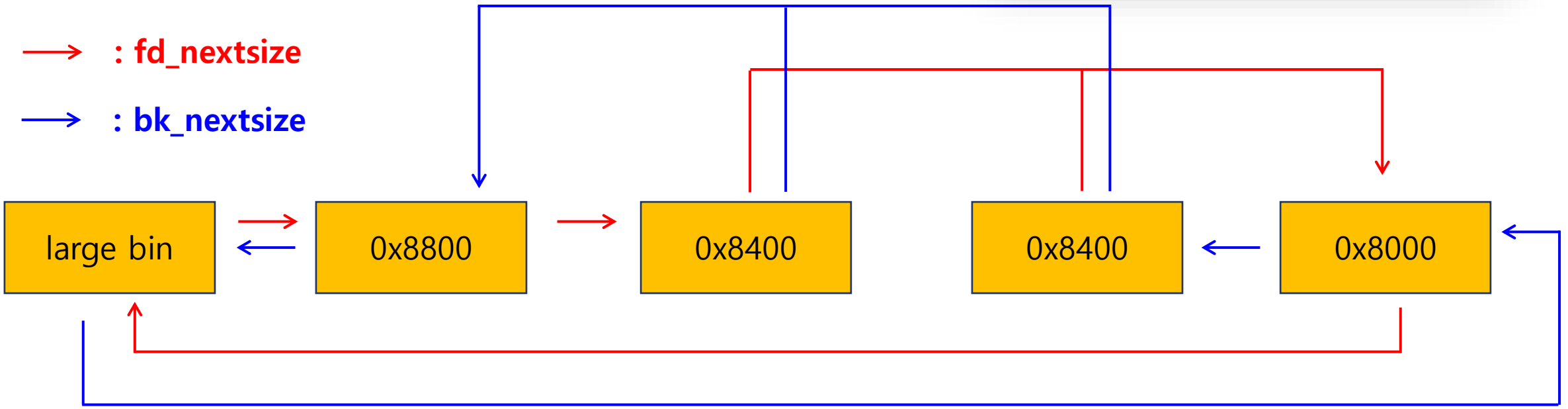
- **fd와 bk**

- fastbin을 제외한 다른 bin에서 모두 사용 (fastbin은 fd만 사용)

- 물리적으로 근접한 chunk가 아닌 bin 내부에서 앞, 뒤로 인접한 chunk를 가리키는 포인터

- 가장 오른쪽이 먼저 들어온 chunk이며, 가장 먼저 재사용되는 chunk

Chunk의 구조

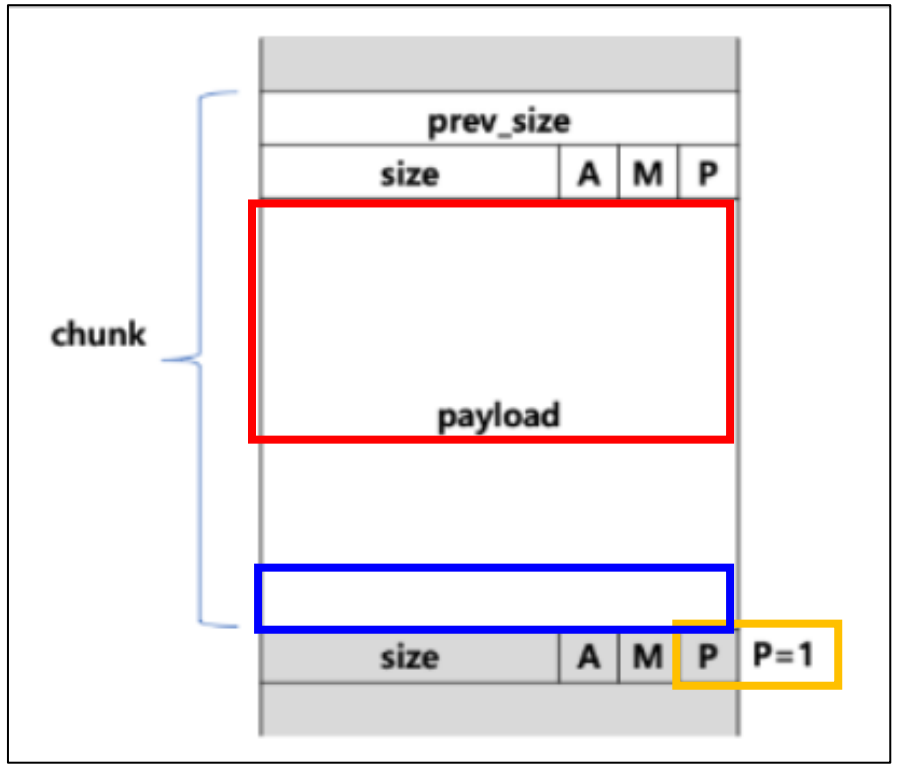


- **fd_nextsize와 bk_nextsize**

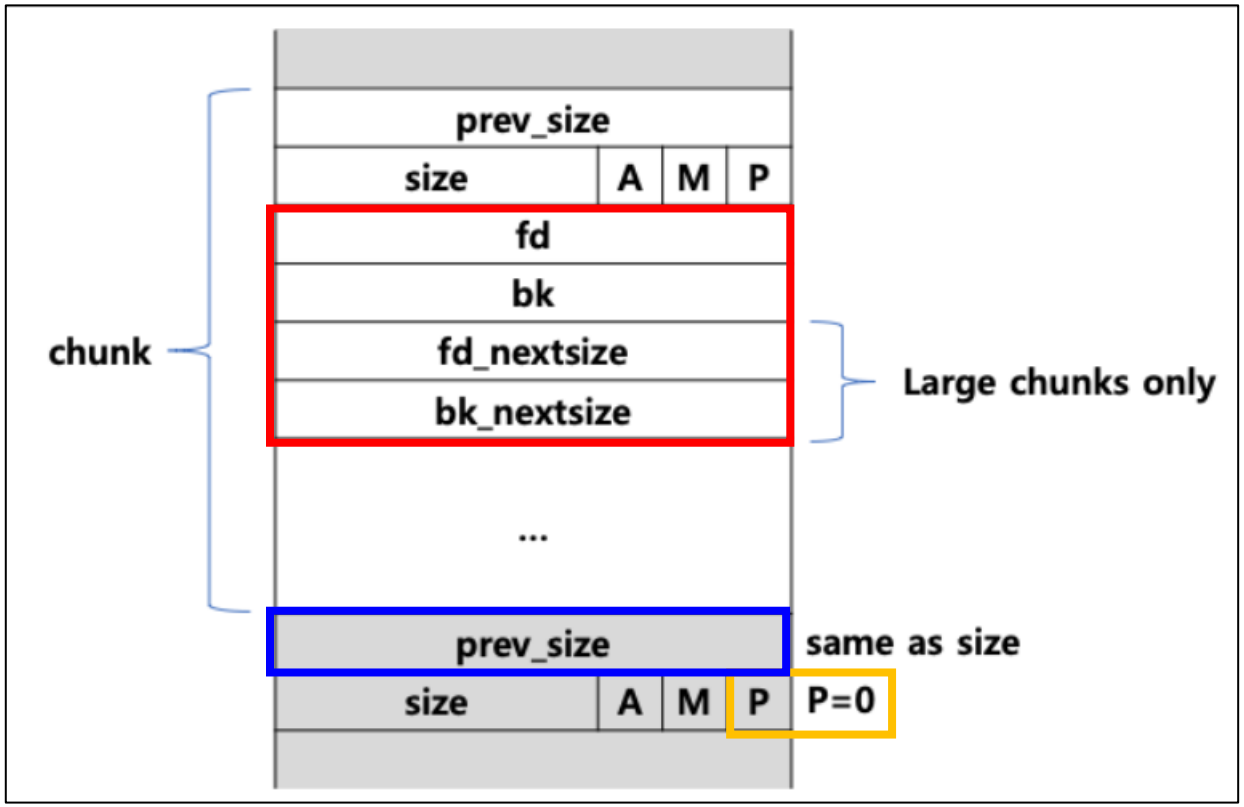
- large bin에서만 사용 (bin 내부에서 크기 순으로 정렬하기 때문)

- fd나 bk와 다르게 바로 앞을 가리키지 않고, 크기가 다른 첫번째 chunk를 가리키는 포인터

- in-use chunk



- free chunk

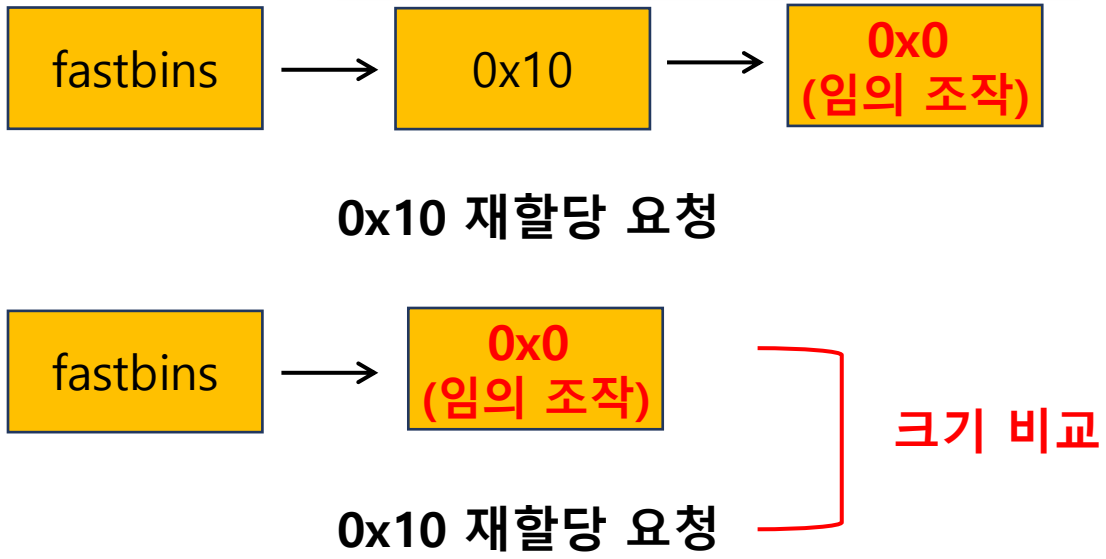


여기서 P는 Prev_inuse bit로 물리적으로 인접한 이전 chunk의 사용여부를 나타낸다.

0 : 미사용, 1: 사용

_int_malloc security check


```
#define fastbin_index(sz) \
  ((((unsigned int) (sz)) >> (SIZE_SZ == 8 ? 4 : 3)) - 2)

idx = fastbin_index (nb);
if (victim != 0)
{
    if (__builtin_expect (fastbin_index (chunksize (victim)) !=
idx, 0))
    {
        errstr = "malloc(): memory corruption (fast)";
      errout:
        malloc_printerr (check_action, errstr, chunk2mem
(victim), av);
        return NULL;
    }
}
```



—> 같은 bin에 속하지 않으므로 error 발생

- **malloc(): memory corruption (fast)**

- 할당하려는 fastbin의 크기와 할당될 영역의 크기를 구한 후, 두 크기가 같은 bin에 속하는지 확인

_int_malloc security check


```
while ((victim = unsorted_chunks (av)->bk) != unsorted_chunks (av))
{
  bck = victim->bk;
  if (__builtin_expect (victim->size <= 2 * SIZE_SZ, 0)
      || __builtin_expect (victim->size > av->system_mem, 0))
    malloc_printerr (check_action, "malloc(): memory corruption",
        chunk2mem (victim), av);
```

- **malloc(): memory corruption ()**

- unsorted bin의 bk가 unsorted bin 주소인지 확인 (unsorted bin이 비어있는지 확인)

- 비어있으면 2 * SIZE_SZ(최소 사이즈)보다 작거나 av->system_mem(최대 사이즈)보다 큰지 확인

- 해당 사이즈에 포함되지 않으면 error 발생

_int_free security check


```
#define misaligned_chunk(p) \
  ((uintptr_t)(MALLOC_ALIGNMENT == 2 * SIZE_SZ ? (p) :
chunk2mem (p)) \
   & MALLOC_ALIGN_MASK)
size = chunksize (p);

if (__builtin_expect ((uintptr_t) p > (uintptr_t) -size, 0)
    || __builtin_expect (misaligned_chunk (p), 0))
{
  errstr = "free(): invalid pointer";
  errout:
  if (!have_lock && locked)
        (void) mutex_unlock (&av->mutex);
    malloc_printerr (check_action, errstr, chunk2mem (p), av);
```

```
gdb-peda$ x/10x 0x602010-0x10
0x602000:    0x0000000000000000    0xfffffffffffffff00
0x602010:    0x0000000000000000    0x0000000000000000
```

error 발생

- **free(): invalid pointer**

- 해제하려는 chunk의 size가 음수로 변환하여 chunk 주소랑 비교 후 작으면 에러

- 해제하려는 chunk의 주소에 malloc_chunk 구조체가 존재하는지 확인

- 사이즈가 음수거나, malloc_chunk 구조체 아니면 error 발생

_int_free security check


```
#define MIN_CHUNK_SIZE        (offsetof(struct malloc_chunk,
fd_nextsize))
#define MINSIZE  \
  (unsigned long)(((MIN_CHUNK_SIZE+MALLOC_ALIGN_MASK) &
~MALLOC_ALIGN_MASK))

if (__glibc_unlikely (size < MINSIZE || !aligned_OK (size)))
{
  errstr = "free(): invalid size";
  goto errout;
}
```

```
gdb-peda$ x/10gx 0x602000
0x602000:    0x0000000000000000    0x0000000000000019
0x602010:    0x0000000000000000    0x0000000000000000
0x602020:    0x0000000000000000    0x0000000000020fe1
```

error 발생

- **free(): invalid size**

- 해제하려는 chunk의 size가 최소 사이즈 0x20 (32bit는 0x10)보다 작은지 확인

- 해제하려는 chunk의 주소에 malloc_chunk 구조체가 존재하는지 확인

- 사이즈가 최소 사이즈보다 작거나, malloc_chunk 구조체 아니면 error 발생

_int_free security check


```
#define chunk_at_offset(p, s)  ((mchunkptr) (((char *) (p)) + (s)))

if (have_lock
      || ({ assert (locked == 0);
      mutex_lock(&av->mutex);
      locked = 1;
      chunk_at_offset (p, size)->size <= 2 * SIZE_SZ
        || chunksize (chunk_at_offset (p, size)) >= av->system_mem;
        }))
{
      errstr = "free(): invalid next size (fast)";
      goto errout;
}

if (__builtin_expect (nextchunk->size <= 2 * SIZE_SZ, 0)
  || __builtin_expect (nextsize >= av->system_mem, 0))
{
  errstr = "free(): invalid next size (normal)";
  goto errout;
}
```

- **free(): invalid next size(fast, normal)**

- 해제하려는 chunk의 주소에 size만큼 더한 (다음 chunk의 주소) chunk의 size를 검사

- 다음 chunk의 size가 최소 사이즈보다 작은지 확인

- 다음 chunk의 size가 최대 크기보다 큰지 확인

```
gdb-peda$ x/10x 0x602000
0x602000:   0x0000000000000000  0x0000000000000021
0x602010:   0x0000000000000000  0x0000000000000000
0x602020:   0x0000000000000000  0x0000000000022001
0x602030:   0x0000000000000000  0x0000000000000000
0x602040:   0x0000000000000000  0x00000000000020fc1
```

해제하려는 chunk

다음 chunk size

_int_free security check


```
mchunkptr old = *fb, old2;
if (__builtin_expect (old == p, 0))
{
  errstr = "double free or corruption (fasttop)";
  goto errout;
}
```

double free 발생하지만 error 발생 X

- **double free or corruption (fasttop)**

- old 포인터는 이전에 해제한 포인터 저장

- 연속적으로 같은 포인터를 해제할 경우 error 발생

- 연속적으로만 해제하지 않으면 double free 가능

_int_free security check


```
if (__glibc_unlikely (p == av->top))
{
  errstr = "double free or corruption (top)";
  goto errout;
}
```

- **double free or corruption (top)**

- 해제하려는 chunk가 top chunk인 경우 error

- top chunk는 힙의 제일 상단에 위치한 chunk로 힙 메모리 영역의 끝, 아직 사용되지 않은 힙 영역

_int_free security check


```
nextchunk = chunk_at_offset(p, size);

if (__builtin_expect (contiguous (av)
        && (char *) nextchunk
        >= ((char *) av->top + chunksize(av->top)), 0))
{
  errstr = "double free or corruption (out)";
  goto errout;
}
```

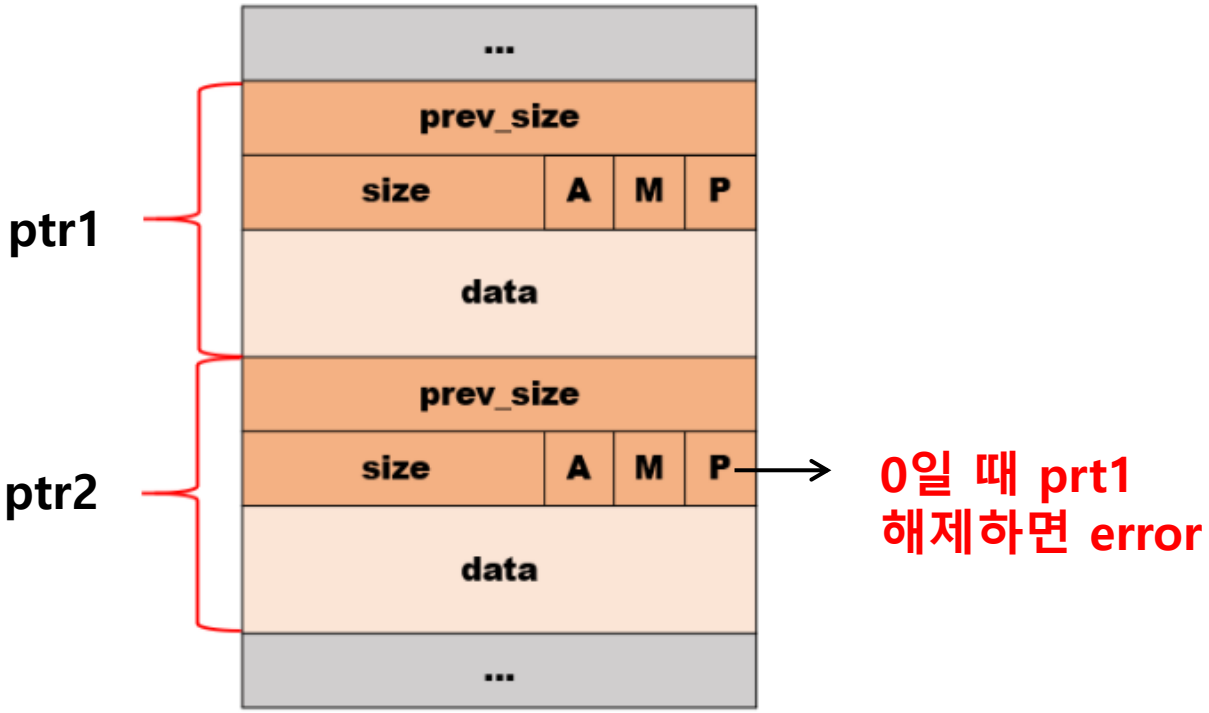
- **double free or corruption (out)**

- 해제하려는 chunk의 size 뒤의 크기가 heap 영역 안에 존재하는지 확인

- 해제하려는 chunk 주소 + chunk size가 heap 영역 밖이면 error 발생

_int_free security check


```
if (__glibc_unlikely (!prev_inuse(nextchunk)))
{
  errstr = "double free or corruption (!prev)";
  goto errout;
}
```



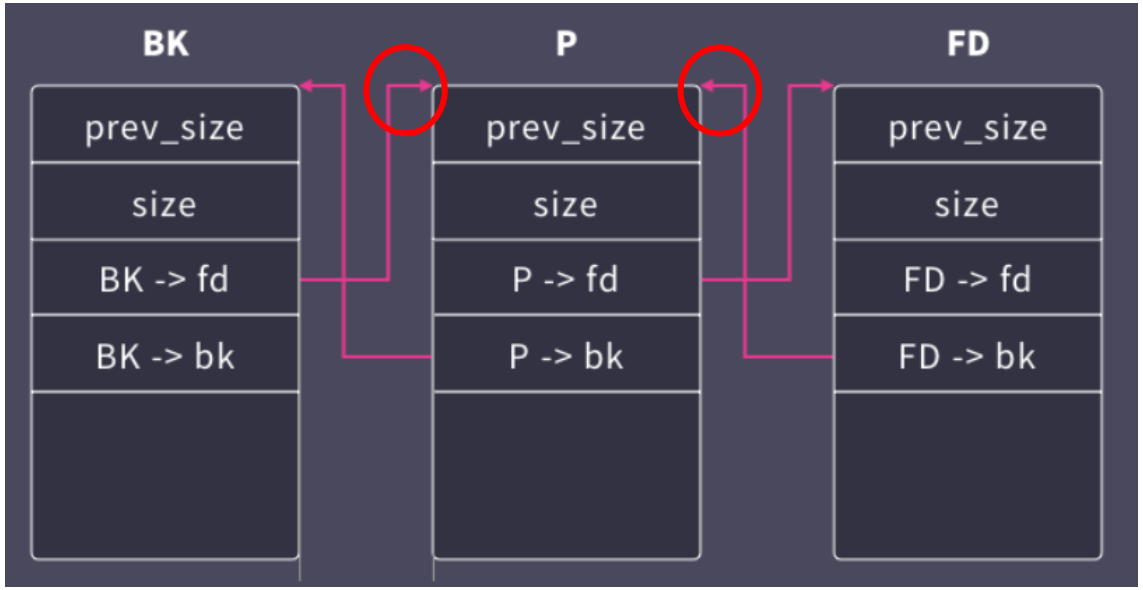
- **double free or corruption (!prev)**

- 해제하려는 chunk 뒤에 존재하는 chunk의 size를 확인하여 Flags 중 prev_inuse bit 설정 여부 확인

- prev_inuse bit가 0인 경우, 이미 해제된 chunk로 간주하여 double free or corruption 오류 발생

unlink security check


```
#define unlink(AV, P, BK, FD) {
          \
    FD = P->fd;                                      \
    BK = P->bk;                                      \
    if (__builtin_expect (FD->bk != P || BK->fd != P, 0))
     \
      malloc_printerr (check_action, "corrupted double-linked
list", P, AV);  \
    else {                                     \
        FD->bk = BK;                                  \
        BK->fd = FD;
```



- **corrupted double-linked list**

- unlik 매크로는 물리적으로 연속적으로 해제된 chunk를 병합하기 위한 매크로

- 현재 청크의 fd의 bk(현재 청크 주소)와 bk의 fd(현재 청크 주소)를 확인하여 현재 청크 주소와 같은지 비교

- 만약 다른 경우라면 error 발생

unlink security check

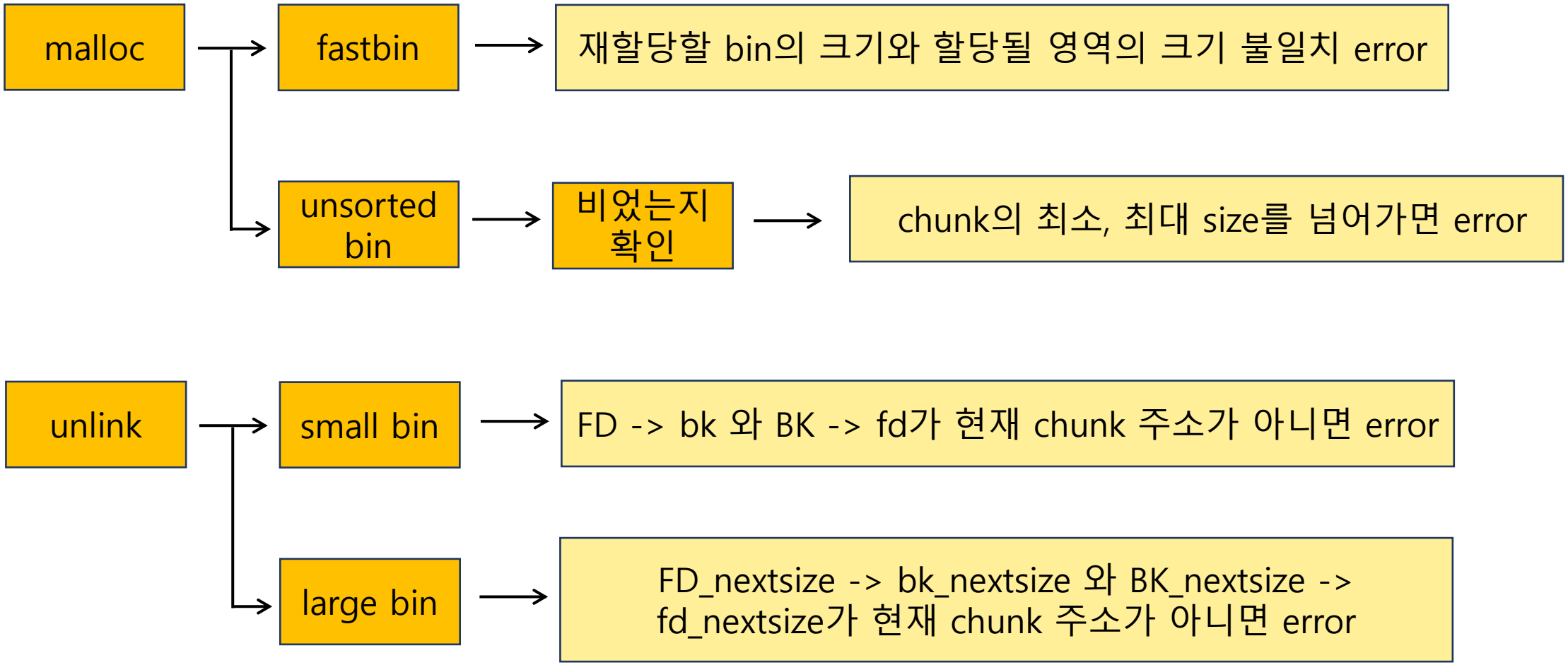

```
#define unlink(AV, P, BK, FD) {
 \
    FD = P->fd;                      \
    BK = P->bk;                      \
    else {                        \
      FD->bk = BK;                    \
      BK->fd = FD;                    \
      if (!in_smallbin_range (P->size)             \
            && __builtin_expect (P->fd_nextsize != NULL, 0)) {          \
        if (__builtin_expect (P->fd_nextsize->bk_nextsize != P, 0)
\
            || __builtin_expect (P->bk_nextsize->fd_nextsize != P, 0))
\
              malloc_printerr (check_action,             \
                               "corrupted double-linked list (not small)",    \
                               P, AV);
```

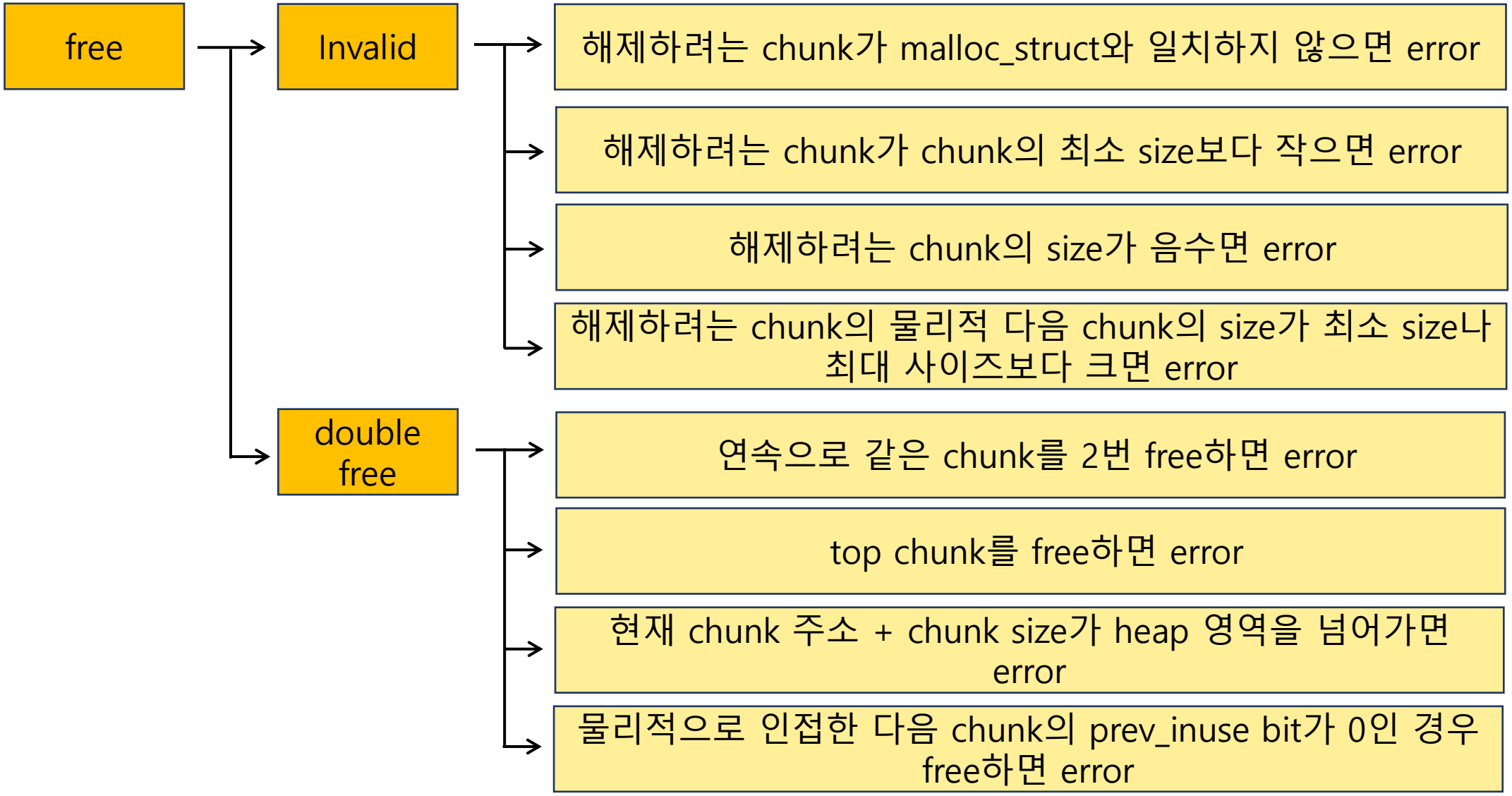
- **corrupted double-linked list (not small)**

- large bin에서 unlink하는 경우 검증하는 코드

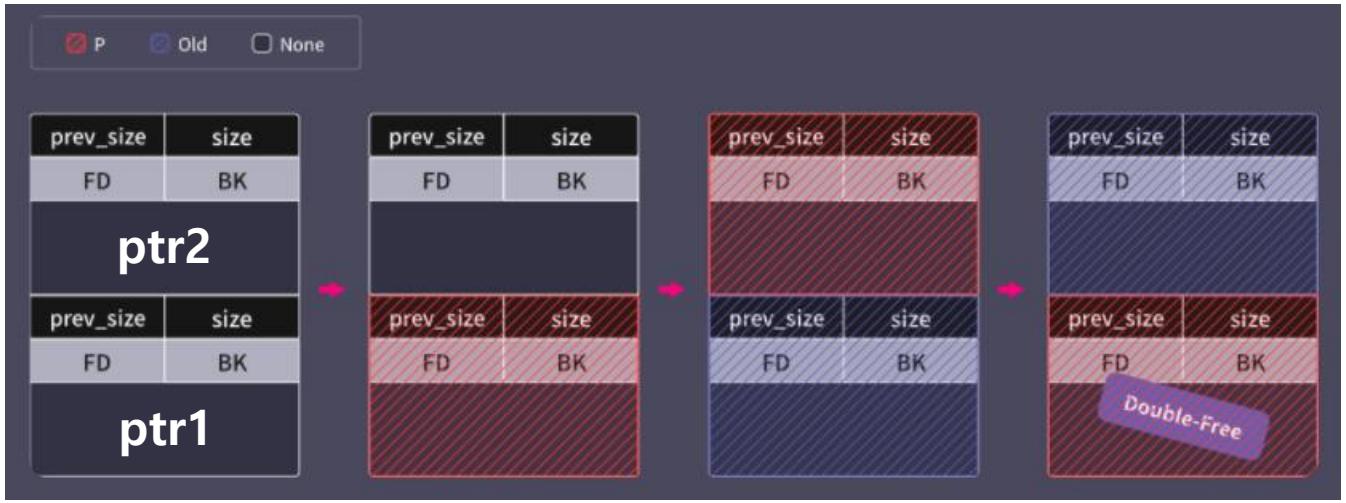
- 이전과 유사하지만 fd와 bk가 아닌 fd_nextsize와 bk_nextsize를 통해 검증

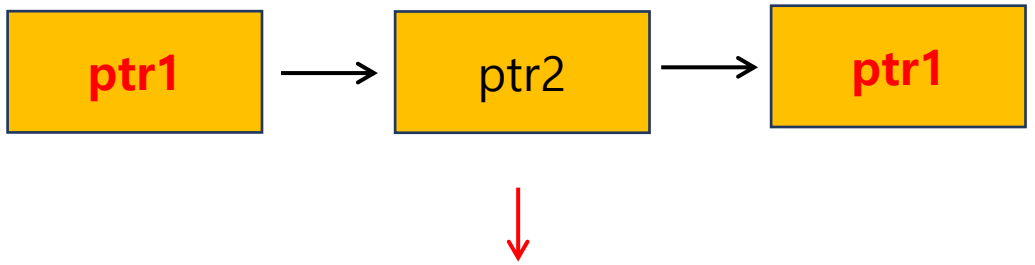
- fd_nextsize -> bk_nextsize와 bk_nextsize -> fd_nextsize가 다른 경우라면 error 발생


malloc
fastbin
재할당할 bin의 크기와 할당될 영역의 크기 불일치 error
unsorted bin
비었는지 확인
chunk의 최소, 최대 size를 넘어가면 error
unlink
small bin
FD -> bk 와 BK -> fd가 현재 chunk 주소가 아니면 error
large bin
FD_nextsize -> bk_nextsize 와 BK_nextsize -> fd_nextsize가 현재 chunk 주소가 아니면 error


free
Invalid
해제하려는 chunk가 malloc_struct와 일치하지 않으면 error
해제하려는 chunk가 chunk의 최소 size보다 작으면 error
해제하려는 chunk의 size가 음수면 error
해제하려는 chunk의 물리적 다음 chunk의 size가 최소 size나 최대 사이즈보다 크면 error
double free
연속으로 같은 chunk를 2번 free하면 error
top chunk를 free하면 error
현재 chunk 주소 + chunk size가 heap 영역을 넘어가면 error
물리적으로 인접한 다음 chunk의 prev_inuse bit가 0인 경우 free하면 error

Double Free 취약점





재할당을 할 경우
2개의 객체가 같은 메모리를 가리킨다.

- **Double Free 취약점**

- 연속적으로 free를 하지않고 중간에 다른 chunk를 free하고 다시 chunk를 free하면 Double free 취약점이 발생한다.

- 두 개의 객체가 하나의 메모리를 사용할 수 있게 된다.

UAF 취약점


```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
        char* a = (char*)malloc(0x100);
        char* b = (char*)malloc(0x20);

        free(a);
        free(b);

        char* a2 = (char*)malloc(0x100);
        char* b2 = (char*)malloc(0x20);

        strcpy(a2,"hello, a");
        strcpy(b2,"hello, b");

        printf("a: %s\nb: %s\n",a,b);

        return 0;
}
```

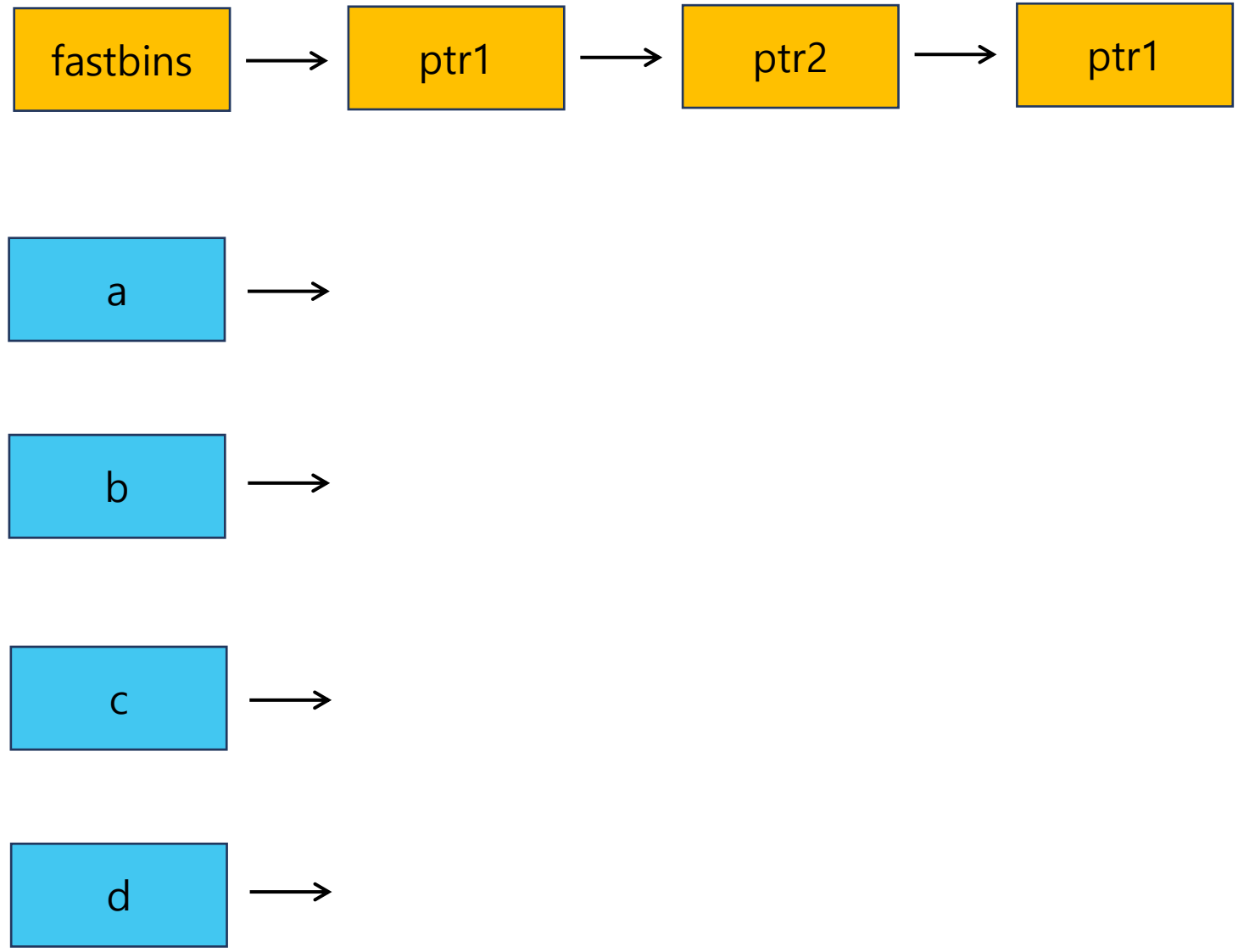
```
root@ubuntu:/home/hamacu6/chall# ./uaf
a: hello, a
b: hello, b
```

- **Use After Free 취약점**

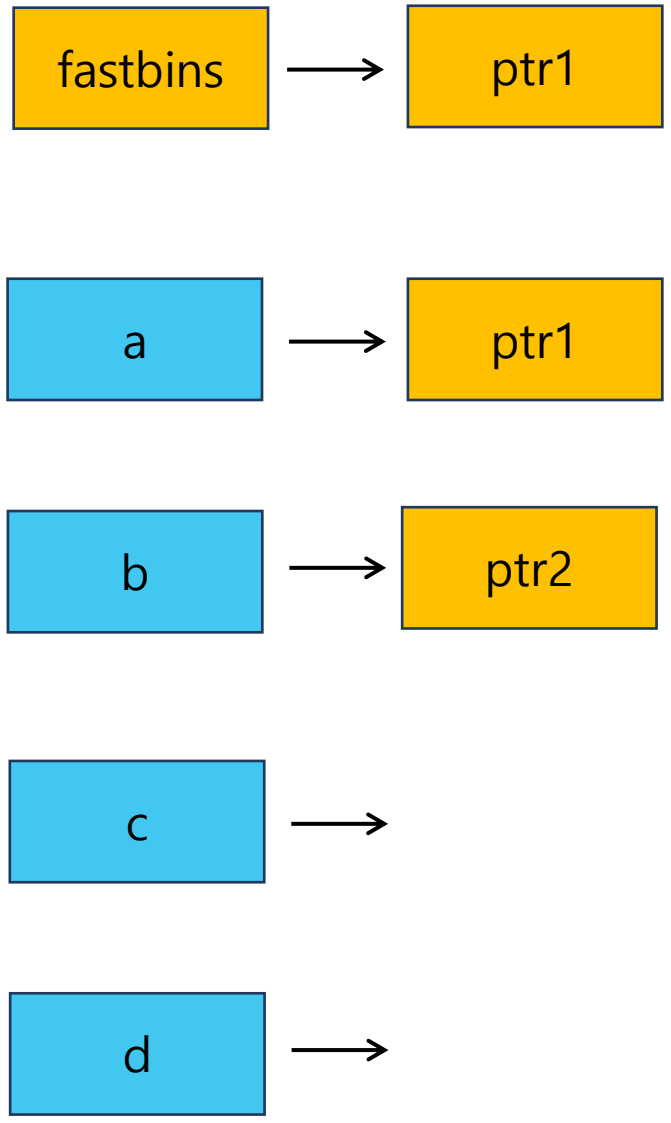
- Heap 영역에서 할당된 공간을 free로 영역을 해제하고, 메모리를 다시 할당 시 같은 공간을 재사용 하면서 생기는 취약점

- 이미 해제된 포인터여도, 재할당이 되면 같은 곳을 가리키고 있어서 두 개의 객체가 하나의 메모리를 사용

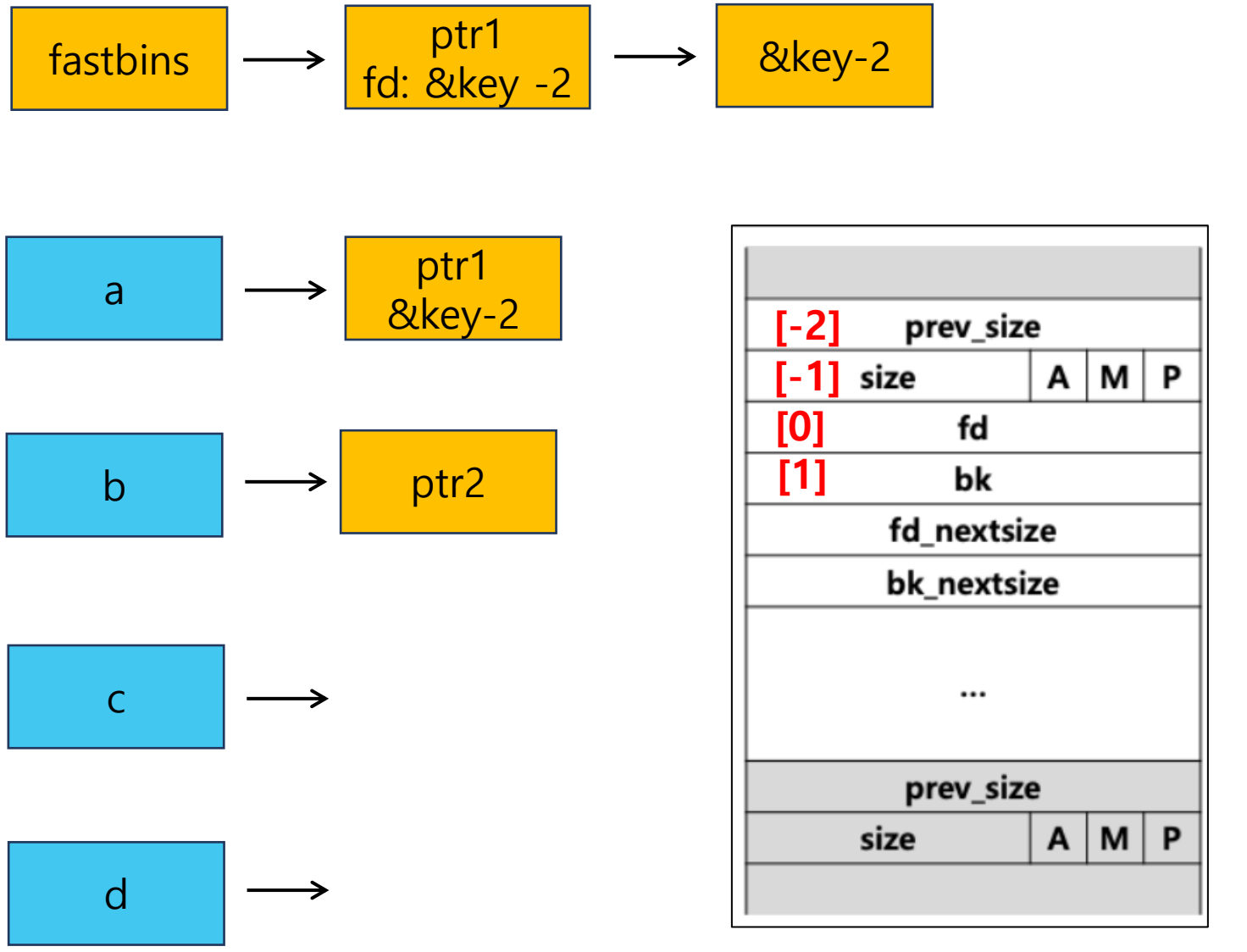
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char key = 'a';

int main()
{
        char* a = (char*)malloc(0x20);
        char* b = (char*)malloc(0x20);

        free(a);
        free(b);
        free(a);

        a = (char*)malloc(0x20);
        b = (char*)malloc(0x20);

        a[0] = &key-2;
        char* c = (char*)malloc(0x20);
        char* d = (char*)malloc(0x20);
        d[0] = 'k';

        if(key == 'k')
                printf("sucees fastbin dup!\n");
        else
                printf("fail fastbin dup :(\n");


        return 0;
}
```



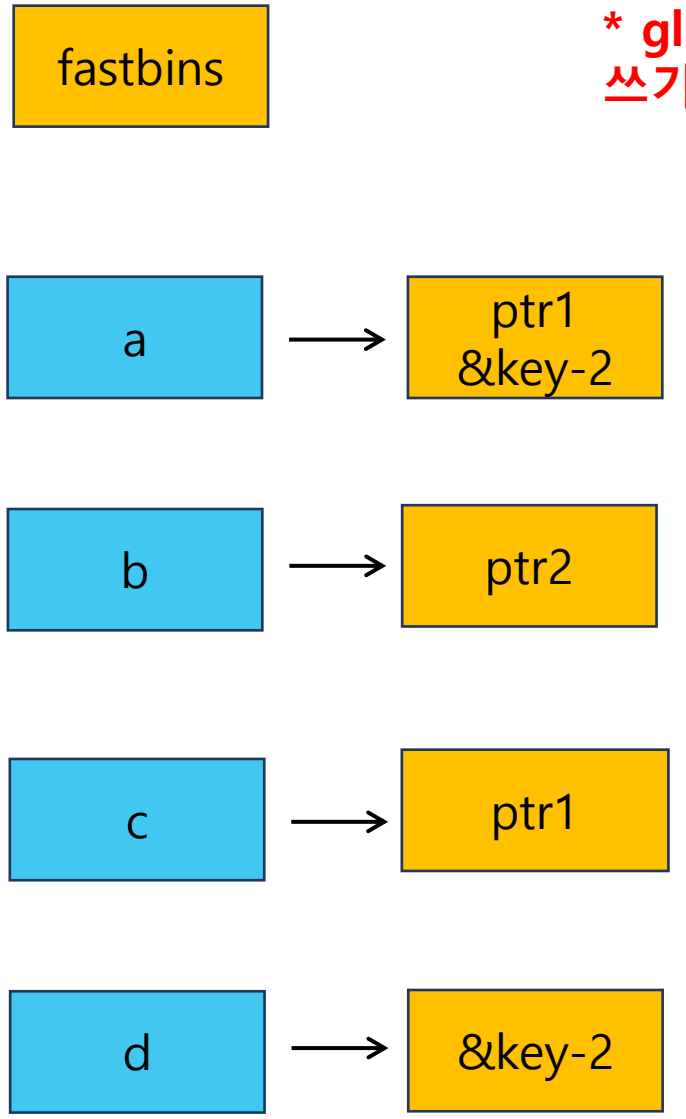
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char key = 'a';

int main()
{
        char* a = (char*)malloc(0x20);
        char* b = (char*)malloc(0x20);

        free(a);
        free(b);
        free(a);

        a = (char*)malloc(0x20);
        b = (char*)malloc(0x20);

        a[0] = &key-2;
        char* c = (char*)malloc(0x20);
        char* d = (char*)malloc(0x20);
        d[0] = 'k';

        if(key == 'k')
                printf("sucees fastbin dup!\n");
        else
                printf("fail fastbin dup :(\n");


        return 0;
}
```



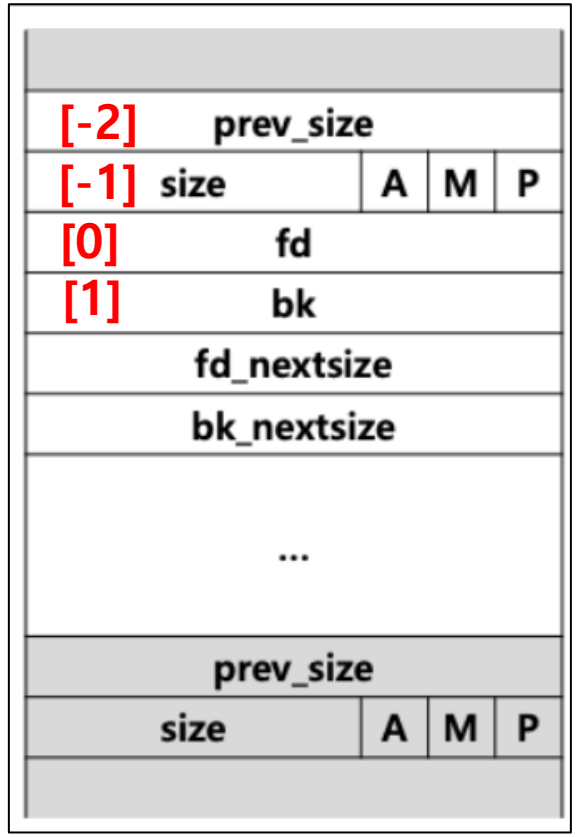
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char key = 'a';

int main()
{
        char* a = (char*)malloc(0x20);
        char* b = (char*)malloc(0x20);

        free(a);
        free(b);
        free(a);

        a = (char*)malloc(0x20);
        b = (char*)malloc(0x20);

        a[0] = &key-2;
        char* c = (char*)malloc(0x20);
        char* d = (char*)malloc(0x20);
        d[0] = 'k';

        if(key == 'k')
                printf("sucees fastbin dup!\n");
        else
                printf("fail fastbin dup :(\n");


        return 0;
}
```



```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char key = 'a';

int main()
{
        char* a = (char*)malloc(0x20);
        char* b = (char*)malloc(0x20);

        free(a);
        free(b);
        free(a);

        a = (char*)malloc(0x20);
        b = (char*)malloc(0x20);

        a[0] = &key-2;
        char* c = (char*)malloc(0x20);
        char* d = (char*)malloc(0x20);
        d[0] = 'k';

        if(key == 'k')
                printf("sucees fastbin dup!\n");
        else
                printf("fail fastbin dup :(\n");


        return 0;
}
```



* glibc 2.29이상부터는 tcache를 쓰기 때문에 double free 발생 가능



```
#define have_fastchunks(M)     (((M)->flags & FASTCHUNKS_BIT) == 0)
else
{
    idx= largebin_index (nb);
    if (have_fastchunks (av))
        malloc_consolidate (av);
}
```

- **fastbin dup consolidate**

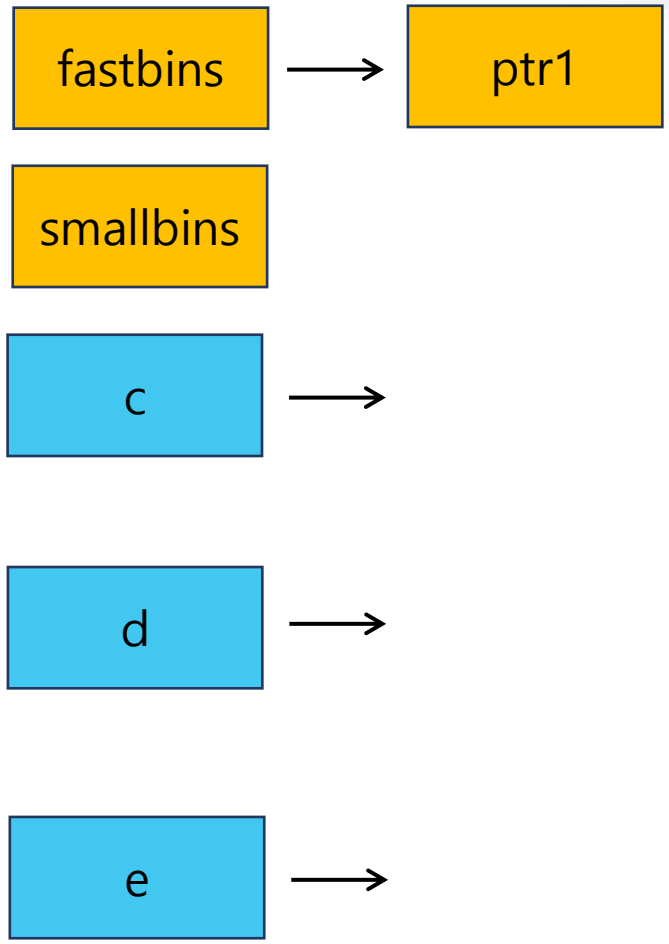
- fastbin에 들어간 힙 청크들을 병합시켜 smallbin에 넣어 Double Free 검증을 우회하는 기법

- malloc 함수 내부에서 largebin 크기의 할당 요청이 들어올 때 have_fastchunks 매크로를 통해 fastbin에 할당된 청크가 존재하는지 확인하고, 존재한다면 malloc_consolidate 함수를 통해 존재하는 청크들을 병합 (fastbin에 있는 chunk가 small bin으로 들어간다.)

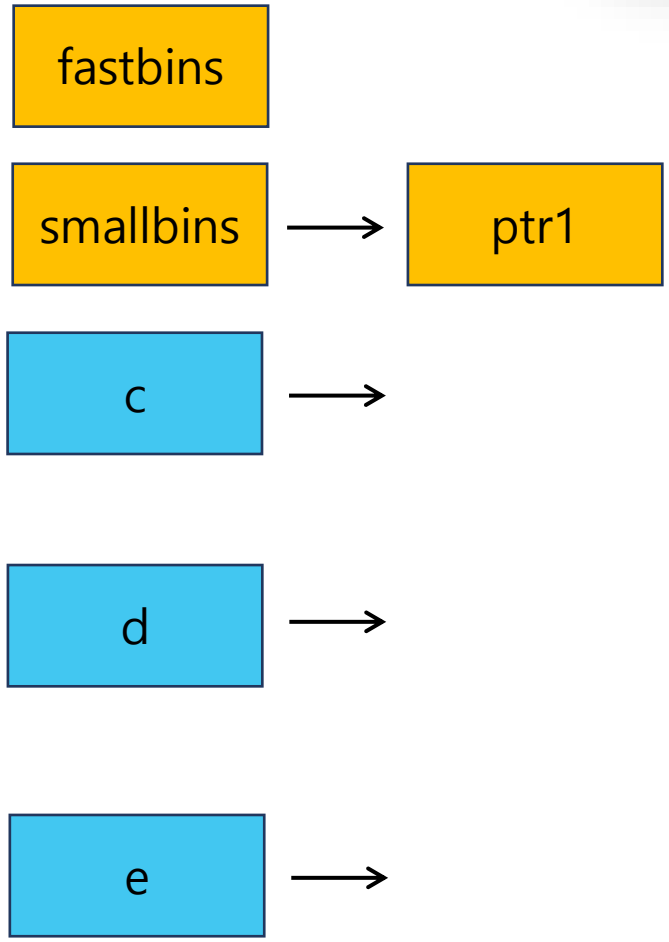
- fastbin dup처럼 fd를 조작하여 원하는 주소의 값 조작 가능

fastbin dup

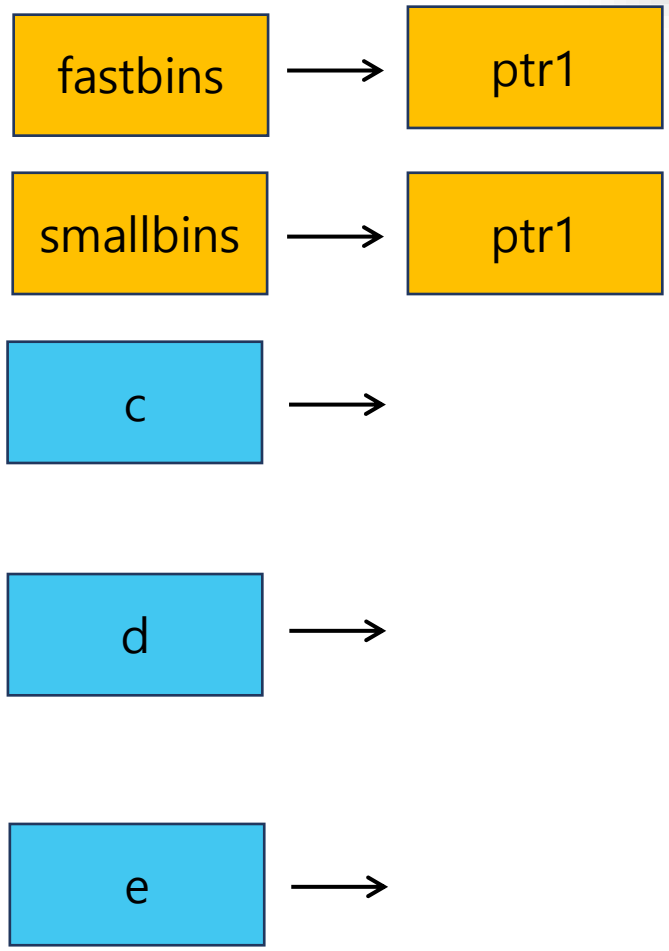

```
char key = 'a';

int main()
{
        char* a = (char*)malloc(0x20);
        char* a1 = (char*)malloc(0x20);

        free(a);

        char* b = (char*)malloc(0x400);

        free(a);

        char* c = (char*)malloc(0x20);
        c[0]=&key-2;
        char* d = (char*)malloc(0x20);
        char* e = (char*)malloc(0x20);
        e[0] = 'k';

        if(key = 'k')
                printf("success dup!\n");
        else
                printf("fail dup:(\n");

        return 0;
}
```



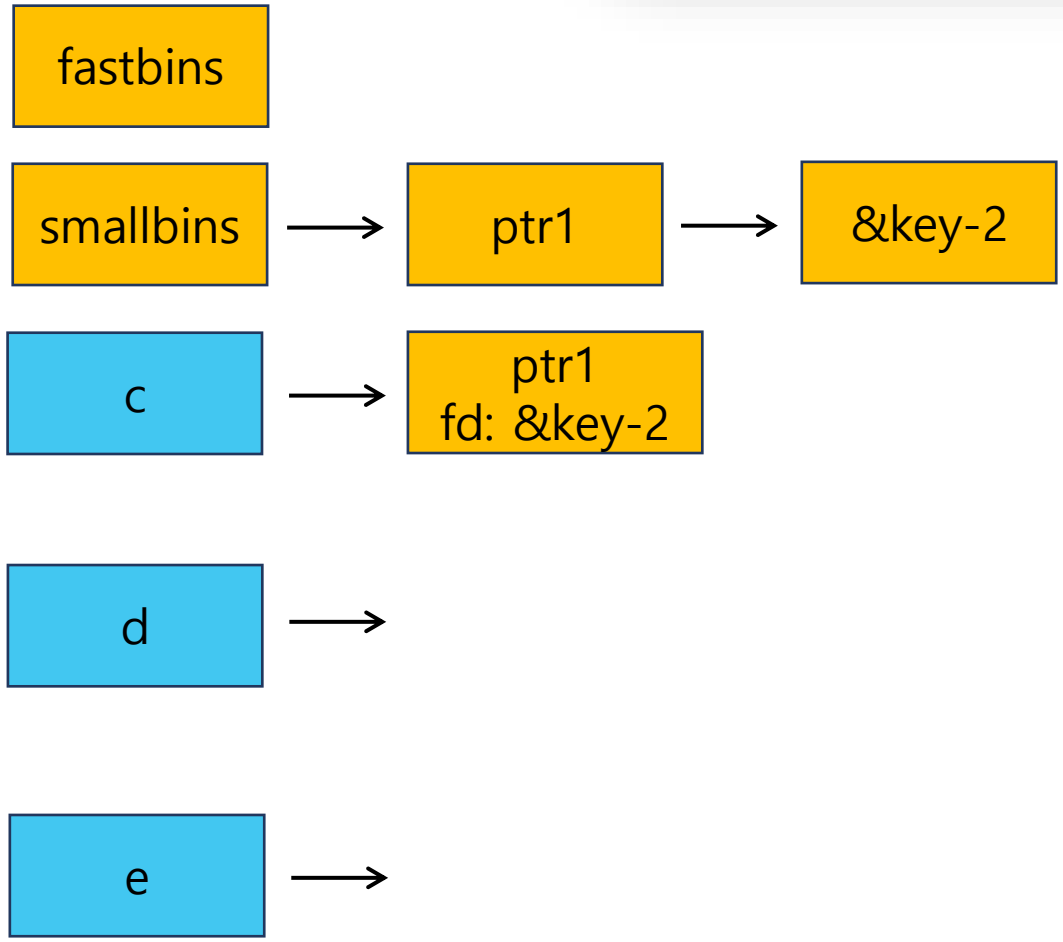
```
char key = 'a';

int main()
{
        char* a = (char*)malloc(0x20);
        char* a1 = (char*)malloc(0x20);

        free(a);

        char* b = (char*)malloc(0x400);

        free(a);

        char* c = (char*)malloc(0x20);
        c[0]=&key-2;
        char* d = (char*)malloc(0x20);
        char* e = (char*)malloc(0x20);
        e[0] = 'k';

        if(key = 'k')
                printf("success dup!\n");
        else
                printf("fail dup:(\n");

        return 0;
}
```



```
char key = 'a';

int main()
{
        char* a = (char*)malloc(0x20);
        char* a1 = (char*)malloc(0x20);

        free(a);

        char* b = (char*)malloc(0x400);

        free(a);

        char* c = (char*)malloc(0x20);
        c[0]=&key-2;
        char* d = (char*)malloc(0x20);
        char* e = (char*)malloc(0x20);
        e[0] = 'k';

        if(key = 'k')
                printf("success dup!\n");
        else
                printf("fail dup:(\n");

        return 0;
}
```



```
char key = 'a';

int main()
{
        char* a = (char*)malloc(0x20);
        char* a1 = (char*)malloc(0x20);

        free(a);

        char* b = (char*)malloc(0x400);

        free(a);

        char* c = (char*)malloc(0x20);
        c[0]=&key-2;
        char* d = (char*)malloc(0x20);
        char* e = (char*)malloc(0x20);
        e[0] = 'k';

        if(key = 'k')
                printf("success dup!\n");
        else
                printf("fail dup:(\n");

        return 0;
}
```



```
char key = 'a';

int main()
{
        char* a = (char*)malloc(0x20);
        char* a1 = (char*)malloc(0x20);

        free(a);

        char* b = (char*)malloc(0x400);

        free(a);

        char* c = (char*)malloc(0x20);
        c[0]=&key-2;
        char* d = (char*)malloc(0x20);
        char* e = (char*)malloc(0x20);
        e[0] = 'k';

        if(key = 'k')
                printf("success dup!\n");
        else
                printf("fail dup:(\n");

        return 0;
}
```