Heap Internals

Chester Rebeiro

Indian Institute of Technology Madras

Heap

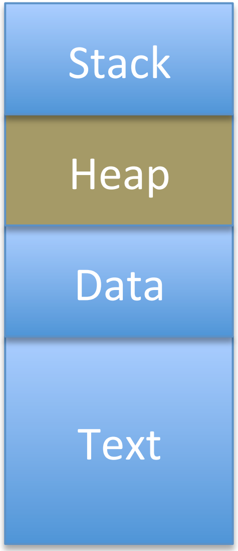
- Just a pool of memory used for dynamic memory allocation

```
int main()
{
    char * buffer = NULL;

    /* allocate a 0x100 byte buffer */
    buffer = malloc(0x100);

    /* read input and print it */
    fgets(stdin, buffer, 0x100);
    printf("Hello %s!\n", buffer);

    /* destroy our dynamically allocated buffer */
    free(buffer);
    return 0;
}
```



Heap vs Stack

- Heap
  - Slow
  - Manually done by free and malloc
  - Used for objects, large arrays, persistent data (across function calls)
- Stack
  - Fast
  - Automatically done by compiler
  - Temporary data store

Heap Management

- Several different types of implementations
  - Doug Lea’s forms the base for many
  - glibc uses ptmalloc2 or ptmalloc3
  - Others include
    - tcmalloc (from Google)
    - jemalloc (used in Android)
    - nedmalloc
    - Hoard
  - Trade off between speed of memory management vs fragmented memory
  - Other aspects include scalability, multi-threaded support

ftp://g.oswego.edu/pub/misc/malloc.c

ptmalloc 2

- Used in glibc
- Internally uses brk and mmap syscalls to obtain memory from the OS
- Arena:
  - main arena
  - Per-thread arena (dynamic arena)
  - Each arena can have multiple heaps (each heap is of 132 KB)
- Heaps
  - Split into memory chunks of different sizes and used depending on how malloc and free are invoked
- Memory chunks
  - Of two types: free chunk and allocated chunk
  - Free chunks stored in a linked list

Arena

```
void* threadFunc(void* arg)
{
        char* addr = (char*) malloc(1000);
        free(addr);
}

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

        addr = (char*) malloc(1000);
        free(addr);
        ret = pthread_create(&t1, NULL,
           threadFunc, NULL);
        ret = pthread_join(t1, &s);
        return 0;
}
```

Process starts with no heap segment

Arena

```
void* threadFunc(void* arg)
{
      char* addr = (char*) malloc(1000);
      free(addr);
}

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

      addr = (char*) malloc(1000);
      free(addr);
      ret = pthread_create(&t1, NULL,
          threadFunc, NULL);
      ret = pthread_join(t1, &s);
      return 0;
}
```

Arena of size 132 KB created on the first malloc invocation.
The arena is created by invoking the system call brk.
Future allocations use this arena until it gets completely used up. In which case the arena can grow or shrink.

```
chester@optiplex:~$ cat /proc/1897/maps
00400000-00401000 r-xp 00000000 08:07 2490714        ..a.out
00600000-00601000 r--p 00000000 08:07 2490714        ..a.out
00601000-00602000 rw-p 00001000 08:07 2490714        ..a.out
00602000-00623000 rw-p 00000000 00:00 0              [heap]
7ffff77f3000-7ffff79b1000 r-xp 00000000 08:06 161656  /lib/x86_64-l
7ffff79b1000-7ffff7bb1000 ---p 001be000 08:06 161656  /lib/x86_64-l
7ffff7bb1000-7ffff7bb5000 r--p 001be000 08:06 161656  /lib/x86_64-l
7ffff7bb5000-7ffff7bb7000 rw-p 001c2000 08:06 161656  /lib/x86_64-l
```

Arena

```
void* threadFunc(void* arg)
{
      char* addr = (char*) malloc(1000);
      free(addr);
}

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

      addr = (char*) malloc(1000);
      free(addr);
      ret = pthread_create(&t1, NULL,
          threadFunc, NULL);
      ret = pthread_join(t1, &s);
      return 0;
}
```

Even after free, the arena will still exist.

```
chester@optiplex:~$ cat /proc/1897/maps
00400000-00401000 r-xp 00000000 08:07 2490714          ..a.out
00600000-00601000 r--p 00000000 08:07 2490714          ..a.out
00601000-00602000 rw-p 00001000 08:07 2490714          ..a.out
00602000-00623000 rw-p 00000000 00:00 0                [heap]
7ffff77f3000-7ffff79b1000 r-xp 00000000 08:06 161656  /lib/x86_64-linu
7ffff79b1000-7ffff7bb1000 ---p 001be000 08:06 161656  /lib/x86_64-linu
7ffff7bb1000-7ffff7bb5000 r--p 001be000 08:06 161656  /lib/x86_64-linu
7ffff7bb5000-7ffff7bb7000 rw-p 001c2000 08:06 161656  /lib/x86_64-linu
```

Arena

```
void* threadFunc(void* arg)
{
      char* addr = (char*) malloc(1000);
      free(addr);
}

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

      addr = (char*) malloc(1000);
      free(addr);
      ret = pthread_create(&t1, NULL,
          threadFunc, NULL);
      ret = pthread_join(t1, &s);
      return 0;
}
```

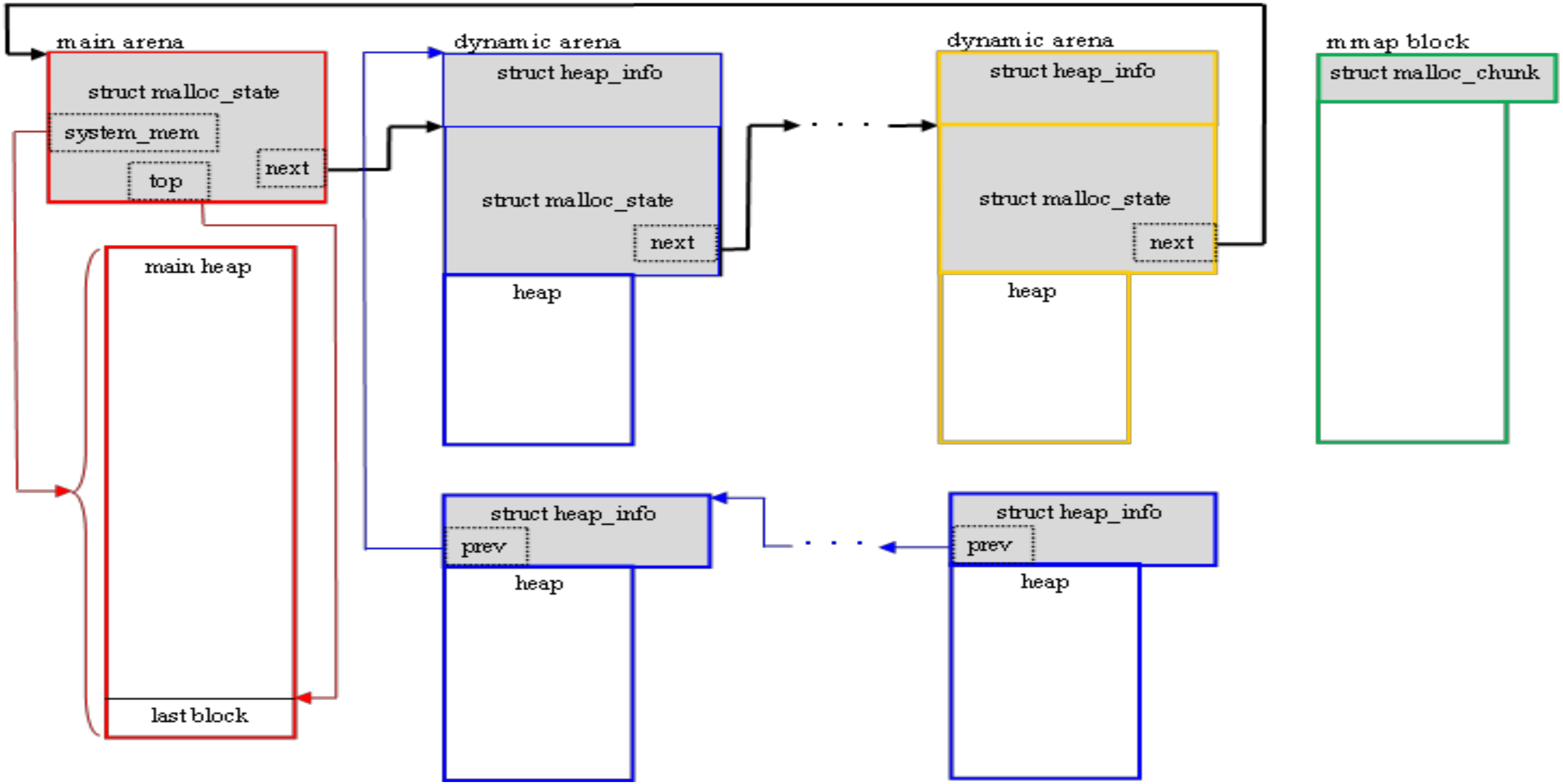
When threads are created, it may lead to new arenas being created. These new arenas are also of 132 KB and obtained by invoking mmap on the OS.

```
chester@optiplex:~$ cat /proc/2283/maps
00400000-00401000 r-xp 00000000 08:07              a.out
00600000-00601000 r--p 00000000 08:07 2490714      a.out
00601000-00602000 rw-p 00001000 08:07 2490714      a.out
00602000-00623000 rw-p 00000000 00:00 0            [heap]
7ffff0000000-7ffff0021000 rw-p 00000000 00:00 0
7ffff0021000-7ffff4000000 ---p 00000000 00:00 0
7ffff6ff2000-7ffff6ff3000 ---p 00000000 00:00 0
7ffff6ff3000-7ffff77f3000 rw-p 00000000 00:00 0 [stack:2330]
```

The Whole Structure

- Each arena can have multiple heaps (possibly non-contiguous)
  - One or more arenas present in a process.
  - struct malloc_state : manages the arena. aka. Arena header
  - struct heap_info: manages specific heaps within the arena
- Each heap can have multiple memory chunks
  - These chunks store data and allocated up on user request
  - struct malloc_chunk : manages a chunk of memory
- Types of memory chunks
  - Allocated chunk
  - Free chunk
  - Top chunk : contains the unused memory allocated to the heap by the OS but not yet allocated to hold any data.
  - Last remainder chunk : last chunk that was split

Ptmalloc: the whole structure



More about Arenas

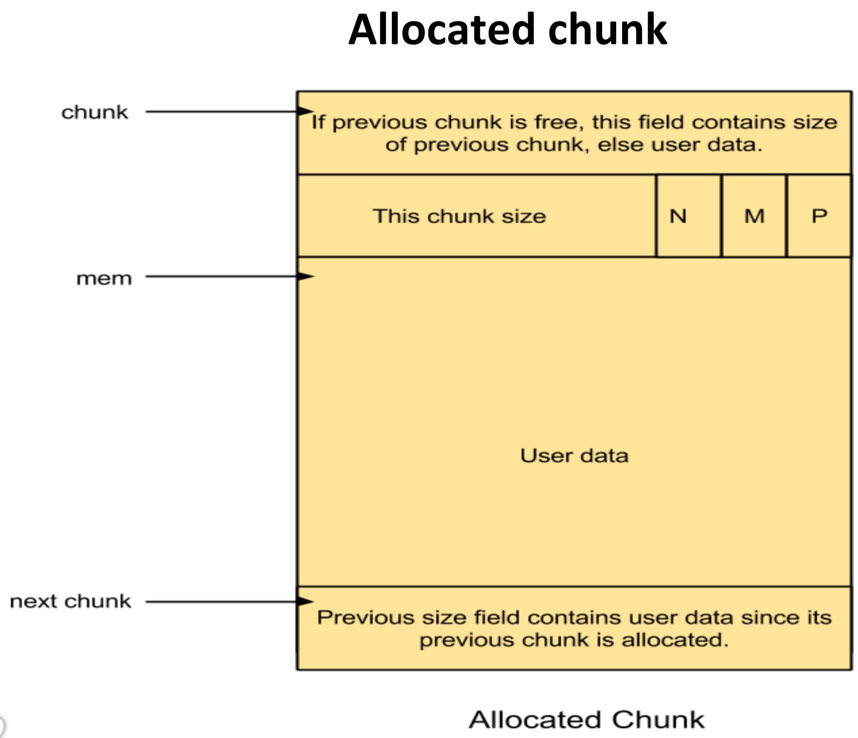
- Maximum number of Arenas restricted by the number of cores in the system:
  - 32 bit: #MaxArenas = 2 x Num.ofCores
  - 64 bit: #MaxArenas = 8 x Num.ofCores
  - If num. of threads is less than #MaxArenas, then we get quick mallocs and frees as there is no contention
- One arena can service one memory request at a time (i.e. one malloc / free)
- If more threads are present than MaxArenas then multiple threads need to share one arena.
  - This leads to contention and hence slower mallocs and frees
  - Structure ***malloc_state***, contains all the management information for an arena

Points to Ponder

- Maximum number of Arenas restricted by the number of cores in the system:
  - 32 bit: #MaxArenas = 2 x Num.ofCores
  - 64 bit: #MaxArenas = 8 x Num.ofCores

Why restrict the number of Arenas?
Why not have as many Arenas as the number of threads present?

Allocated Chunk



P : previous chunk in use (PREV_INUSE bit)

If P=0, then the word before this contains the size of the previous chunk.

The very first chunk always has this bit set
Preventing access to non-existent memory.

M : set if chunk was obtained with mmap

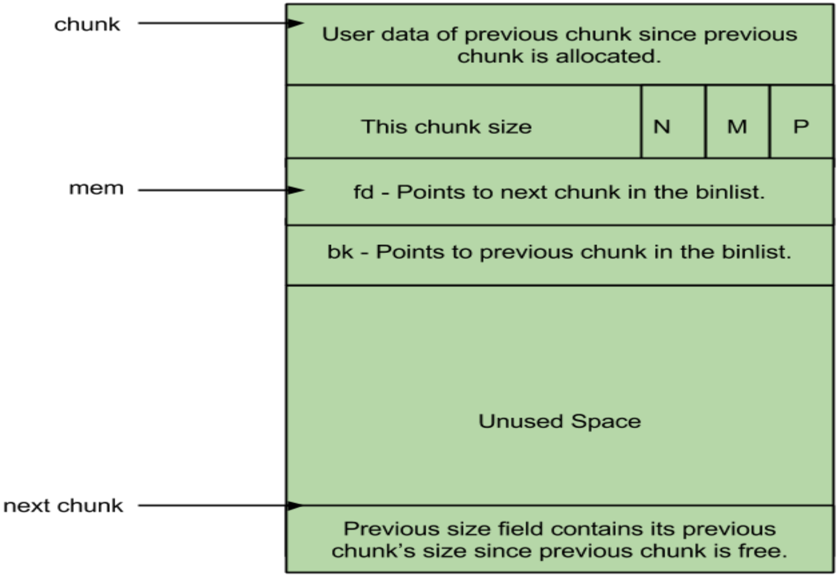
N : set if chunk belongs to thread arena

mem. Is the pointer returned by malloc.
chunk. Is the pointer to metadata for malloc

User data size for malloc(n) is
N = 8 + (n/8)*8 bytes.
Total size of chunk is N+8 bytes

Free Chunk

Free chunk



Free Chunk

P : previous chunk in use (PREV_INUSE bit)

If P=0, then the word before this contains the size of the previous chunk.

The very first chunk always has this bit set
Preventing access to non-existent memory.

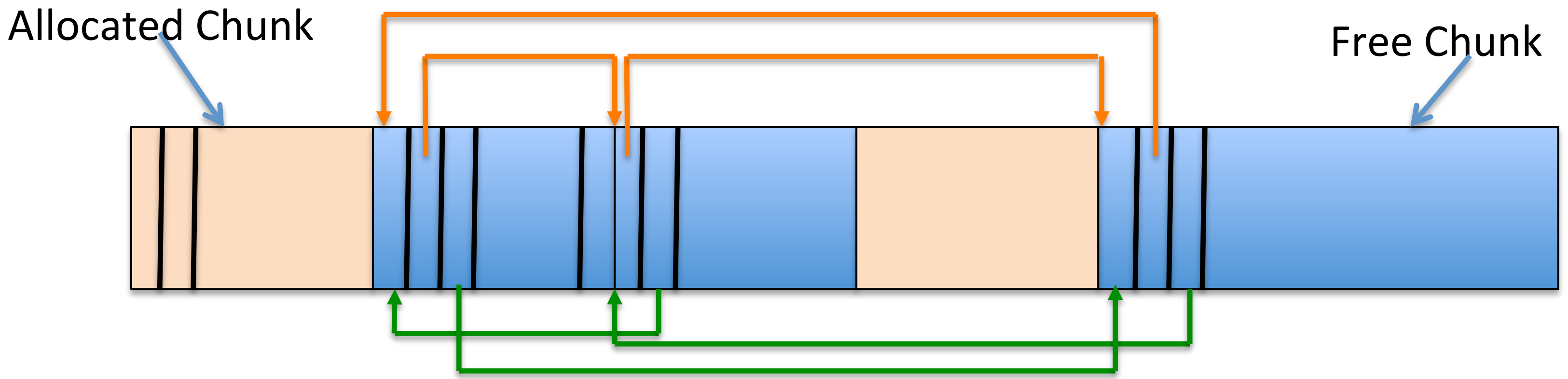
M : set if chunk was obtained with mmap

N : set if chunk belongs to thread arena

mem. Is the pointer returned by malloc.
chunk. Is the pointer to metadata for malloc

On 32 bit machine,
User data size for malloc(n) is
N = 8 + (n/8)*8 bytes.
Total size of chunk is N+8 bytes

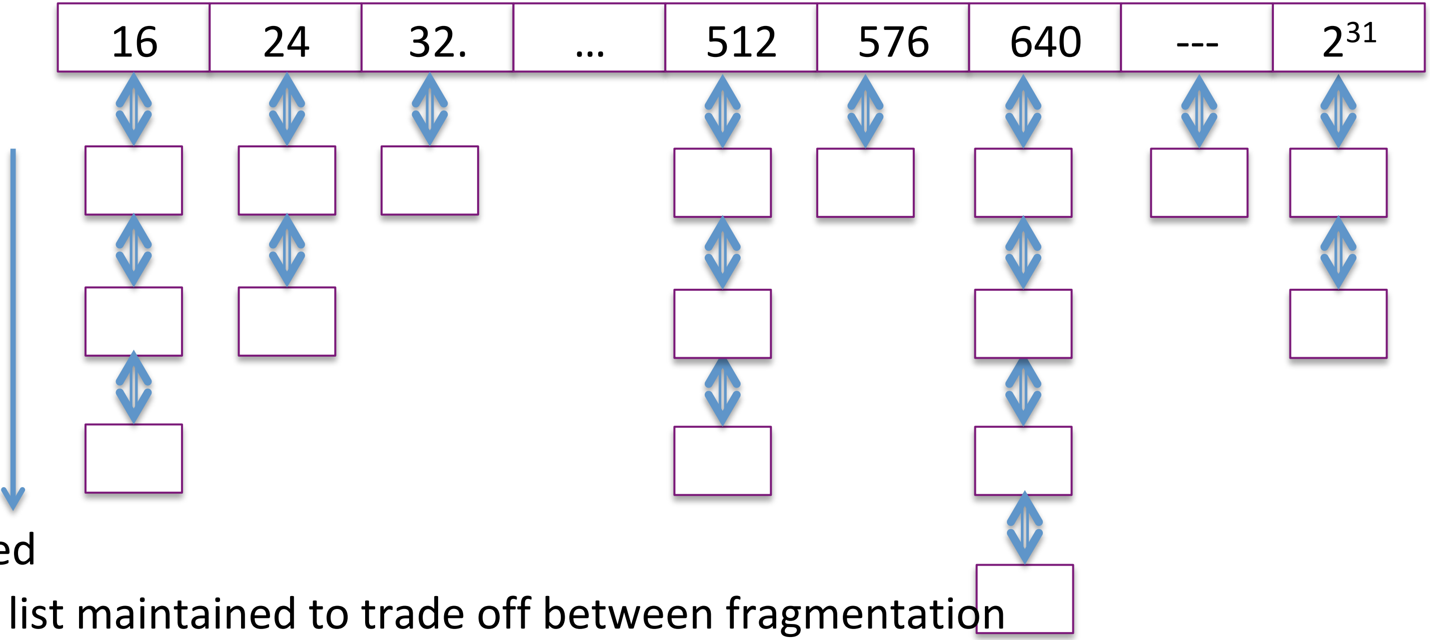
List of Free Chunks



Free chunks (blue) are maintained in a linked list.

The linked list is called bins and can vary in size and characteristic

Binning



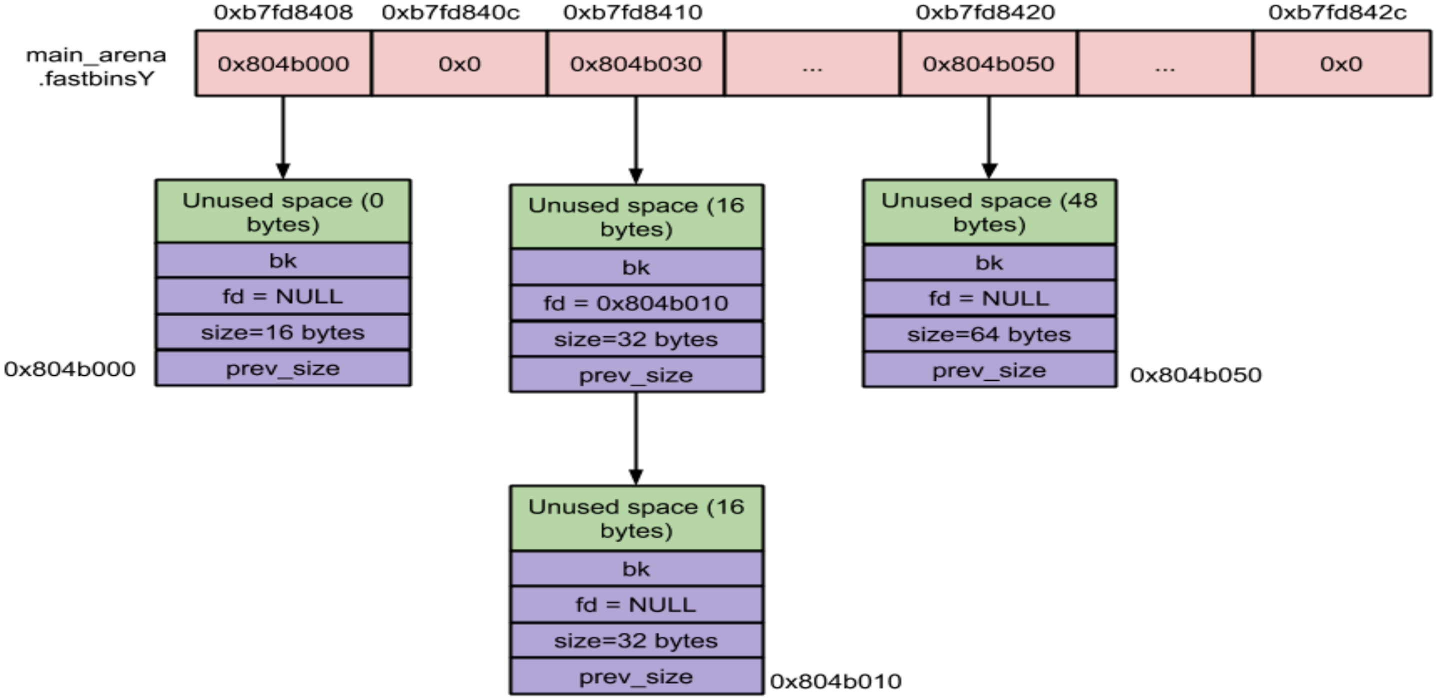
Different list maintained to trade off between fragmentation and speed of malloc / free.

Types of Bins

Fast Bins | Unsorted Bins | Small Bins | Large Bins | Top Chunk | Last Reminder Chunk

Single link list
8 byte chunks; defined by NFASTBINS in malloc.c (12 of them)
(16, 24, 32, …., 80)
No coalescing (could result in fragmentation; but speeds up free)
LIFO
Pointer to list maintained in the arena (malloc_info)

Fastbin Example



Example of Fast Binning

x and y end up in the same bin.

```
void main()
{
        char *x, *y;

        x = malloc(15);
        printf("x=%08x\n", x);

        free(x);

        y = malloc(13);

        printf("y=%08x\n", y);

        free(y);


}
```

```
x=09399008
y=09399008
```

x and y end up in different bins.

```
void main()
{
        char *x, *y;

        x = malloc(8);
        printf("x=%08x\n", x);

        free(x);

        y = malloc(13);

        printf("y=%08x\n", y);

        free(y);


}
```

```
x=08564008
y=08564018
```

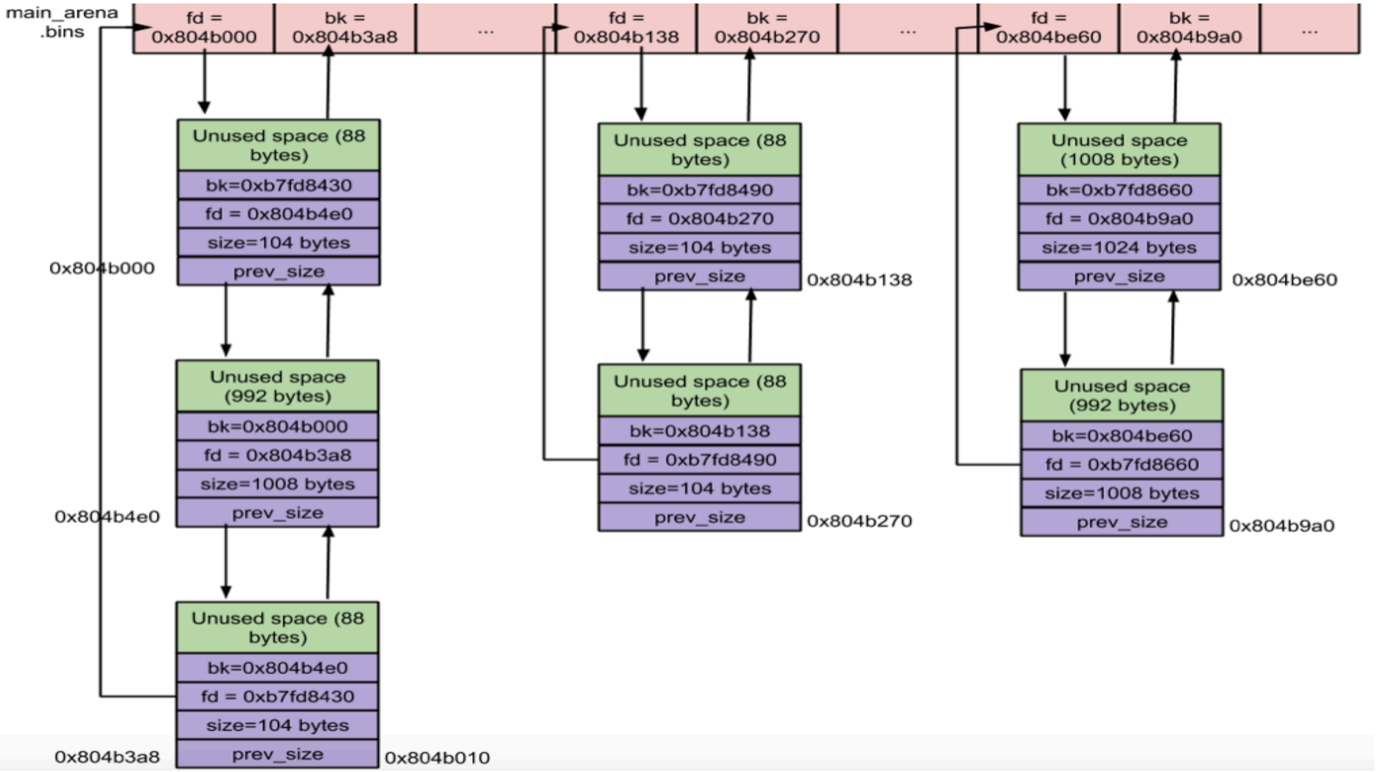
Types of Bins

Fast Bins | **Unsorted Bins** | Small Bins | Large Bins | Top Chunk | Last Reminder Chunk

Single link list
8 byte chunks
ntation; but speeds up free)

1 bin
Doubly link list
Chunks of any size
Uses the first chunk that fits.
When a chunk is freed, it is first added here.
Helps reuse recently used chunks

Unsorted Bin



CR

Glib's first fit allocator

First Fit scheme used for allocating chunk

```
int main()
{
        char* a = malloc(512);
        char* b = malloc(256);
        char* c;

        printf("Address of A: %p\n", a);
        printf("Address of B: %p\n", b);
        strcpy(a, "This is A\n");
        printf("first allocation %p points to %s\n", a, a);
        printf("Freeing the first one...\n");
        free(a);

        c = malloc(50);
        strcpy(c, "This is C\n");
        printf("Address of C: %p\n", c);
        printf("Address of A is %p it contains %s\n", a, a);
}
```

```
chester@aahalya:~/sse/malloc$ ./a.out
Address of A: 0x9b10008
Address of B: 0x9b10210
first allocation 0x9b10008 points to This is A

Freeing the first one...
Address of C: 0x9b10008
Address of A is 0x9b10008 it contains This is C
```

Allocating a memory chunk of 512 bytes

Now freeing it

Now allocating another chunk < 512 bytes.

The first free chunk available corresponds to the freed 'a'. So, 'c' gets allocated the same address as 'a'

https://github.com/shellphish/how2heap (first_fit.c)

Types of Bins

Fast Bins | Unsorted Bins | **Small Bins** | Large Bins | Top Chunk | Last Reminder Chunk

Single link list

1 bin

Doubly link list

Chunks of any size

Helps reuse recently

62 bins ; less than 512 bytes

Chunks of 8 bytes

Circular doubly linked list – because chunks are unlinked from the middle of the list

Coalescing – join to free chunks which are adjacent to each other

FIFO

Types of Bins

Fast Bins | Unsorted Bins | Small Bins | **Large Bins** | Top Chunk | Last Reminder Chunk

63 bins ;

First 32 bins are 64 bytes apart
Next 16 bins are 512 bytes apart
Next 8 bins are 4096 bytes apart
Next 4 bins are 32768 bytes apart
Next 2 bins are 262144 bytes apart
1 bin of remaining size

Each bin is circular doubly linked list
Since contents of bin are not of same size; they are stored in decreasing order of size

Coalescing – join to free chunks which are adjacent to each other

Single link list
1 bin
Doubly link list
Chunks of any size
Helps reuse recent
nks are unlinked from
re adjacent to each
FIFO

CR

Types of Bins

Fast Bins | Unsorted Bins | Small Bins | Large Bins | Top Chunk | Last Reminder Chunk

Single link list

1 bin

Doubly link list

Chunks of any size

Helps reuse recently

Top of the arena;
Does not belong to any bin;
Used to service requests when there is no free chunk available.

If the top chunk is larger than the requested memory it is split into two: user chunk (used for the requests memory and last reminder chunk which becomes the new top chunk)

If the top chunk is smaller than the requested chunk
It grows by invoking the brk() or sbrk() system call

Which defines the end of the process’ data segment

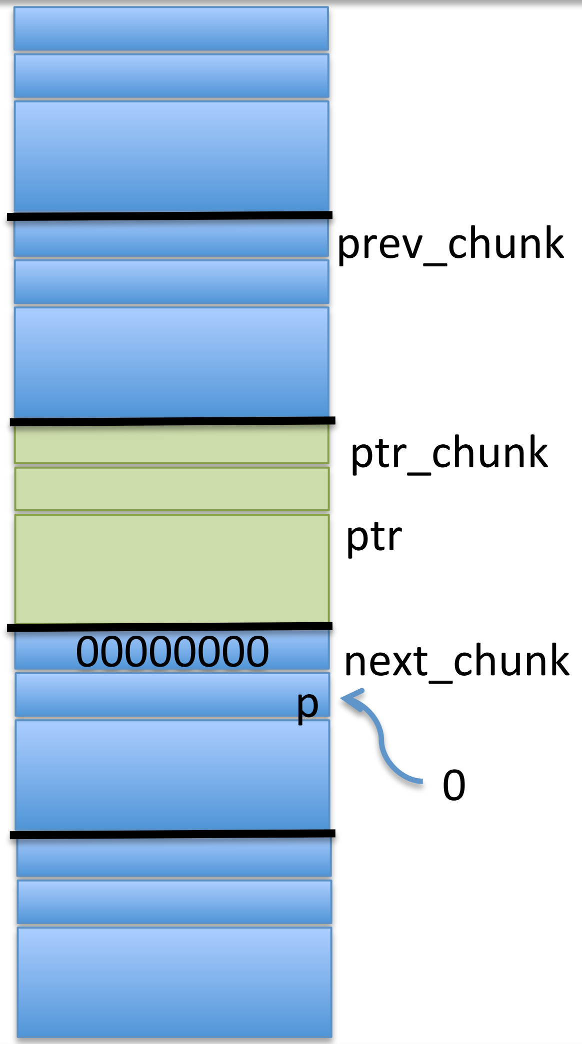
Heap Exploits

Chester Rebeiro

Indian Institute of Technology Madras

free(ptr)

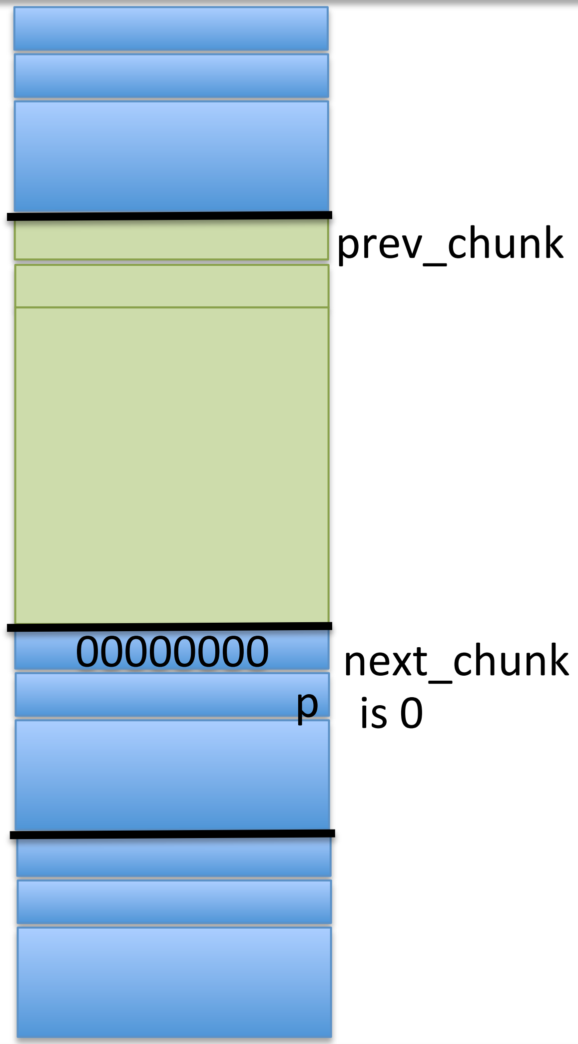
1. If the next chunk is allocated then
   - Set size to zero
   - Set p bit to 0



free(ptr)

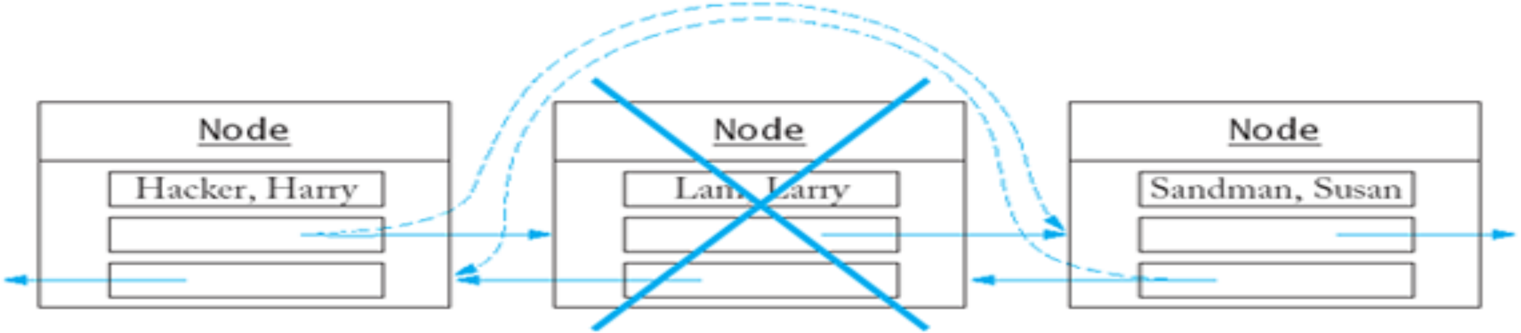
2. If the previous chunk is free then
   - Coalesce the two to create a new free chunk
   - This will also require unlinking from the current bin and placing the larger chunk in the appropriate bin

Similar is done if the next chuck is free as well.



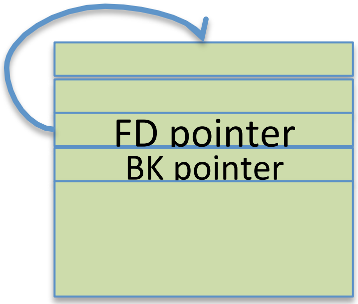
Unlinking from a free list

```
void unlink(malloc_chunk *P, malloc_chunk *BK, malloc_chunk *FD){
        FD = P->fd;
        BK = P->bk;
        FD->bk = BK;
        BK->fd = FD;
}
```



More recent Unlinking

```
/* Take a chunk off a bin list */
void unlink(malloc_chunk *P, malloc_chunk *BK, malloc_chunk *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);
	else {
		FD->bk = BK;
		BK->fd = FD;
	}
}
```



Detects cases such as these

Causing programs like this to crash

```
void main()
{
	char *a = malloc(10);
	free(a);
	free(a);
}
```

Some double frees are detected

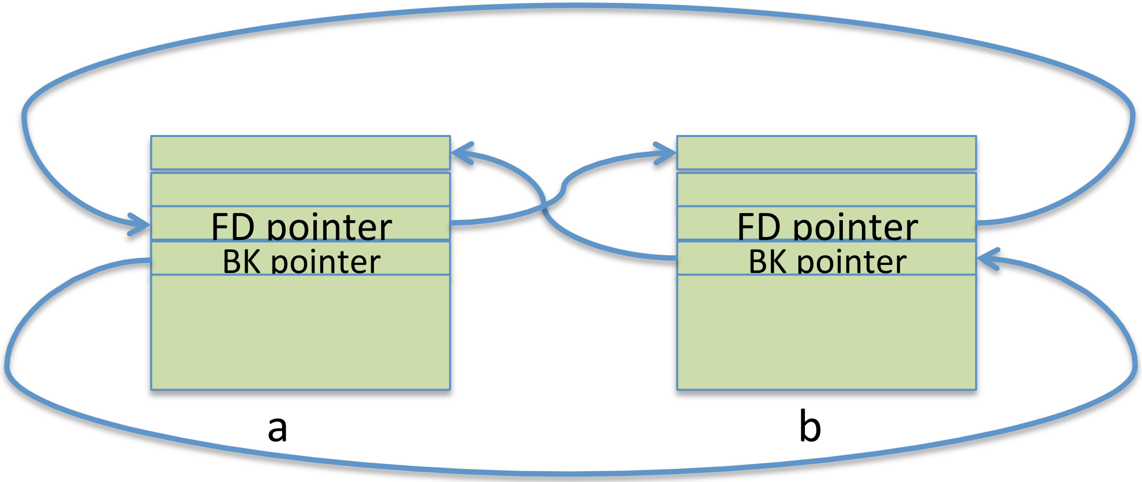
```
/* Take a chunk off a bin list */
FD)
ked list",P);
```

```
chester@aahalya:~/sse/malloc$ ./a.out
*** glibc detected *** ./a.out: double free or corruption (fasttop): 0x0961d008 ***
======= Backtrace: =========
/lib/i686/cmov/libc.so.6(+0x6af71)[0xb7610f71]
/lib/i686/cmov/libc.so.6(+0x6c7c8)[0xb76127c8]
/lib/i686/cmov/libc.so.6(cfree+0x6d)[0xb76158ad]
./a.out[0x8048425]
/lib/i686/cmov/libc.so.6(__libc_start_main+0xe6)[0xb75bcca6]
./a.out[0x8048361]
======= Memory map: ========
08048000-08049000 r-xp 00000000 00:15 82314386   /home/chester/sse/malloc/a.out
08049000-0804a000 rw-p 00000000 00:15 82314386   /home/chester/sse/malloc/a.out
0961d000-0963e000 rw-p 00000000 00:00 0          [heap]
b7400000-b7421000 rw-p 00000000 00:00 0
b7421000-b7500000 ---p 00000000 00:00 0
b7587000-b75a4000 r-xp 00000000 08:01 884739     /lib/libgcc_s.so.1
b75a4000-b75a5000 rw-p 0001c000 08:01 884739     /lib/libgcc_s.so.1
b75a5000-b75a6000 rw-p 00000000 00:00 0
b75a6000-b76e6000 r-xp 00000000 08:01 901176     /lib/i686/cmov/libc-2.11.3.so
b76e6000-b76e7000 ---p 00140000 08:01 901176     /lib/i686/cmov/libc-2.11.3.so
b76e7000-b76e9000 r--p 00140000 08:01 901176     /lib/i686/cmov/libc-2.11.3.so
b76e9000-b76ea000 rw-p 00142000 08:01 901176     /lib/i686/cmov/libc-2.11.3.so
b76ea000-b76ed000 rw-p 00000000 00:00 0
b76ff000-b7701000 rw-p 00000000 00:00 0
b7701000-b7702000 r-xp 00000000 00:00 0          [vdso]
b7702000-b771d000 r-xp 00000000 08:01 884950     /lib/ld-2.11.3.so
b771d000-b771e000 r--p 0001b000 08:01 884950     /lib/ld-2.11.3.so
b771e000-b771f000 rw-p 0001c000 08:01 884950     /lib/ld-2.11.3.so
bff35000-bff4a000 rw-p 00000000 00:00 0          [stack]
Aborted
```

```
void main()
{
        char *a = malloc(10);
        free(a);
        free(a);
}
```

Most double frees are not detected

After the second free

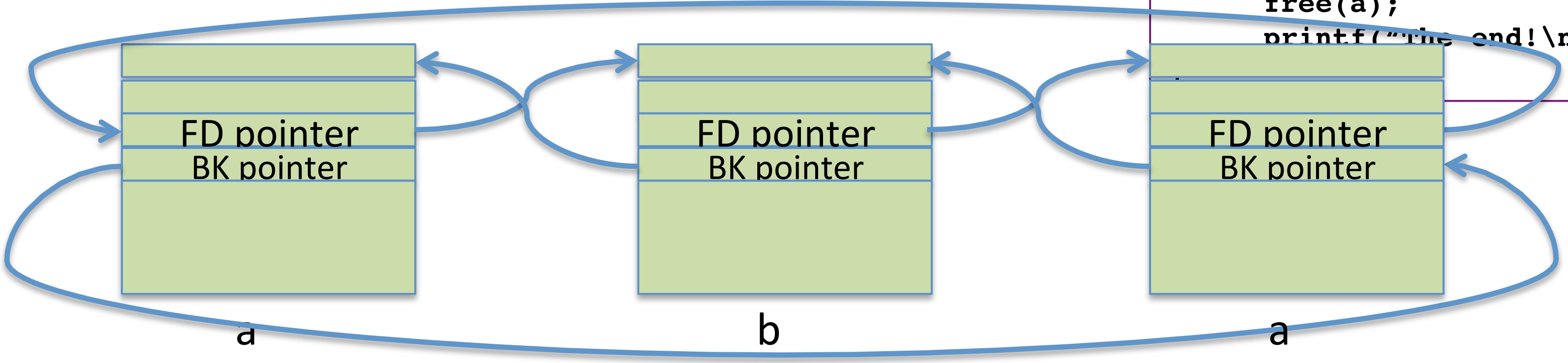



```
void main()
{
        char *a = malloc(10);
        char *b = malloc(10);
        free(a);
        free(b);
        free(a);
        printf("The end!\n");
}
```

Most double frees are not detected

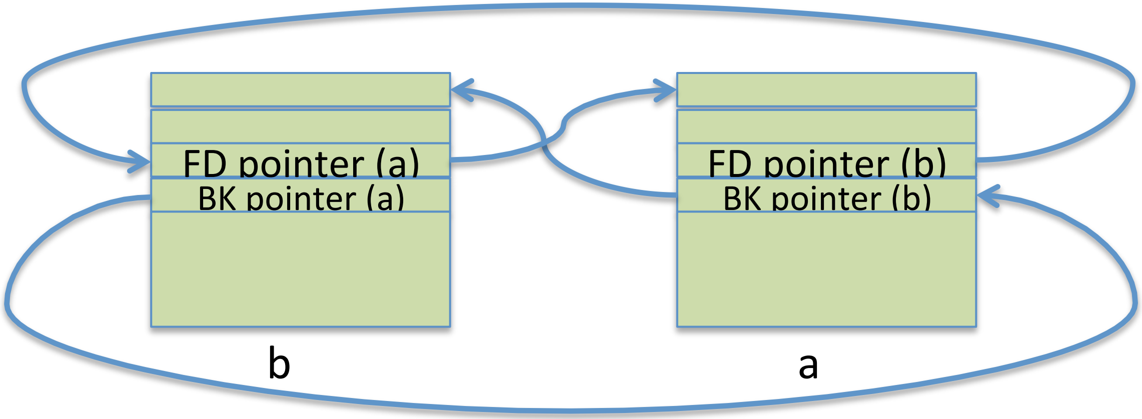
After the third free

```
void main()
{
        char *a = malloc(10);
        char *b = malloc(10);
        free(a);
        free(b);
        free(a);
        printf("The end!\n");
```



Another malloc

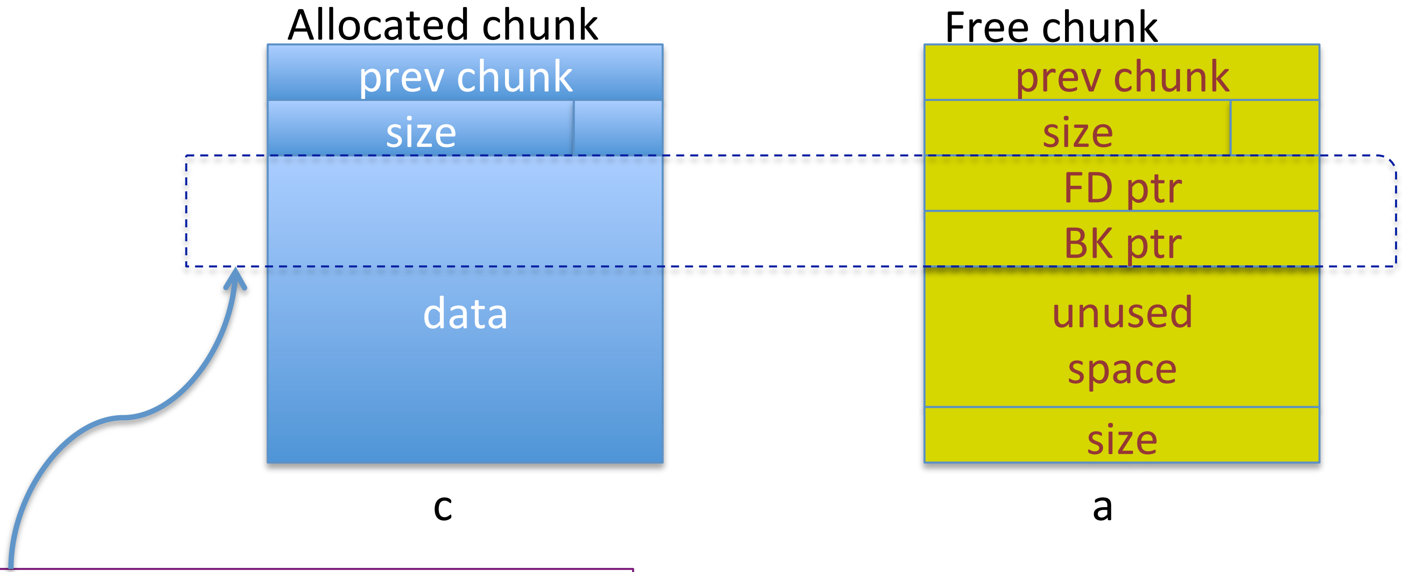
Another malloc
c gets allocated the same address as a




```
void main()
{
        char *a = malloc(10);
        char *b = malloc(10);
        char *c;
        free(a);
        free(b);
        free(a);
        c = malloc(10);
}
```

```
chester@aahalya:~/sse/malloc$ ./a.out
a=09108008
b=09108018
c=09108008
```

Two views of the same chunk




```
*c = 0xdeadbeef;
*(c+4) = 0xdeadbeef;
```

you can control the FD ptr and BK ptr contents using c

Exploiting

```
char payload[] =
“\x33\x56\x78\x12\xac\xb4\x67”;

Void fun1(){}

void main()
{
      char *a = malloc(10);
      char *b = malloc(10);
      char *c;

      fun1();
      free(a);
      free(b);
      free(a);
      c = malloc(10);
      *(c + 0) = GOT entry–12 for fun1;
      *(c + 4) = payload;
      some malloc(10);
      fun1();
}
```

Need to lookout for programs that have (something) like this structure

We hope to execute payload instead of the 2nd invocation of fun1();

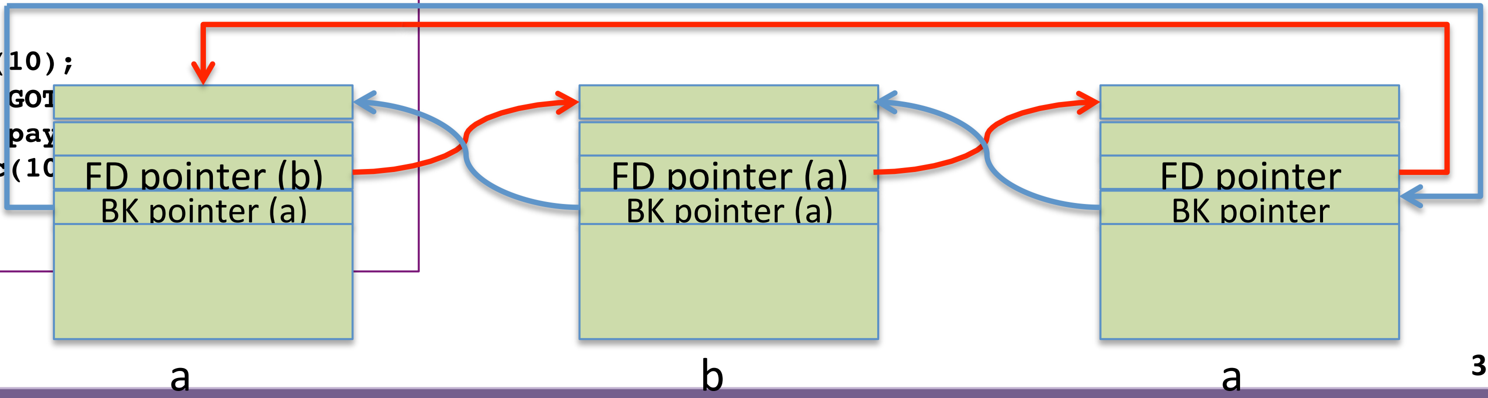
Exploiting

```
char payload[] =
"\x33\x56\x78\x12\xac\xb4\x67";

Void fun1(){}

void main()
{
      char *a = malloc(10);
      char *b = malloc(10);
      char *c;

      fun1();
      free(a);
      free(b);
→     free(a);
      c = malloc(10);
      *(c + 0) = GOT
      *(c + 4) = pay
      some malloc(10
      fun1();
}
```



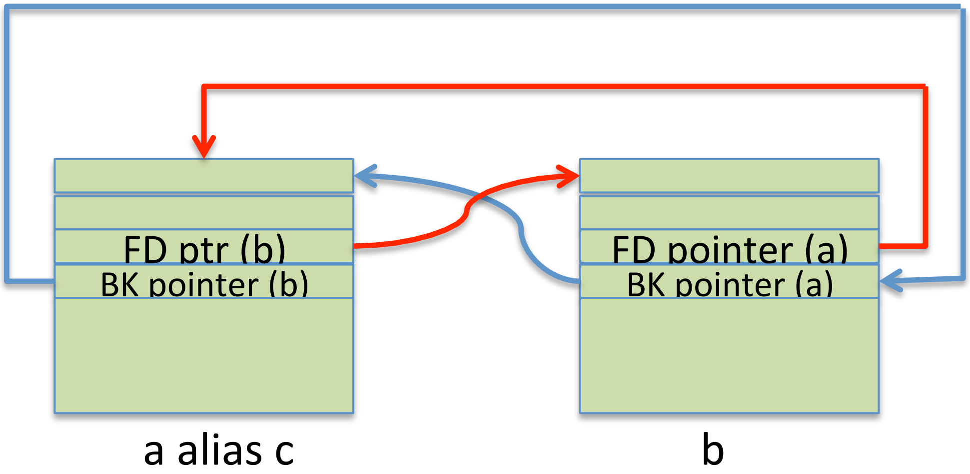
Exploiting

```
char payload[] =
"\x33\x56\x78\x12\xac\xb4\x67";

Void fun1(){}

void main()
{
      char *a = malloc(10);
      char *b = malloc(10);
      char *c;

      fun1();
      free(a);
      free(b);
      free(a);
      c = malloc(10);
      *(c + 0) = GOT entry–12 for fun1;
      *(c + 4) = payload;
      some malloc(10);
      fun1();
}
```



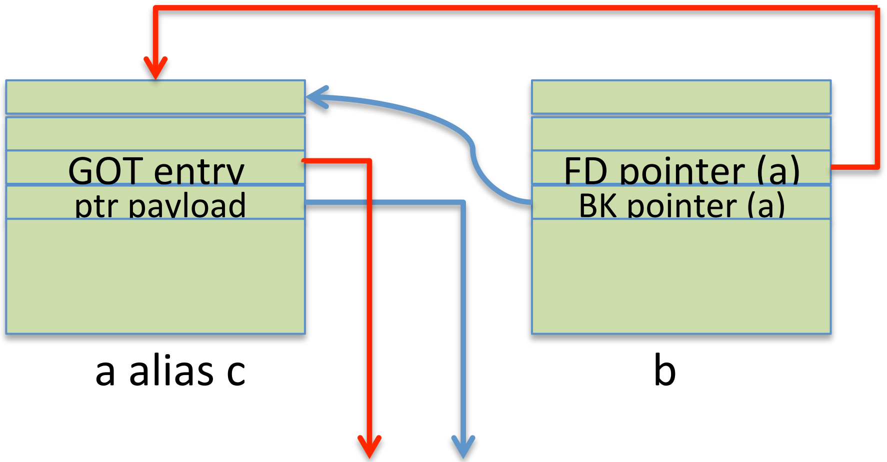
Exploiting

```
char payload[] =
"\x33\x56\x78\x12\xac\xb4\x67";

Void fun1(){}

void main()
{
        char *a = malloc(10);
        char *b = malloc(10);
        char *c;

        fun1();
        free(a);
        free(b);
        free(a);
        c = malloc(10);
        *(c + 0) = GOT entry-12 for fun1;
 →      *(c + 4) = payload;
        some malloc(10);
        fun1();
}
```



Exploiting

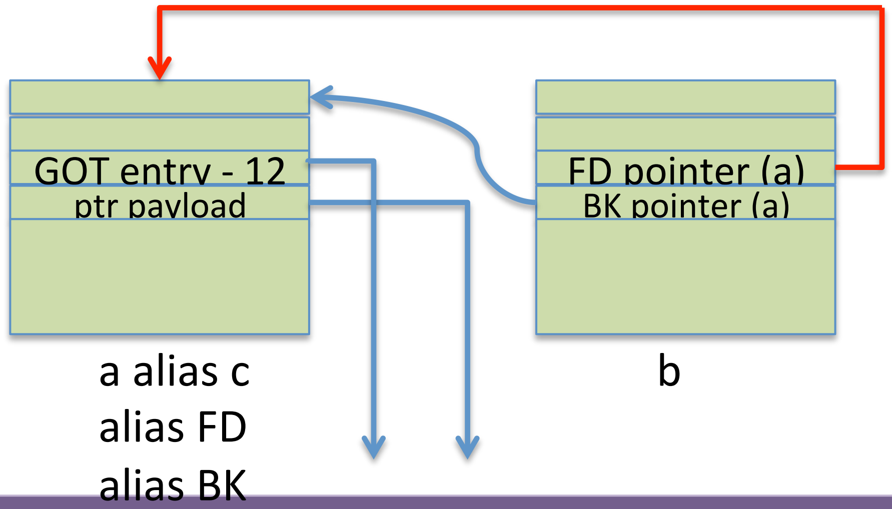
```
char payload[] =
“\x33\x56\x78\x12\xac\xb4\x67”;

Void fun1(){}

void main()
{
      char *a = malloc(10);
      char *b = malloc(10);
      char *c;

      fun1();
      free(a);
      free(b);
      free(a);
      c = malloc(10);
      *(c + 0) =GOT entry—12 for fun1;
      *(c + 4) = payload;
      some malloc(10);
      fun1();
}
```

```
unlink(P){
        FD = P->fd;
        BK = P->bk;
        FD->bk = BK;
        BK->fd = FD;
}
```



Exploiting Heap

```
char payload[] =
“\x33\x56\x78\x12\xac\xb4\x67”;

Void fun1(){}

void main()
{
      char *a = malloc(10);
      char *b = malloc(10);
      char *c;

      fun1();
      free(a);
      free(b);
      free(a);
      c = malloc(10);
      *(c + 0) =GOT entry—12 for fun1;
      *(c + 4) = payload;
      some malloc(10);
      fun1();
}
```

Payload executes

Ponder About

```
char *secret = "THIS IS A SECRET MESSAGE!";

int main(int argc, char **argv){
    int *a, *b, *c, *d, *e;

    a = malloc(32);      /* S1 */
    b = malloc(32):      /* S2 */
    c = malloc(32);      /* S3 */
    free(a);             /* S4 */
    d = malloc(32);      /* S5 */
    free(b);             /* S6 */
    free(d);             /* S7 */
    e = malloc(32);      /* S8 */
    my_malicious_function(e); /* S9 */
    a = malloc(32);      /* S10 */
    printf("%s", a);    /* S11 */
}
```

What does the heap look like after each statement S1 to S10 has completed execution?

Show how a malicious function *my_malicious_function* can be written so that S11 prints the secret message.

CR

Other heap based attacks

- Heap overflows
- Heap spray
- Use after free
- Metadeta exploits