



CHANGED 11 HOURS AGO



OWNED THIS NOTE



Edit

# 你所不知道的 C 語言：記憶體管理、對齊及硬體特性

Copyright (憲C) 2018 宅色夫

直播錄影

## 背景知識

- 指標篇

- C99/C11 規格書 6.2.5 (28) 提到:

A pointer to void shall have the same representation and alignment requirements as a pointer to a character type.

- 在 6.3.2.3 (1) 提到:

A pointer to void may be converted to or from a pointer to any object type. A pointer to any object type may be converted to a pointer to void and back again; the result shall compare equal to the original pointer.

```
int *intptr = NULL;
void *dvoidptr = &intptr; /* 6.3.2.3 (1) */
*(void **) dvoidptr = malloc(sizeof *intptr);
```

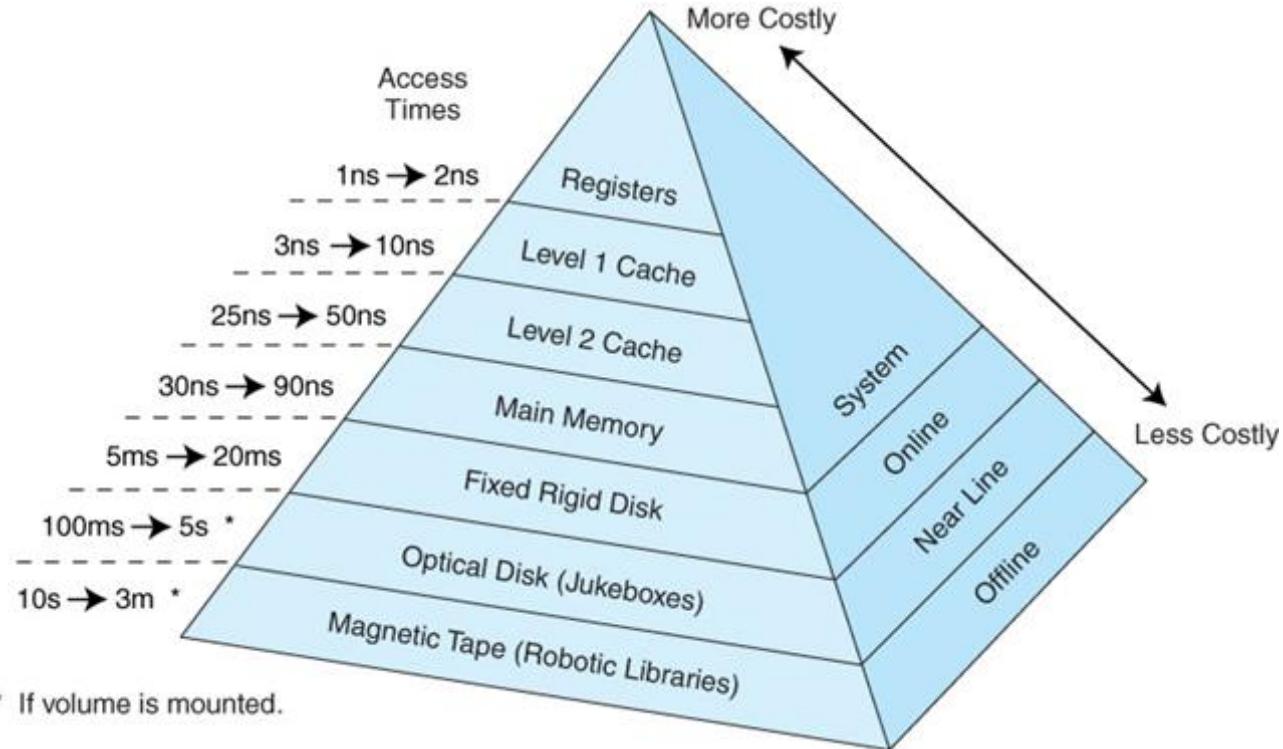
- 將 `int **` 轉型為 `void *`:  (fine); 將 `void *` 轉型為 `void **`:  (not ok)。C 語言不保證你可以安全地轉換 `void *` 為任意型態後，再轉換回原本的型態
- `void *` 的設計，導致開發者必須透過 **explicit (顯式)** 或強制轉型，才能存取最終的 object，否則就會丟出編譯器的錯誤訊息，從而避免危險的指標操作。也就是說，我們無法直接對 `void *` 做數值操作

```
void *p = ...;
void *p2 = p + 1; /* what exactly is the size of void? */
```

- 換言之，`void *` 存在的目的就是為了強迫使用者使用 **顯式轉型** 或是 **強制轉型**，以避免 Undefined behavior 產生
- C/C++ [Implicit conversion vs. Explicit type conversion](#)
- [函式呼叫篇](#)
  - `free()` 釋放的是 pointer 指向位於 heap 的連續記憶體，而非 pointer 本身佔有的記憶體 (`*ptr`)
  - glibc 提供了 `malloc_stats()` 和 `malloc_info()` 這兩個函式，可顯示 process 的 heap 資訊

## 你可能沒想過的 Memory

---



\* If volume is mounted.

- Cache locality
- Understanding virtual memory - the plot thickens

The virtual memory allocator (VMA) may give you a memory it doesn't have, all in a vain hope that you're not going to use it. Just like banks today

1. 現代銀行和虛擬記憶體兩者高度相似
2. malloc 給 valid pointer不要太高興，等你要開始用的時候搞不好作業系統給個 OOM。簡單來說就是一張支票，能不能拿來開等到兌現才知道

- Understanding stack allocation

This is how variable-length arrays (VLA), and also alloca() work, with one difference - VLA validity is limited by the scope, alloca'd memory persists until the current function returns (or unwinds if you're feeling sophisticated).

C99 的 variable length array (VLA) 的運作是因為 stack frame的特性，反正你要多少，stack 在調整時順便加一加。malloc 一樣的原則

- Slab allocator

有的時候程式會allocate並使用多個不連續的記憶體區塊，如樹狀的資料結構。這時候對於系統來說有幾個問題，一是fragment、二是因為不連續，無法使用cache增快效能。

- Demand paging explained

Linux系統提供一系列的記憶體管理API

- 分配，釋放
- 記憶體管理API
  - mlock: 禁止被 swapped out (向 OS 提出需求，OS 不一定會理)
  - madvise: 提供管道告訴系統page管理方式如 MADV\_RANDOM，期待記憶體page讀取行為是隨機的。(不是標準，隨著 Linux 核心改版會有所不同)
  - lazy loading: 配置記憶體先給位址。等到行程要存取時，作業系統O就會發現存取到尚未觸及的記憶體，於是產生 page fault，這時作業系統再去處理 page 分配的問題。

## Copy-on-write

有些情況是一個process要吃別的process已經map到記憶體的內容，而不要把自己改過的資料放回原本的記憶體。也就是說最終會有兩塊記憶體(兩份資料)。當然每次都複製有點多餘，因此系統使用了Copy-on-write機制。要怎麼做呢？就是在mmap使用MAP\_PRIVATE參數即可。

延伸閱讀 (報告編撰中):

- 現代處理器設計: Cache 原理和實際影響
- Cache 原理和實際影響: 進行 CPU caches 中文重點提示並且重現對應的實驗
- 針對多執行緒環境設計的 Memory allocator
- rpmalloc 探討

## 重新看 Heap

“heap”的中文翻譯

- 台灣: 堆積
- 中國: 堆



這裡的 Heap 跟資料結構中的 Heap 無關

動態配置產生，系統會存放在另外一塊空間，稱之為 “Heap”。

依據 [Why are two different concepts both called “heap”?](#)

Donald Knuth 在《The Art of Computer Programming》(Third Ed., Vol. 1, p. 435) 提到:

Several authors began about 1975 to call the pool of available memory a “heap.”

He doesn't say which authors and doesn't give references to any specific papers, but does say that the use of the term “heap” in relation to priority queues is the traditional sense of the word.

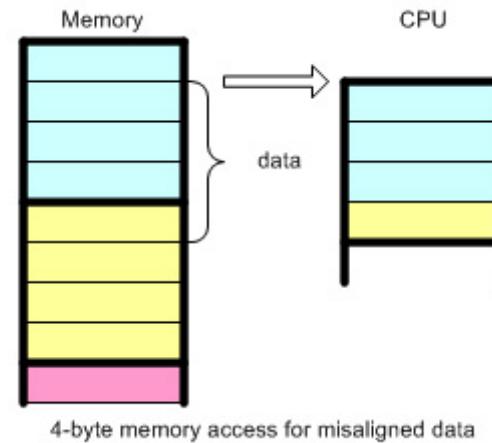
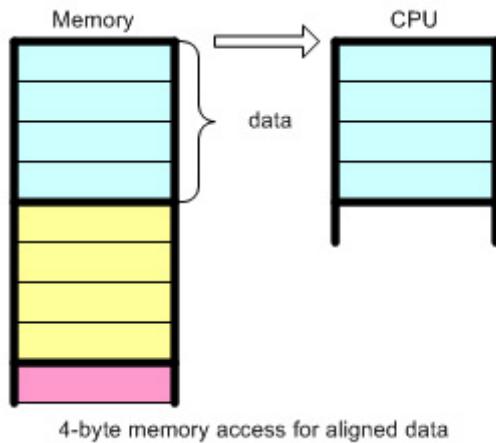
下面的留言和解答也有一些支持用 pool 來稱呼 **memory pool**

## data alignment

---

- 一個 data object 具有兩個特性:
  - value
  - storage location (address)
- 而 data alignment 的意思就是, data 的 address 可以公平的被 1, 2, 4, 8,這些數字整除, 從這些數字可以發現他們都是 2 的 N 次方 (N為從 0 開始的整數)。換句話說, 這些 data object 可以用 1, 2, 4, 8 byte 去 alignment
- 電腦的 cpu 又是如何抓取資料呢 ?cpu 不會一次只抓取 1 byte 的資料, 因為這樣太慢了, 如果有個資料型態是 int 的 資料, 如果你只抓取 1 byte ,就必須要抓 4 次(int 為 4 byte), 有夠慢。所以 cpu 通常一次會取 4 byte(要看電腦的規格 32 位元的 cpu 一次可以讀取 32 bit 的資料, 64 位元一次可以讀取 64 bit), 並且是按照順序取的,  
例如:
  - 第一次取: 0~3

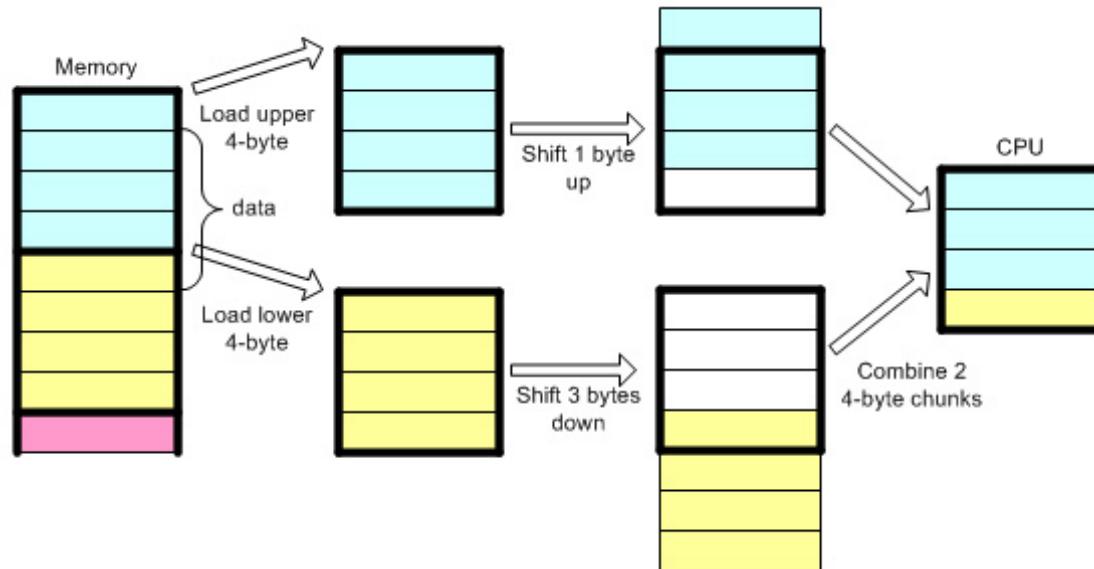
- 第二次取: 4~7



所以如果你的資料是分布在 1~4 那還是會

- 第一次取: 0~3 將, 0 的資料去掉, 留下 1~3
- 第二次取: 4~7 將, 5~7 的資料去掉, 留下 4

- 再將 1~3 4 合起來



由於資料分布不在 4 的倍數，導致了存取速度降低，編譯器在分配記憶體時，就會按照宣告的型態去做 alignment，例如 int 就是 4 byte alignment。

- struct 會自動做 alignment，假設創了一個 struct，如下面 code 所示

```
struct s1 {
    char c;
    int a;
}
```

原本推論 char 為 1 byte，而 int 為 4 byte，兩個加起來應該為 5 byte，然而實際上為 8 byte，由於 int 為 4 byte，所以 char 為了要能 alignment 4 byte 就多加了 3 byte 進去，使得 cpu 存取速度不會因 address 不是在 4 的倍數上，存取變慢。

- 下一個實驗：

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
typedef struct _s1 {
    char a[5];
} s1;
int main() {
    s1 p[10];
    printf("struct s1 size: %ld byte\n", sizeof(s1));
    for(int i = 0; i < 10; i++) {
        printf("the struct p[%d] address =%p\n", i, p + i);
    }
}
```

得到執行結果為

```
struct s1 size: 5 byte
the struct p[0] address =0x7ffc14c80170
the struct p[1] address =0x7ffc14c80175
the struct p[2] address =0x7ffc14c8017a
the struct p[3] address =0x7ffc14c8017f
the struct p[4] address =0x7ffc14c80184
the struct p[5] address =0x7ffc14c80189
the struct p[6] address =0x7ffc14c8018e
the struct p[7] address =0x7ffc14c80193
the struct p[8] address =0x7ffc14c80198
the struct p[9] address =0x7ffc14c8019d
```

由於 char type 的 data 大小只佔 1 byte 所以只要 1 byte alignment，也就是不用使用 padding 讓其變成 4 的倍數。由於編譯器會自動幫我們以 data 的大小做 alignment，假設有 int type (4 byte) 在配置時，已是 4 byte alignment。

- 部分微處理器可不用特別處理 data alignment 的議題，例如 intel x86/x86\_64 系統允許 data unalignment。以下用 x86\_64 實驗 data alignment 跟 data unalignment 的差異。
- 首先我建立兩個 struct 兩個放的東西是相同的，唯一不同的是 t1 有加 pack 這條指令告訴 compiler 說 test1 裡的 data 只要 1 byte alignment 就好，t2 則是會按照宣告的 type 作 alignment 所以 t2 裡會有 padding。

## 重新設計實驗

### 方式一：將 struct 資料結構改大

```
#pragma pack(1)
typedef struct _test1 {
    char c[3];
    int num[256];
} test1;
#pragma pack(pop)

typedef struct _test2 {
    char c[3];
    int num[256];
} test2;
```

### 方式二：去掉 cache 的影響

每次執行程式前，確認以下命令已執行：

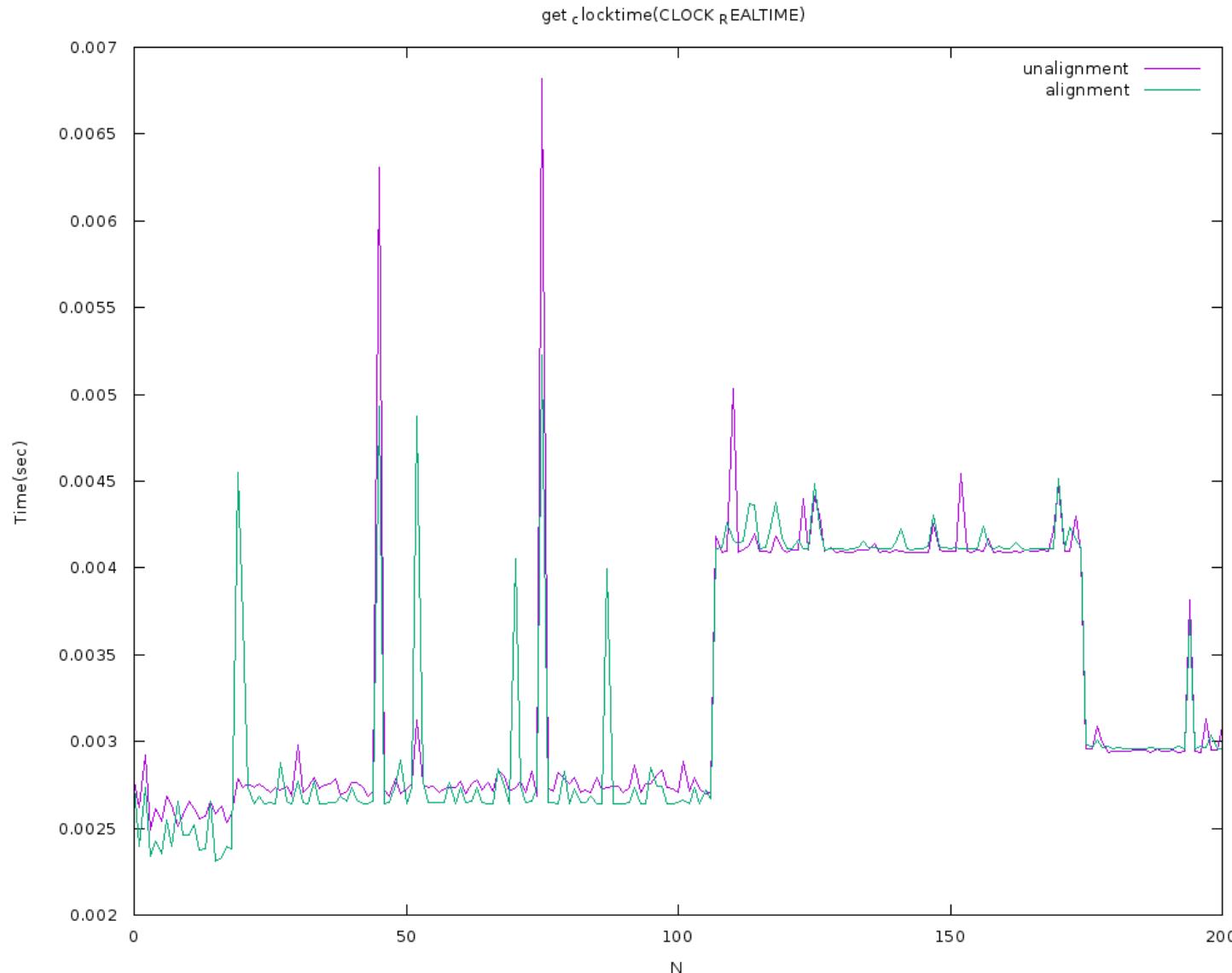
```
sudo sh -c 'echo 3 > /proc/sys/vm/drop_caches'
```

## 清除 pagecache、dentries 及 inodes

這段程式碼執行 200 次 看是否平均起來的執行時間相仿

```
for i in `seq 0 1 200`; do \
    echo 3 > /proc/sys/vm/drop_caches ;\
    printf "%d," $$i;\
    ./pointer; \
done > clock_gettime.txt
```

效能分佈:



malloc 本來配置出來的記憶體位置就有做 alignment，根據 malloc 的 man page 裡提到：

The malloc() and calloc() functions return a pointer to the allocated memory, which is suitably aligned for any built-in type.

實際上到底 malloc 做了怎樣的 data alignment，繼續翻閱 [The GNU C Library - Malloc Example](#)，裡面特別提到：

In the GNU system, the address is always a multiple of eight on most systems, and a multiple of 16 on 64-bit systems.

在大多數系統下，malloc 會以 8 bytes 進行對齊；而在 64-bit 的系統下，malloc 則會以 16 bytes 進行對齊。

對應的實驗，malloc 在 Linux x86\_64 以 16 bytes 對齊：

```
for (int i = 0; i < 10000; ++i) {
    char *z;
    z = malloc(sizeof(char));
}
```

結果用 gdb 測過之後，發現位址的結尾的確都是 0 結尾，表示真的是以 16-byte 做對齊。

### Unaligned memory access

An unaligned memory access is a load/store that is trying to access an address location which is not aligned to the access size.

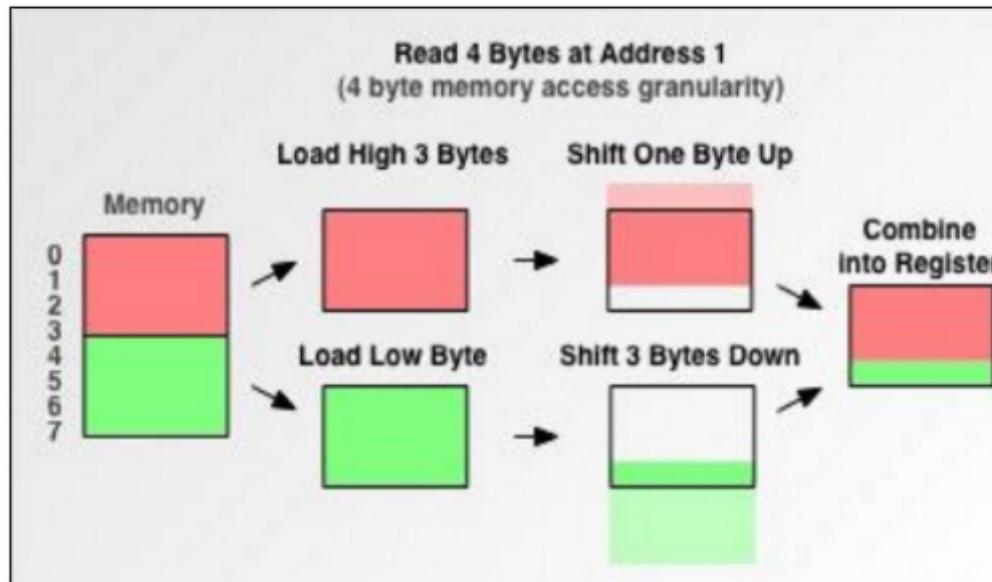
e.g: A load instruction trying to access 4 bytes from address 0x1 is an unaligned access. This typically gets split into two internal operations as shows in the following diagram and merges into one.

Note that in case of systems with caches, there can also be cases of addresses crossing a cache line boundary (a.k.a misaligned access) and sometimes accesses can also be crossing a page boundary.

Then you can also have other complexities like one half of the memory access is hit in cache while the other half is a miss in the cache. The load store execution unit along with cache controllers deals with these complexities but in summary it follows the same basic principle of merging two access

Some architectures (like Intel x86) also has alignment interrupts that help in detecting unaligned memory access.

- How unaligned accesses are handled?



13

考慮以下 `unaligned_get32` 函式的實作: (假設硬體架構為 32-bits)

```
#include <stdint.h>
#include <stddef.h>
uint8_t unaligned_get8(void *src) {
    uintptr_t csrc = (uintptr_t) src;
    uint32_t v = *(uint32_t *) (csrc & 0xfffffffffc);
    v = (v >> (((uint32_t) csrc & 0x3) * 8)) & 0x000000ff;
    return v;
}
uint32_t unaligned_get32(void *src) {
    uint32_t d = 0;
    uintptr_t csrc = (uintptr_t) src;
    for (int n = 0; n < 4; n++) {
        uint32_t v = unaligned_get8((void *) csrc);
        v = v << (n * 8);
        d = d | v;
        csrc++;
    }
    return d;
}
```

對應的 `unaligned_set32` 函式:

```
void unaligned_set8(void *dest, uint8_t value) {
    uintptr_t cdest = (uintptr_t) dest;
    uint32_t d = *(uint32_t *) (cdest & 0xfffffffffc);
    uint32_t v = value;
    for (int n = 0; n < 4; n++) {
        uint32_t v = unaligned_get8((void *) csrc);
        v = v << (n * 8);
        d = d | v;
        csrc++;
    }
    return d;
```

```
}

void unaligned_set32(void *dest, uint32_t value) {
    uintptr_t cdest = (uintptr_t) dest;
    for (int n = 0; n < 4; n++) {
        unaligned_set8((void *) cdest, value & 0x000000ff);
        value = value >> 8;
        cdest++;
    }
}
```

參考資料:

- Data Alignment
- Linux kernel: unaligned memory access
- Data alignment: Straighten up and fly right
- Interleaved Pixel Lookup for Embedded Computer Vision, Page 10

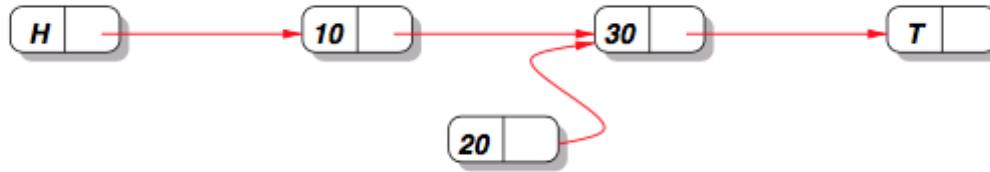
## 案例分析: concurrent-II

---

concurrent-II

這個實作的基礎是 [A Pragmatic Implementation of Non-Blocking Linked Lists](#) 這篇論文。

文中首先提到對「天真版」的 linked list 插入與刪除，就算直接用 compare-and-swap 這個最小操作改寫，也沒有辦法保證結果正確。在只有 insert 的狀況下可以成立，但如果加上 delete 的話，如果 insert 跟 delete 發生時機很接近，有可能會發生以下的狀況：



情境是這樣：準備插入 20，並且刪除 10。這是做到一半的狀況。

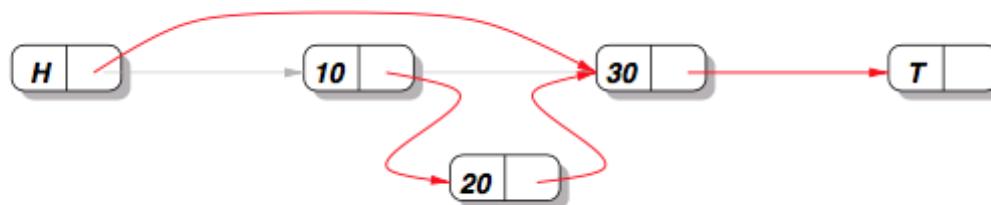
- delete 做的事情是這樣：

```
CAS(&(H->next), head_next, head->next->next)
```

- insert 準備執行的動作是這樣：

```
CAS(&(node_10->next), node_10_next, node_20)
```

但是兩個 CAS 的動作的先後是無法確定的。如果 insert 的先發生，那結果是正確的。但是如果 delete 的先發生，就會變成下面這個樣子：



結果顯然不正確，因為本來沒有要刪除 20 那個節點。那到底要怎麼辦呢？這篇文章提到一個作法：刪除時不要真的刪除，而是掛一個牌子說「此路段即將刪除」，但是還是過得去，像這樣：



然後等到真的要拆他的時候（比如說需要插入的時候），再把它拆掉。

所以需要有個方法標示「不需要的節點」，然後要有 atomic operation.

## 標示不用的節點

不過看了一陣子，發現了幾個問題：為什麼只用位元運算就可以做到「邏輯上」刪除節點？

答案是用 data alignment。首先是 C99 的規格中 6.7.2.1 提到：

- 12

Each non-bit-field member of a structure or union object is aligned in an implementation-defined manner appropriate to its type.

- 13

Within a structure object, the non-bit-field members and the units in which bit-fields reside have addresses that increase in the order in which they are declared. A pointer to a structure object, suitably converted, points to its initial member (or if that member is a bit-field, then to the unit in which it resides), and vice versa. There may be unnamed padding within a structure object, but not at its beginning.

所以這個東西是 implementation-defined. 查看 [gcc 的文件](#) 怎麼說：

The alignment of non-bit-field members of structures (C90 6.5.2.1, C99 and C11 6.7.2.1).  
Determined by ABI.

先回去翻 C99 的規格，裡面提到關於 `intptr_t` 這個資料型態：

#### 7.18.1.4 Integer types capable of holding object pointers

The following type designates a signed integer type with the property that any valid pointer to void can be converted to this type, then converted back to pointer to void, and the result will compare equal to the original pointer: `intptr_t`

所以 `intptr_t` 是個 integral type, 要至少可以裝得下 pointer

然後對照 `/usr/include/stdint.h`, 查到以下內容：

```
/* Types for `void *' pointers. */
#if __WORDSIZE == 64
# ifndef __intptr_t_defined
typedef long int      intptr_t;
# define __intptr_t_defined
# endif
typedef unsigned long int    uintptr_t;
#else
# ifndef __intptr_t_defined
typedef int      intptr_t;
# define __intptr_t_defined
# endif
typedef unsigned int      uintptr_t;
#endif
```

繼續找 [x86\\_64 ABI](#), 裡面其中一個附表：

Figure 3.1: Scalar Types

Type	C	sizeof	Alignment (bytes)	AMD64 Architecture
------	---	--------	----------------------	-----------------------

Integral	<code>_Bool<sup>†</sup></code>	1	1	boolean
	<code>char</code>	1	1	signed byte
	<code>signed char</code>			
	<code>unsigned char</code>	1	1	unsigned byte
	<code>short</code>	2	2	signed twobyte
	<code>signed short</code>			
	<code>unsigned short</code>	2	2	unsigned twobyte
	<code>int</code>	4	4	signed fourbyte
	<code>signed int</code>			
	<code>enum<sup>††</sup></code>			
	<code>unsigned int</code>	4	4	unsigned fourbyte
	<code>long</code>	8	8	signed eightbyte
	<code>signed long</code>			
	<code>long long</code>			
Pointer	<code>signed long long</code>			
	<code>unsigned long</code>	8	8	unsigned eightbyte
	<code>unsigned long long</code>	8	8	unsigned eightbyte
	<code>__int128<sup>††</sup></code>	16	16	signed sixteenbyte
	<code>signed __int128<sup>††</sup></code>	16	16	signed sixteenbyte
Pointer	<code>unsigned __int128<sup>††</sup></code>	16	16	unsigned sixteenbyte
	<code>any-type *</code>	8	8	unsigned eightbyte
Floating-point	<code>any-type (*)()</code>			
	<code>float</code>	4	4	single (IEEE-754)
	<code>double</code>	8	8	double (IEEE-754)
	<code>long double</code>	16	16	80-bit extended (IEEE-754)
Decimal-floating-point	<code>__float128<sup>††</sup></code>	16	16	128-bit extended (IEEE-754)
	<code>_Decimal32</code>	4	4	32bit BID (IEEE-754R)
	<code>_Decimal64</code>	8	8	64bit BID (IEEE-754R)
Packed	<code>_Decimal128</code>	16	16	128bit BID (IEEE-754R)
	<code>__m64<sup>††</sup></code>	8	8	MMX and 3DNow!
	<code>__m128<sup>††</sup></code>	16	16	SSE and SSE-2

然後又說：

- Aggregates and Unions

Structures and unions assume the alignment of their most strictly aligned component. Each member is assigned to the lowest available offset with the appropriate alignment. The size of any object is always a multiple of the object's alignment.

An array uses the same alignment as its elements, except that a local or global array variable of length at least 16 bytes or a C99 variable-length array variable always has alignment of at least 16 bytes.<sup>4</sup>

Structure and union objects can require padding to meet size and alignment constraints. The contents of any padding is undefined.

因此可推論結構：

```
typedef intptr_t val_t;

typedef struct node {
    val_t data;
    struct node *next;
} node_t;
```

的定址當中，不會發生位址的最後一個 bit 被使用到的狀況(因為都是 4 的倍數 + 必須對齊)。所以就把最後一個 bit 設成 1 當作是刪掉這個節點的 mark. 而把最後一個 bit 設成 0, 就表示恢復原狀。

## glibc 的 malloc/free 實作

背景考量: [Deterministic Memory Allocation for Mission-Critical Linux](#)

## Memory request size

- 大request( $\geq 512$  bytes) : 使用best-fit的策略，在一個 range(bin)的 list 中尋找
- 小request( $\leq 64$  bytes) : caching allocation，保留一系列固定大小區塊的 list 以利迅速回收再使用
- 在兩者之間：試著結合兩種方法，有每個 size 的 list (小的特性)，也有合併(coalesce)機制、double linked list(大)
- 極大的 request( $\geq 128KB$  by default) : 直接使用 mmap()，讓 memory management 解決

arena 發音為 [əˋrinə]

## arena and thread

arena 即為 malloc 從系統取得的連續記憶體區域，分為 main arena 與 thread arena 兩種：

- main arena: 空間不足時，使用 brk() 延展空間，預設一次 132 KiB
- thread arena: 使用 mmap() 取得新記憶體，預設 map 1 MiB，分配給 heap 132 KiB，尚未分配給 heap 的空間設定為不可讀寫，1 MiB 使用完後會再 map 一個 1 MiB 的新 heap

每個 thread 在呼叫 malloc() 時，會分配到一個 arena，在開始時 thread 與 arena 是一對一的(per-thread arena)，但 arena 的總數有限制，超過時 threads 會開始共用 arena：

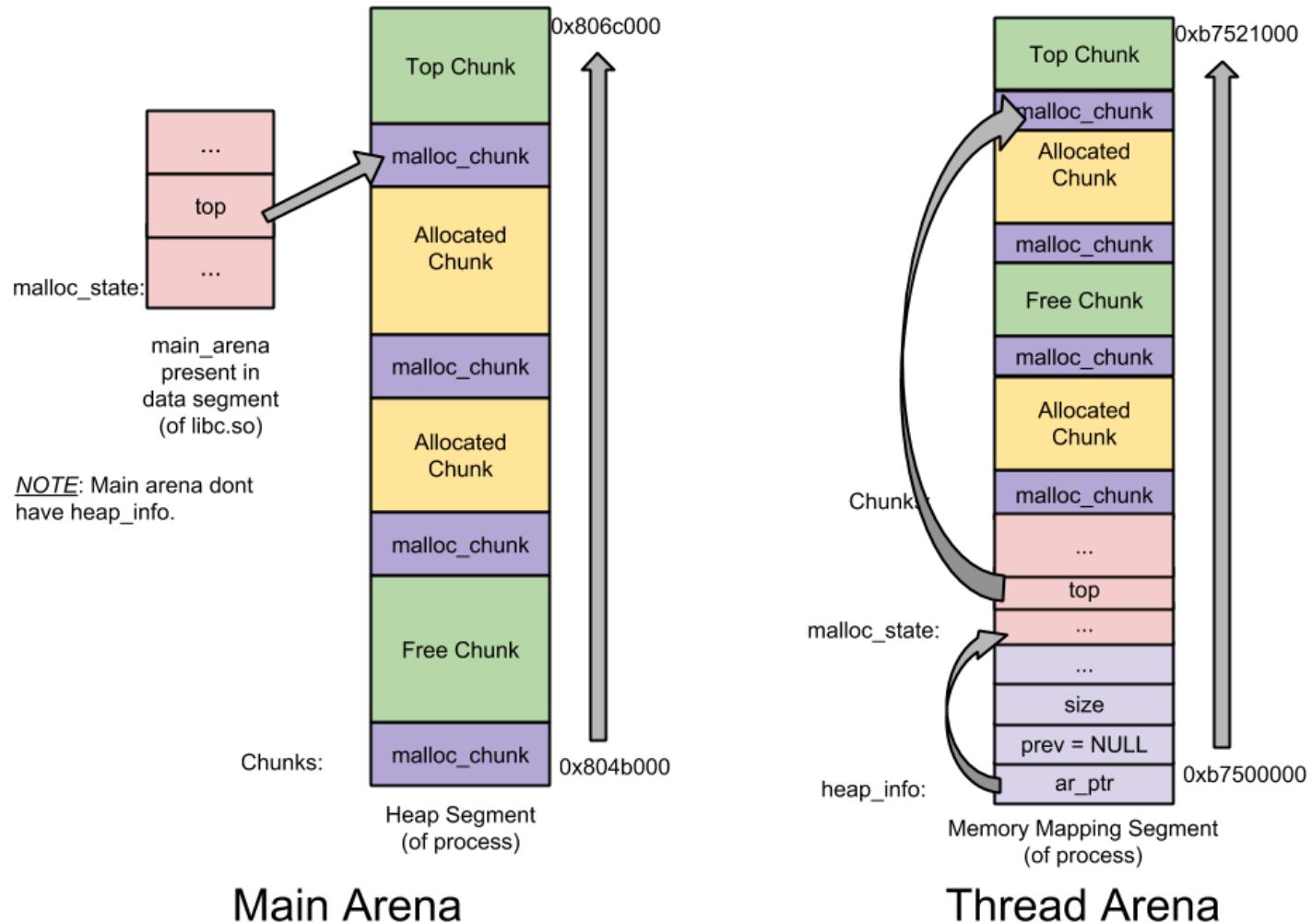
- 32 bit :  $2 * \text{number of cores}$
- 64 bit :  $8 * \text{number of cores}$

此設計是為了減少多 threads 記憶體浪費，但也因此 glibc malloc 不是 lockless allocator，對於有許多 threads 的 server 端程式來說，很有可能影響效能

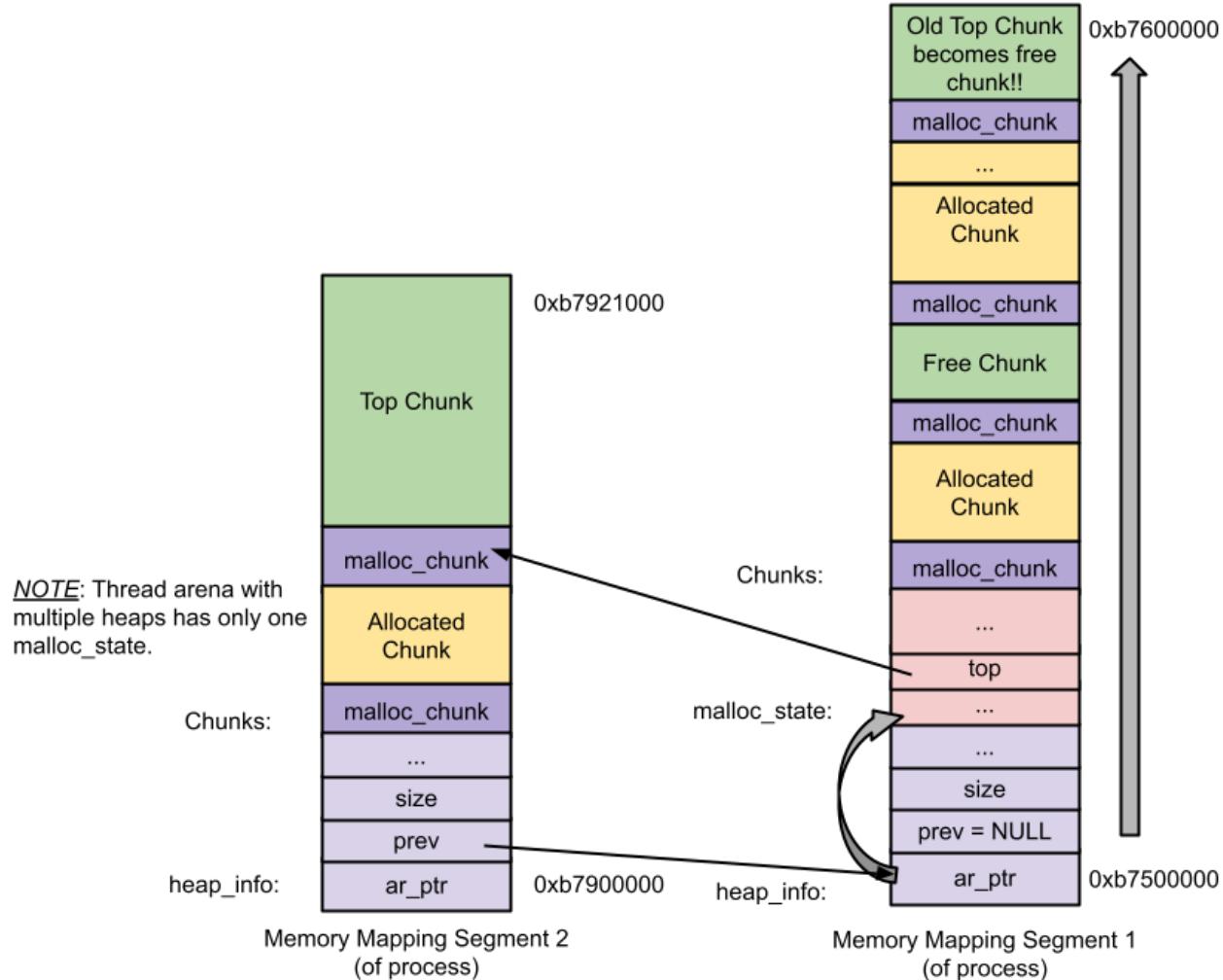
## Heap data structures

- **malloc\_state (Arena Header)** : 每個 arena 一個。紀錄 arena 的資訊，包含 bins (free list), top chunk, last remainder chunk 等等 allocation/free 需要的資訊。其中 main arena 的 header 在 data segment(static)，thread 的在 heap 中
- **heap\_info (Heap Header)** : 在 thread arena 中，每個 arena 可能有超過一個 heap。紀錄前一個heap、所屬的 arena、已使用/未使用的 size
- **malloc\_chunk (Chunk Header)** : 一個 heap 被分為許多區段，稱為 chunk，是 user data 儲存的地方，又依使用狀態分為 free 與 allocated。紀錄 chunk 本身的 size 與型態

*Main arena vs Thread arena :*



*multiple heap Thread arena :*

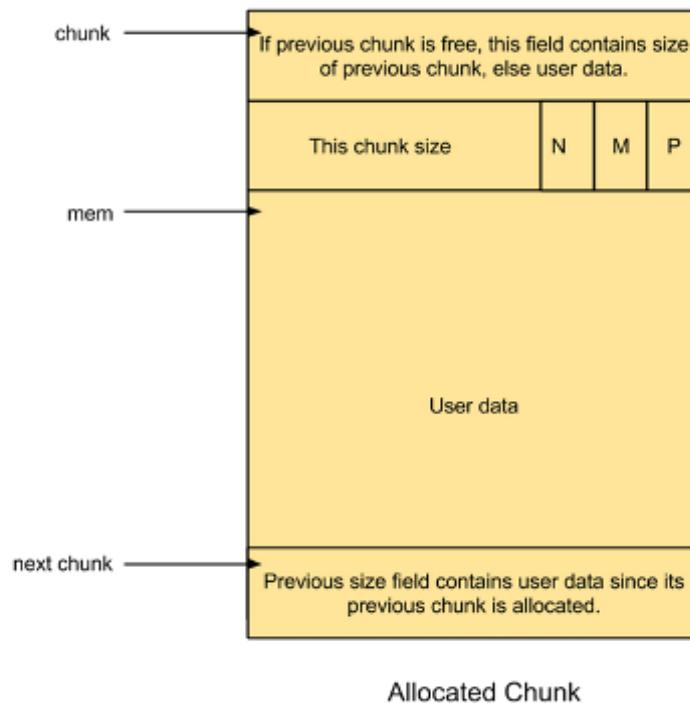


## Thread Arena (with multiple heaps)

### Chunks

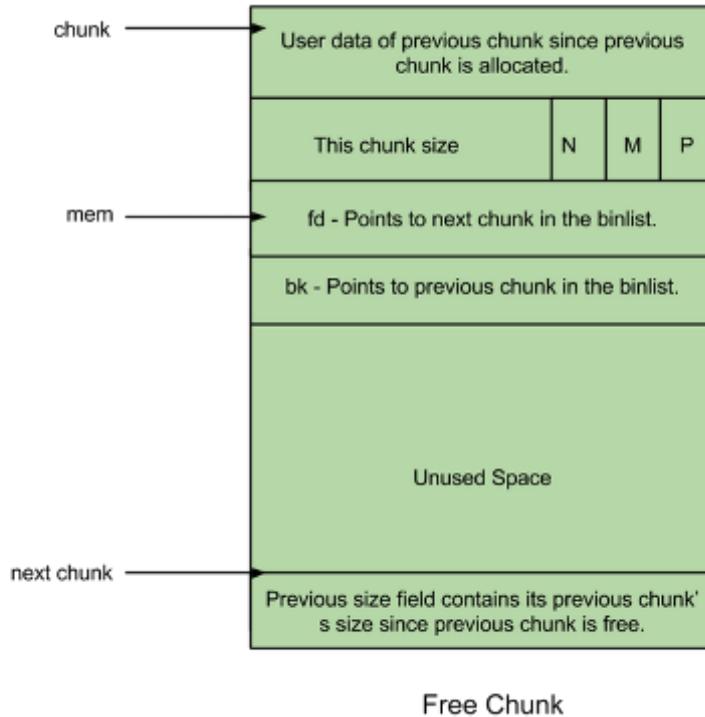
- 可用來分配或正在使用中的記憶體區塊，分為 allocated 與 free
- chunk 的大小在 32 bit 下最小 16 bytes，對齊 8 bytes；64 bit 下最小 32 bytes，對齊 16 bytes
  - 需要至少4個 word 作為 free chunk 時的 field

### *Allocated chunk :*



- **prev\_size** : 如果前一個 chunk 是 free chunk，包含前一個 chunk 的 size，若是 allocated，則包含 user data
- **size** : 此 chunk 的大小，最後 3 bit 作為 flag 使用
  - **PREV\_INUSE** § : 前一個 chunk 是 allocated
  - **IS\_MAPPED** (M) : 是 mmap 取得的獨立區塊
  - **NON\_MAIN\_ARENA** (N) : chunk 是 thread arena 的一部分

### *Free Chunk:*

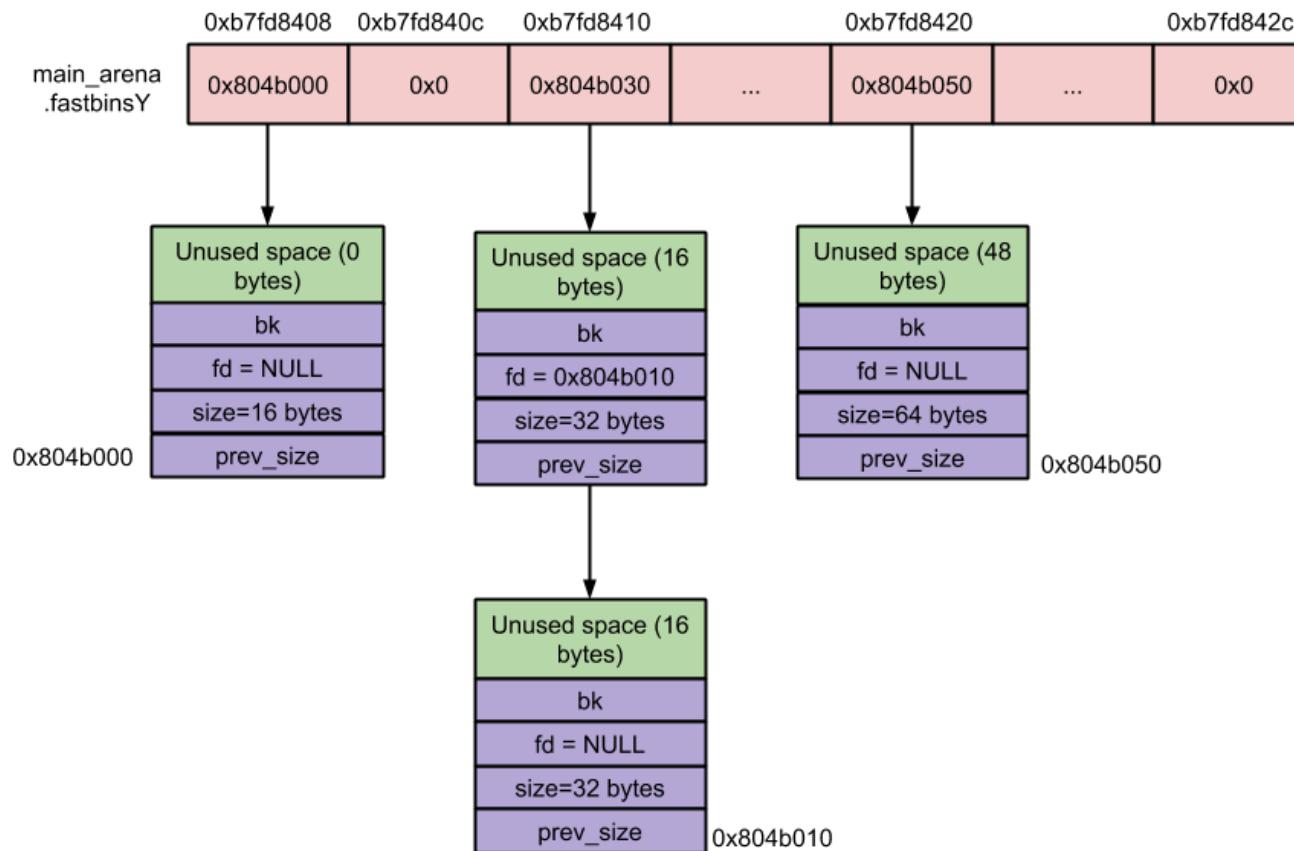


- **prev\_size** : 因兩個 free chunk 會合併，只會出現在 fast bin 的情況下
- fast bin 的 chunk 在 free 時不會做合併，只有在 malloc 步驟中的 consolidate fastbin(fast /small bin miss) 才會合併
- 下一個 chunk 的 PREV\_INUSE flag 平時是不 set 的
- **fd**(Forward pointer) : 指向在同一個 bin 中的下一個 chunk
- **bk**(Backward pointer) : 指向在同一個 bin 中的上一個 chunk

## Bins

bins 是紀錄 free chunks 的資料結構(freelist)，依據其大小和特性分成4種：

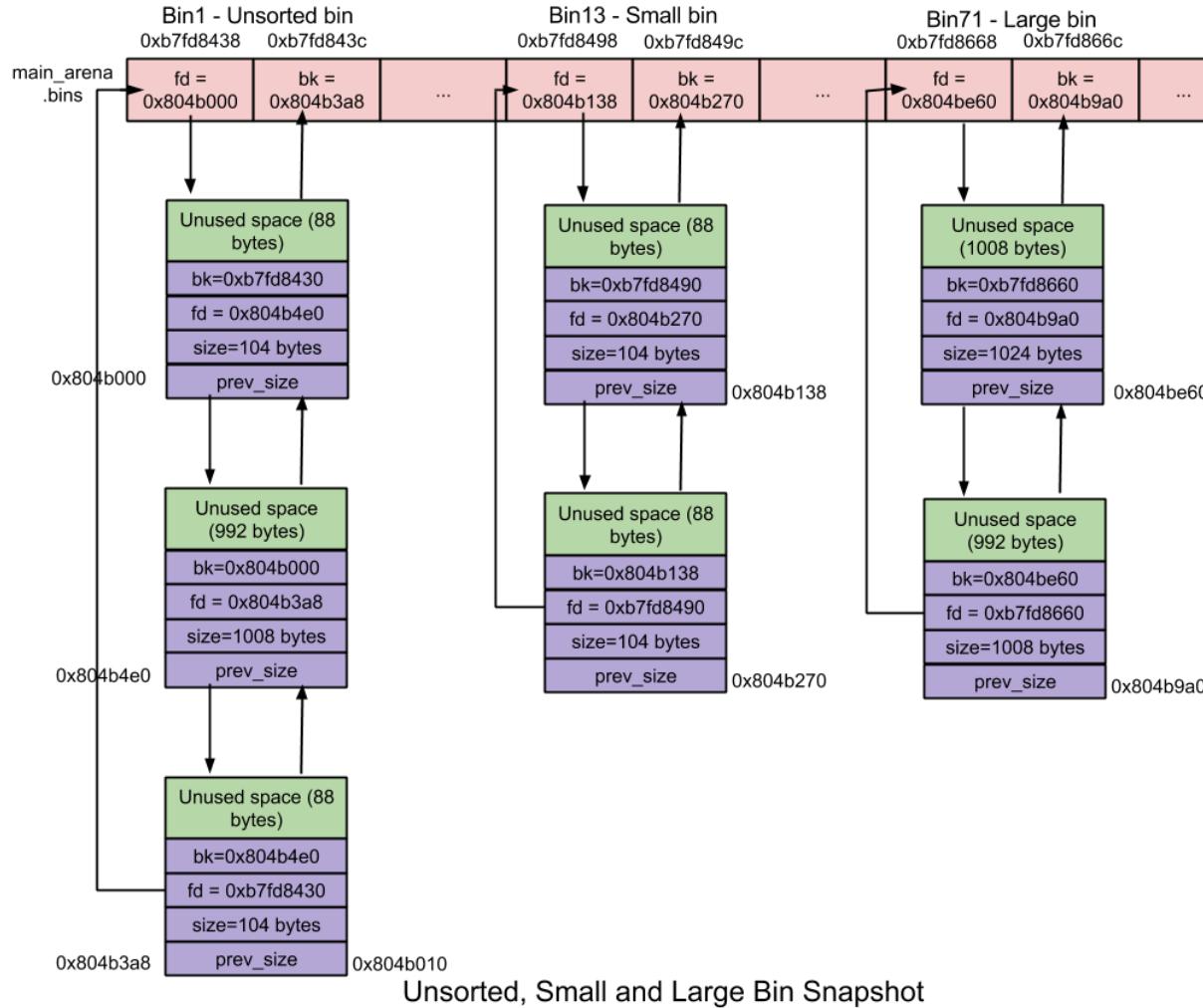
(以下的數值為32bit系統，64bit \*2)



Fast Bin Snapshot

*Fast Bin:*

- 10個 bin
- 使用 linked list, malloc 時 allocate 最後一項(pop from top)
- chunk 大小在16~最大80(可設定, 預設64) byte 之間
- size : 每個 bin 差8 bytes
- 不執行合併 : 在 free 時不會清除下一個 chunk 的 PREV\_INUSE flag



### Unsorted Bin:

- 最近 free 的 small / large chunk, 不分大小
- 重複使用 chunks 以加速 allocation

### Small Bin:

- chunk < 512 bytes
- 62 個 bin
- size : 每個 bin 差 8 bytes
- 合併 : 兩個相鄰的 free chunk 合併，減少 fragmentation (但較耗時)

*Large Bin:*

- chunk  $\geq 512$  bytes
- 63 個 bin
- size : 每個 bin 差距不同
  - 前 32 個 bin 差 64 bytes ( $n=1$ )
  - 再  $2^{(6-n)}$  個 bin 差  $8^{(n+1)}$  bytes
  - $n=6$  : 1 個 bin 262144 bytes
  - 剩最後一個 bin 存放所有更大的 chunk
- bin 裡的 chunk 依大小排序(因 size 差距不再是最小單位)，allocate 時需搜尋最適大小

### Top chunk

在 arena 邊界的 chunk，若所有 bin 皆無法 allocate 則由這裡取得，若仍不夠則用 brk / mmap 延展

### Last Remainder Chunk

最近一次分割的 large chunk，被用來滿足 small request 所剩下的部分

### malloc 流程

- 調整 malloc size : 加上 overhead 並對齊，若  $< 32$  byte (64bit 最小 size, 4 pointers) 則補上
- 檢查 fastbin : 若對應 bin size 有符合 chunk 即 return chunk
- 檢查 smallbin : 若對應 bin size 有符合 chunk 即 return chunk

- 合併(consolidate) fastbin : (若 size 符合 large bin 或前項失敗)呼叫 malloc\_consolidate 進行 fastbin 的合併(取消下一 chunk 的 PREV\_INUSE)，並將合併的 bin 歸入 unsorted
- 處理 unsorted bin :
  - 若 unsorted bin 中只有 last\_remainder 且大小足夠，分割 last\_remainder 並return chunk。剩下的空間則成為新的 last\_remainder
  - loop 每個 unsorted bin chunk，若大小剛好則 return，否則將此 chunk 放至對應 size 的 bin 中。此過程直到 unsorted bin 為空或 loop 10000次為止
  - 在 small / large bin 找 best fit，若成功則 return 分割的 chunk，剩下的放入 unsorted bin (成為 last\_remainder) ；若無，則繼續 loop unsorted bin，直到其為空
- 使用 top chunk : 分割 top chunk，若 size 不夠則合併 fastbin，若仍不夠則 system call

## free流程

- 檢查：檢查 pointer 位址、alignment、flag 等等，以確認是可 free 的 memory
- 合併(consolidate)
  - fastbin size : 不進行合併
  - 其他：
    - 檢查前一 chunk，若未使用則合併
    - 檢查後一 chunk，若是 top chunk 則整塊併入 top chunk，若否但未使用，則合併
  - 將合併結果放入 unsorted bin
- ref : [allocation過程](#)
  - **glibc malloc allocation informations**
  - [mallinfo](#) - obtain memory allocation information

- [malloc\\_stats](#) - print memory allocation statistics
- [malloc\\_info](#) - export malloc state to a stream

簡單測試：`malloc(40 * sizeof(int));`

- [mallinfo\(\) example](#)

```
Total non-mmapped bytes (arena): 135168
# of free chunks (ordblks): 1
# of free fastbin blocks (smblocks): 0
# of mapped regions (hblkds): 0
Bytes in mapped regions (hblkhd): 0
Max. total allocated space (usmblks): 0
Free bytes held in fastbins (fsmblks): 0
Total allocated space (uordblks): 176
Total free space (fordblks): 134992
Topmost releasable block (keepcost): 134992
```

## 待整理

---

- [Why does calloc exist?](#)
  - 對應的 [Hacker News 討論](#)