A Malloc Tutorial

Tác giả: Trần Thanh Tịnh.

Tài liệu chỉ có tính chất tham khảo cho đồ án TH KTLT. Mọi người chỉ nên đọc hiểu, thực hiện và thảo một bản báo cáo riêng. Tài liệu được dựa vào giáo trình Malloc Tutorial của tác giả Marwan Burelle. Link: http://www.inf.udec.cl/~leo/Malloc_tutorial.pdf

I. Giới thiệu:

- Malloc là gì? Nếu bạn thậm chí không cái tên này, bạn có thể bắt đầu học C trong môi trường Unix trước khi đọc hướng dẫn này. Đối với một lập trình viên, malloc là hàm để cấp phát bộ nhớ trong một chương trình C, hầu hết mọi người không biết rằng hàm này thực sự đang làm gì, thậm chí một vài người nghĩ rằng nó là một Syscall hoặc là một từ khóa. Sự thật là malloc không phải là một hàm đơn giản và có thể nắm hết được với một ít kiến thức về C và không có tí kiến thức gì về hệ thống.
- Mục đích của bài hướng dẫn này là lập trình một hàm malloc đơn giản nhằm hiểu hơn về các khái niệm đơn giản. Chúng ta sẽ không lập trình một hàm malloc phức tạp, chỉ là một hàm đơn giản, nhưng khái niệm có thể sử dụng để hiểu bộ nhớ được quản lý, cấp phát và giải phóng như thế nào.

Vậy malloc là gì:

➢ Malloc tuân thủ các quy tắc sau:

- Cấp phát ít nhất số byte được yêu cầu.
- Con trỏ được trả về từ malloc trỏ tới một vùng nhớ ở đó ta có thể đọc và viết thành công.
- Không có lần gọi nào từ malloc cấp phát nó hoặc bất kỳ phần nào của nó. Trừ khi con trỏ trỏ tới vùng nhớ này được giải phóng.

- Malloc là một hàm dễ xử lý vì vậy kết quả phải được trả về nhanh chóng.

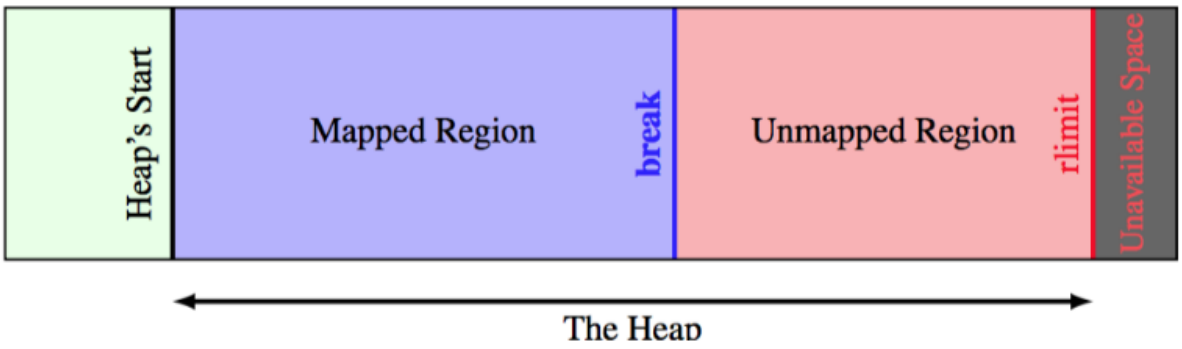
Cách gọi hàm: `void* malloc(size_t size);`

II. Headp và brk và sbrk:

- Trước khi viết một hàm malloc, chúng ta cần hiểu bộ nhớ được quản lý như thế nào. Chúng ta sẽ giữ các khái niệm trừu tưởng cho môn hệ điều hành giải quyết, vì ở đây tôi cũng chả biết. hihi

1. Tiến trình của bộ nhớ:

- Mỗi quá trình có không gian địa chỉ ảo riêng của nó tự động chuyển thành bộ nhớ vật lý, đỉa chỉ của MMU (kernel-search google để biết thêm). Vùng này được chia phần, tất cả những gì chúng ta cần biết là chúng ta phải tìm kiếm ít nhất vài khoảng trống để code, một stack nơi mà lưu giữ biến địa phương và có thể thay đổi dữ liệu đã lưu giữ, một vài vùng cho hằng số và giá trị cục bộ, vùng này được gọi là heap.
- Heap là một vùng nhớ liên tiếp của bộ nhớ với 3 vùng:
  - Starting point: điểm khởi đầu.
  - Maximum limit: điểm kết thúc, là giá trị quy định trong sys/resource.h, thể hiện vùng nhớ max có thế láy được thông qua cấp phát động.
  - Break: đánh dấu nơi kết thúc của vùng nhớ, một phần của địa chỉ ảo, thể hiện đã có bao nhiêu vùng nhớ đã được sử dụng.



- Để code được malloc thì chúng ta chỉ cần biết 2 giá trị: địa chỉ vùng nhớ bắt đầu heap (starting point), và vị trí vùng nhớ của break point. Và để biết được 2 giá trị này, kernel cấp cho chúng ta 2 hàm syscall là brk và sbrk. Lưu ý lại với bạn đọc là chúng ta không thể thao tác với vùng nhớ được quản lý bằng kernel bằng việc "tay không bắt giặc" được, mà phải thông qua các hàm low level hơn được cung cấp bởi kernel.

2. Brk và sbrk:
   - brk và sbrk là 2 hàm dùng để thao tác với vùng nhớ heap ở mức kernel. Chúng ta hãy xem signature của chúng nhé:

     ```
     int brk(const void *addr);
     void* sbrk(intptr_t incr);
     ```

   - Cả 2 hàm này đều dùng để thao tác với "break position", bạn có thể hình dung nó như một cái rào chắn. Kernel cung cấp cho chúng ta bộ tool để thao tác với cái rào chắn đó. Mỗi lần cần cấp phát vùng nhớ mới cho process sử dụng, bạn sử dụng những hàm này để "nâng" cái rào chắn lên, nhằm báo cho kernel là: tao dùng đến đây nhé, nhớ đừng cho thằng khác đụng vào vùng nhớ của tao.

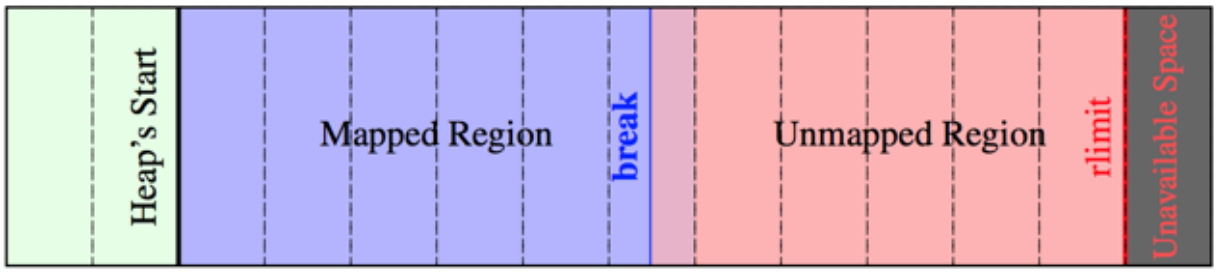
   - Cả 2 hàm đều có nhiệm vụ là "nâng rào chắn", tuy nhiên có chút khác nhau về interface: hàm brk sẽ đặt rào tại địa chỉ được truyền vào addr, còn hàm sbrk sẽ nâng rào lên thêm incr bytes kể từ vị trí hiện tại. Rõ ràng là hàm sbrk cho chúng ta một interface gần với hàm malloc hơn, tiện hơn, còn nếu sử dụng brk chúng ta sẽ phải tự tính toán địa chỉ nên sẽ mất công hơn. Do đó trong bài viết này chúng ta sẽ sử dụng sbrk là chính.

3. Vùng nhớ unmapped và vùng nhớ tự do:

   - Trờ lại hình của vùng nhớ Heap ở trên, chúng ta thấy có 2 vùng nằm phía sau rào chắn là Unmapped Region và Unvailable Space. Nếu sử dụng phải phần đằng sau rào chắn thì chúng ta

sẽ gặp phải lỗi bus error. Bạn có thể tìm hiểu thêm về lỗi này tại wiki.

- Một điểm cần lưu ý nữa là vùng nhớ nói chung được kernel quản lý theo trang (Pages), tức là theo từng cụm 4096 bytes một sẽ là một trang (trên hầu hết các hệ thống hiện tại).



- Ở hình trên chúng ta có thể thấy khi địa chỉ của break không nằm ở một vị trí chia hết cho 4096, thì hiển nhiên địa chỉ break sẽ nằm lơ lửng trong 1 page. Vậy sẽ có một "đoạn" bộ nhớ nằm từ vị trí break đến địa chỉ của page kế tiếp, và đoạn bộ nhớ đó có sử dụng được không? Thực tế là đoạn bộ nhớ đó **có thể sử dụng được**, và nó được gọi là vùng nhớ tự do. Vùng nhớ này chính là nguyên
  nhân của rất nhiều loại bug khi bạn thao tác không chuẩn với pointer, khiến cho nó "nhảy" nhầm đến vùng nhớ tự do này.

4. Tự làm một hàm malloc đơn giản:

```
#include <sys/types.h>
#include <unistd.h>

void *malloc(size_t size) {
 void* p;
 p = sbrk(0);
 if (sbrk(size) == (void*)-1) return NULL;
 return p;

}
```

Đoạn code làm những điều khá đơn giản

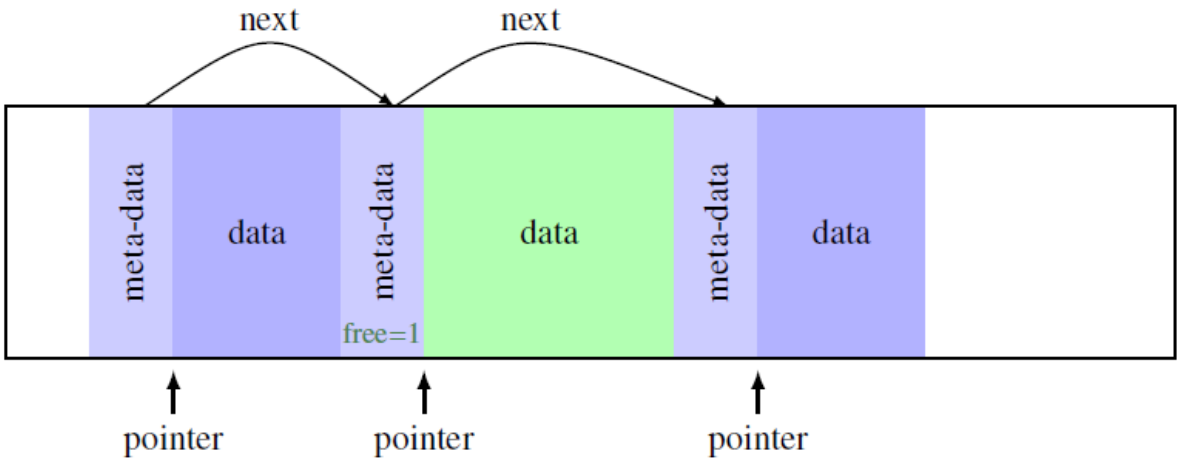
- Đầu tiên gọi sbrk(0) là để lưu lại địa chỉ của rào chắn hiện tại để trả về vị trí hiện tại của rào chắn vào biến p
- Sau đó gọi sbrk(size) để nâng rào chắn thêm size bytes
- Cuối cùng trả về vị trí của p để có thể sử dụng vùng nhớ vừa được cấp phát

Xem xét ngoài góc độ lập trình:

- Giả sử bạn có một cánh đồng bao la rộng lớn cỡ 2GB. Bạn có 2 cách cho thuê là sử dụng kích thước của một mẫu và dựng hàng rào. Đương nhiên là phải liên tục rồi, ai mà đi thuê từng mảnh nhỏ chứ. Khi khách hàng trả lại mẫu đất ta có thể cho thuê tiếp, mặc kệ trên đó còn lúa hay không (nghĩa là khi giải phóng thì những gì trên mẫu đất vẫn còn cho đến khi khách hàng tiếp theo đến và sang bằng nó).
- Một điểm quan trọng là bạn có thể cho thuê mảnh đất của người khác không (nằm ngoài heap). Suy nghĩ thử nhé, ngồi tù mấy năm nhỉ. Hihi.

5. Làm thế nào đánh dấu vùng nhớ đã cấp phát nhỉ?

- Vì vậy những gì ta cần là một biển báo ở đầu mỗi đoạn có chứa các thông tin, được gọi là meta-data. Khối này bao gồm ít nhất là một con trỏ đến biển báo tiếp theo, một biển báo đánh dấu miễn phí khối và kích thước của vùng dữ liệu. Đương nhiên là khối thông tin này là trước khi con trỏ trả về bởi malloc (người yêu còn có thể đổi được mà, malloc là cái chi chi mà không thay đổi ahihi).



- Hình trên trình bày một ví dụ về tổ chức heap với meta-data ở phía trước của phân bố khối. Mỗi vùng dữ liệu bao gồm một khối dữ liệu, theo sau là một khối dữ liệu khác.

- Con trỏ trả về bởi malloc được chỉ ra ở phần dưới của biểu đồ, lưu ý rằng trỏ đến trên khối dữ liệu, không phải trên đoạn hoàn chỉnh.

- Nào! Nghĩ xem đây giống gì nhỉ? Mình đoán bạn cũng nghĩ giống mình, đây là danh sách liên kết mà. Cùng xem nó được viết như thế nào:

```
1  typedef struct s_block *t_block;
2
3  struct s_block {
4    size_t        size;
5    t_block       next;
6    int           free;
7  };
```

Sao nhỉ? Nhìn có vẻ như nó là một sự lãng phí không gian để sự dụng một kiểu int cho một lá cờ. Nhưng vì struct được sắp xếp bởi kiểu mặc định, nó sẽ không thay đổi bất cứ thứ gì, chúng ta sẽ thấy sau này chúng ta có thể làm giảm kích thước của siêu dữ liệu như thế nào. Một điểm khác mà chúng ta sẽ thấy đó là malloc phải trả về địa chỉ. Một câu hỏi nhức cả đầu là làm thế nào chúng ta có thể tạo ra một cấu trúc mà không có hàm malloc làm việc? Câu trả lời là, bạn chỉ cần biết những gì thực sự là một struct block. Trong

bộ nhớ một struct đơn giản là sự nối liền của các trường, vì vậy trong ví dụ cảu chúng ta một struct s block chỉ 12byte (với số nguyên 32bit).

- Khi trình biên dịch gặp phải một truy cập vào trường struc t này (giống như s.free hoặc p->free) nó dịch đến địa chỉ của cơ sở struct cộng với tổng chiều dài của trường trước (vì vậy p->free tương đương với *((char*)p+8) và s.free là *((char*)&s+8). Tất cả những gì phải làm là phân bổ đủ không gian với sbrk.

  - Nói dài dòng: tóm lại là sử dụng s_block và đủ bộ nhớ để dịch sbrk.

```
/* Example of using t_block without malloc
t_block         b;
/* save the old break in b
b = sbrk(0);
/* add the needed space
/* size is the parameter of malloc
sbrk(sizeof(struct s_block) + size);
b->size = size;
/*
```

6. Biển báo truyền kì:

- Ở đây thường được yêu cầu rằng các con trỏ được căn chỉnh với kích thước số nguyên. Trong môi trường 32bit con trỏ của chúng ta phải là một bội số của 4 (32bit = 4 chứ gì nữa). Kể từ khi data meta của chúng ta được liên kết điều chúng ta cần làm là sắp xếp kích thước của khối dữ liệu.

- Ở đây chúng ta sẽ trải qua một combo toán học nhức đầu và sấp mặt **, vì thế mình chỉ để đây ai muốn biết thì có thể gg hoặc liên hệ mình.

```
#define align4(x) ((((x)-1)>>2)<<2)+4)
```

7. Tìm đoạn tiếp theo: con đường xưa em đi:

   - Việc tìm kiếm một đoạn đủ rộng khá đơn giản: Chúng ta bắt đầu tại địa chỉ cơ sở của heap (phần này hồi sau sẽ rõ) kiểm tra đoạn hiện tại, nếu nó "đủ rộng" thì còn chần chờ gì nữa come on baby thôi. Trả lại địa chỉ nhà của nó thôi. Nếu không đủ yêu cầu thì ta lượn qua chỗ khác cho đến khi tìm được một nơi ưng ý. Điều quan trọng duy nhất là giữ vùng cuối cùng, vì vậy malloc có thể dễ dàng vượt ra khỏi heap nếu chúng ta không tìm thấy vùng cần thiết. Cùng xem chương trình ăn gì tối nay nào:

```
t_block find_block(t_block *last, size_t size){
  t_block b=base;
  while (b && !(b->free && b->size >= size)) {
    *last = b;
    b = b->next;
  }
  return (b);
}
```

   Hàm trả về vùng nhớ phù hợp hoặc NULL nếu không tìm thấy. Sau khi hoàn thành con trỏ last sẽ trỏ đến vùng nhớ cuối mà chúng trỏ đến.

8. Cửu âm chân kinh: (tham khảo, giỏi quá không ai chơi đâu)

   - Đọc rồi thì cứ đọc thôi: bây giờ chúng ta thường xuyên không tìm thấy vùng nhớ hợp lệ, chúng ta cần mở rộng heap. Nó đơn giản thôi, chúng ta cần di chuyển break và khởi tạo các block mới, dương nhiên là chúng ta cần nâng cấp các trường tiếp theo của block cũ trên heap.

   - Trong phát triển sau này, chúng ta cần làm một vài thứ với kích thước của struct s block, vì vậy chúng ta cần định nghĩa

một macro để giữ kích thước của meta-data block, cùng xem môn võ công này nào:

```
#define BLOCK_SIZE sizeof(struct s_block)
```

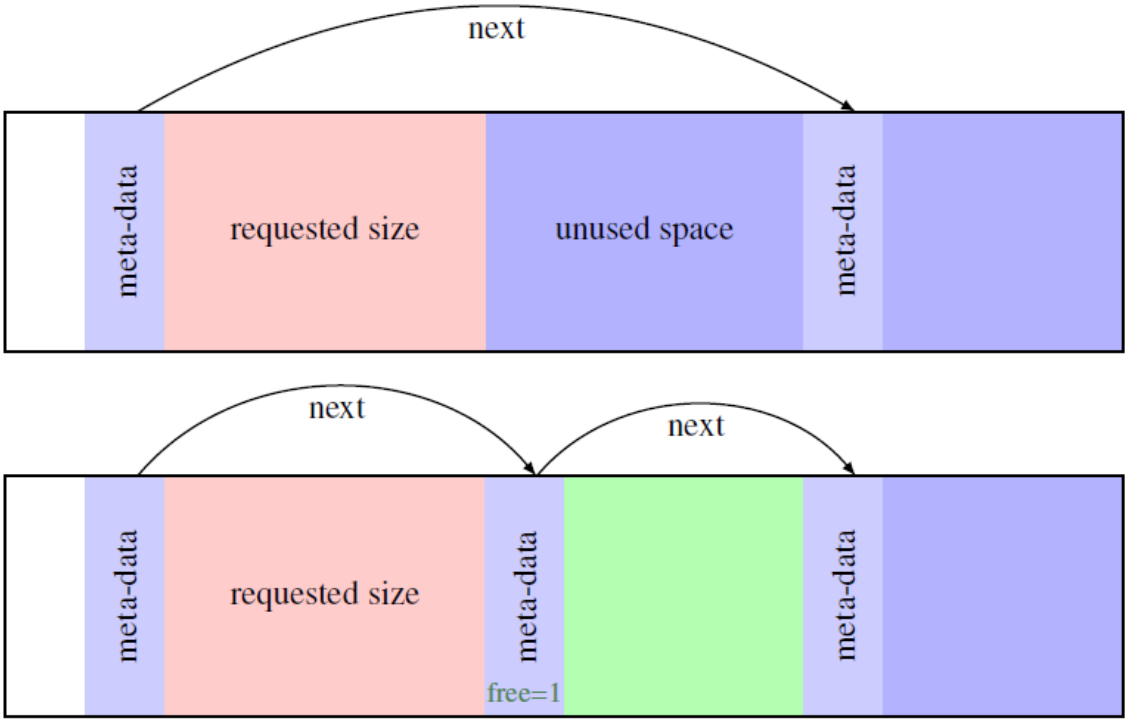
- Không có gì hay ho nhỉ, chỉ là tự cung trước khi luyện thôi mà. Chúng ta chỉ trả về NULL nếu sbrk không thành công (đừng cố gắng hiểu vì sao, kệ nó đi). Chỉ chú ý rằng chúng ta không chắc chắn rằng sbrk sẽ trả lại break trước đó. Đầu tiên, chúng ta lưu nó và di chuyển break. Chúng ta có thể tính toán bằng cách sử dụng last và last-size.

```
t_block extend_heap(t_block last, size_t s){
  t_block        b;
  b = sbrk(0);
  if (sbrk(BLOCK_SIZE + s) == (void*)-1)
    /* sbrk fails, go to die */
    return (NULL);
  b->size = s;
  b->next = NULL;
  if (last)
     last->next = b;
   b->free = 0;
   return (b);
}
```

9. Môn đầu tiên - cắm biển báo:

- Bạn có thể thông báo rằng chúng tôi sử dụng khối đầu tiên có sẵn bất kể kích thước của nó (cung cấp rằng nó đủ rộng). Nếu chúng ta làm điều đó, sẽ mất rất nhiều không gian (suy nghĩ xem: bạn yêu cầu 2 bytes và tìm một khóa của 256 bytes, nếu bạn là người hào phóng ahihi).

- Giải pháp đầu tiên là chia khối: khi một đoạn đủ rộng đủ để giữ kích thước của yêu cầu (ít nhất là BLOCK SIZE + 4), chúng ta sẽ chèn một đoạn mới vào danh sách.



- Ngầu chưa ahihihihi!

- Chỉ sử dụng kungfu này khi có đất để thể hiện thôi nha. Sự cung cấp kích thước cần phải được liên kết, trong hàm này chung ta sẽ có vài con trỏ, để tránh lỗi chúng ta sẽ sử dụng vài kĩ xảo để chắc rang tất cả hoạt động, chúng ta làm việc với độ chính xác 1byte (nhớ rằng p + 1 phụ thuộc vào kiểu được con trỏ trỏ tới).

- Chúng ta chỉ cần thêm 1 trường khác vào khối struct của kiểu mảng ký tự. Mảng trong cấu trúc chỉ đơn giản là dặt ở đó, mảng nắm trực tiếp trong toàn bộ memory block của cấu trúc tại con trỏ đã định nghĩa, vì vậy con trỏ của mảng chỉ ra sự kết thúc của meta-data.

- C cấm mảng có độ dài bằng 0, vì vậy chúng ta định nghĩa một mảng dài một byte và điều này giải thích tại sao chúng ta cần chỉ định một macro cho kích thước của struct s block.

```
struct s_block {
  size_t        size;
  t_block       next;

  int           free;
  char          data[1];
};

/* Of course, update the BLOCK_SIZE macro */
#define BLOCK_SIZE 12 /* 3*4 ... */
```

- Lưu ý rằng tiện ích mở rồng này không yêu cầu bất kỳ sửa đổi nào để mở rộng heap kể từ khi trường mới không sử dụng trực tiếp.

- Bây giờ, chia đất nào, hàm này sẽ cắt block thông qua đối số để làm data block với kích cỡ mong muốn. Hình 4 trang trước cho thấy những gì được thức hiện ở đây.

```
void split_block(t_block b, size_t s){
  t_block        new;
  new       = b->data + s;
  new->size = b->size - s - BLOCK_SIZE;
  new->next = b->next;
  new->free = 1;
  b->size   = s;
  b->next   = new;
}
```

- Lưu ý rằng sử dung b->data trong con trỏ trên dòng 3. Vì dữ liệu của trường là char[], chúng ta chắc chắn rằng tổng được thực hiện bởi độ chính xác byte.

10. Lăng 3 vi bộ 10 thành – malloc fuction:

- Bây giờ chúng ta có thể sự dụng hàm malloc của chúng ta. Nó chủ yếu là một cái vỏ xung quanh các chức năng trước đó.

Chúng ta có yêu cầu kích thước size, kiểm tra nếu chúng ta là người gọi đầu tiên hay không, và những thứ khác đã được gọi chưa. Và trong phần trước hàm find block sử dụng một biến toàn cục và nó được đĩnh nghĩa như sau:

```
void *base=NULL;
```

- Nó là một con trỏ void và nó là được khởi tạo là NULL. Điều đầu tiên mà malloc làm là kiểm tra, nếu nó NULL thì đó là lần đầu chúng được lên dĩa, không thì chúng ta bắt đầu tử lần trước đó.

- Malloc làm việc như sau:

  - Căn chỉnh kích thước yêu cầu.

  - Nếu base được khởi tạo:

➢ Tìm kiếm vùng đủ rộng.

➢ Nếu tìm thấy: chia tách ra, đánh dấu chủ quyền b->free = 0;

  - Nếu không khởi tạo: mở rộng heap. Nhớ đánh dấu lần trước là tới đâu nhé.

Cùng tận hưởng nét đẹp của em nó:

```
void *malloc(size_t size){
  t_block                b,last;
  size_t                 s;
  s = align4(size);
  if (base) {
    /* First find a block */
    last = base;
    b = find_block(&last,s);
    if (b) {
      /* can we split */
      if ((b->size - s) >= (BLOCK_SIZE + 4))
        split_block(b,s);
      b->free=0;
    } else {
      /* No fitting block, extend the heap */
      b = extend_heap(last,s);
      if (!b)
        return(NULL);
    }
  } else {
    /* first time */
    b = extend_heap(NULL,s);
    if (!b)
      return(NULL);
    base = b;
  }
  return(b->data);
}
```

11. Calloc, free and realloc:

a) Calloc:

- Như malloc thôi, có điều đặt 0 vào. Xem em này có đẹp không nhé:

```
void *calloc(size_t number, size_t size){
  size_t       *new;
  size_t        s4,i;
  new = malloc(number * size);
  if (new) {
    s4 = align4(number * size) << 2;
    for (i=0; i<s4 ; i++)
      new[i] = 0;
  }
  return (new);
}
```

b) Free:

- Việc giải phóng có thể nhanh chóng và đơn giản nhưng không có nghĩa là nó chính xác, chúng ta có 2 vấn đề: tìm đoạn để được giải phóng và tránh sự phân mảnh dẫn đến rác bộ nhớ.

- Vấn đề lớn đó là sự phân mảnh, sau khi sử dụng malloc và free, chúng ta có thể kết thúc với một vùng nhớ heap bị chia mảnh bởi việc sử dụng malloc.

  Vậy giải quyết như thế nào: giải pháp nằm ở việc giải phóng, khi chúng ta giải phóng một đoạn chúng ta giải phóng luôn vùng nhớ gần đó, chúng ta có thể gọp chúng vào một vùng nhớ lớn hơn. Tất cả những gì chúng ta cần làm là kiểm tra khối tiếp theo và trước đó. Nhưng làm thể nào để tìm đây? Haizzz. Thiệt là khổ quá mà.

  - Thôi thì cũng phải giải thôi:

Giải pháp:

- Tìm kiếm từ đầu, chậm nhưng phải chịu.

- Nếu tìm rồi thì đặt con trỏ đánh dấu.

Phần quan trọng đây rồi: bung link nào

```
t_block fusion(t_block b){
  if (b->next && b->next->free){
    b->size += BLOCK_SIZE + b->next->size;
    b->next = b->next->next;
    if (b->next)
      b->next->prev = b;
  }
  return (b);
}
```

- Nếu vùng nhớ kế tiếp đã giải phóng, chúng ta tổng hợp kích thước của đoạn hiện tại và một đoạn kế tiếp rồi cộng với meta-data size. Tiếp, làm con trỏ cho người dùng kế tiếp có thể biết.

- Người sau lại cập nhật cho người sau.

❖ Cách tìm đúng đoạn:

- Vấn đề ở đây là tìm như thế nào cho hiệu quả, chính xác, với trả về con trỏ của malloc. Thực tế có các vấn đề sau:

  - Xác nhận con trỏ đầu vào (nó thực sự là con trỏ của malloc).

  - Tìm con trỏ meta-data.

- Chúng ta không thể loại bỏ hầu hết các con trỏ không hợp lệ bằng phép thử nhanh, nếu con trỏ nằm ngoài heap, nó không hợp lệ. Các trường hợp còn lại thì sao, làm sao biết

nó được trả về bởi malloc. Chắc phải có phép mới biết được.

- Oh my god! Có phép thật, chúng ta có thể sự dụng chính con trỏ đó. Nếu nói rằng chúng ta có con trỏ ptr trỏ đến data, nếu b->ptr == b->data, sau đó b có lẽ là một vùng hợp lệ. Vì vậy, ở đây là kiểu cấu trúc mở rộng và truy cập block với một con trỏ:

```
/* block struct
struct s_block {
  size_t              size;
  struct s_block    *next;
  struct s_block    *prev;
  int                free;
  void              *ptr;
  /* A pointer to the allocated block */
  char                          data[1];
};
typedef struct s_block *t_block;

/* Get the block from and addr
t_block get_block(void *p)
```

```
{
  char *tmp;
  tmp = p;
  return (p = tmp -= BLOCK_SIZE);
}

/* Valid addr for free
int valid_addr(void *p)
{
  if (base)
    {
      if ( p>base && p<sbrk(0))
        {
          return (p == (get_block(p))->ptr);
        }
    }
  return (0);
}
```

❖ Full hd hàm free xem nào:

Nếu con trỏ hợp lệ:

- Chúng ta nhận được địa chỉ vùng nhớ đó.
- Chúng ta đánh dấu đã giải phóng.
- Nếu đã tồn tại thì giải phóng, lùi lại trong danh sách block và hợp nhất 2 khối.
- Chúng ta thử kết hợp với khối tiếp theo sau đó.
- Nếu chúng ta có last block thì chúng ta giải phóng.

- Nếu không có khối nào nữa, quay lại nơi ta bắt đầu (đặt cơ sở là NULL).

Nếu con trỏ không hợp lệ, im lặng và ra đi, không làm gì nữa.

c) Realloc – Thái cực quyền:

- Hàm này quá dễ, xem nào, không cần giải thích đâu nhỉ.

```
/* The free
/* See free(3)
void free(void *p)
{
  t_block b;
  if (valid_addr(p))
    {
      b = get_block(p);
      b->free = 1;
      /* fusion with previous if possible */
      if(b->prev && b->prev->free)
        b = fusion(b->prev);
      /* then fusion with next */
      if (b->next)
        fusion(b);
      else
        {
          /* free the end of the heap */
          if (b->prev)
            b->prev->next = NULL;
          else
            /* No more block !*/
            base = NULL;
          brk(b);
        }
    }
}
```

```
/* Copy data from block to block                                    *
void copy_block(t_block src, t_block dst)
{
  int           *sdata,*ddata;
  size_t         i;
  sdata = src->ptr;
  ddata = dst->ptr;
  for (i=0; i*4<src->size && i*4<dst->size; i++)
    ddata[i] = sdata[i];
}
```

Ngoài ra, còn có một cuốn bí kíp đơn giản nhưng chạy được:

Luyện như sau:

- Cấp phát bằng malloc
- Sao chép data
- Giải phóng cũ
- Trả lại chỗ mơi

Dể không nào! Ahihi

Nhưng mà, cái gì dễ quá thì không tốt. Chúng ta muốn một cái gì hiệu quả hơn phải không nào, chúng ta sẽ không muốn cấp phát nếu chúng ta không có đủ chỗ đâu nhỉ. Các trường hôp như là:

- Nếu kích thước không đổi thì ném qua ném lại làm gì cho mệt sức.
- Nếu co lại thì sao nhỉ.
- Nếu khối kế tiếp giải phóng và cung cấp đủ, chúng ta gọp lại luôn, chỉ chia nếu cần thiết.

- Đừng quên: realloc(NULL, s) là hợp lệ và có thể thay thế bởi malloc(s);

  Nào cùng xem full hd không **e nào:

```
/* The realloc                                          */
/* See realloc(3)                                       */
void *realloc(void *p, size_t size)
{
  size_t        s;
  t_block       b, new;
  void         *newp;
  if (!p)
    return (malloc(size));
  if (valid_addr(p))
    {
      s = align4(size);
      b = get_block(p);
      if (b->size >= s)
        {
          if (b->size - s >= (BLOCK_SIZE + 4))
            split_block(b,s);
        }
      else
        {
          /* Try fusion with next if possible */
          if (b->next && b->next->free
              && (b->size + BLOCK_SIZE + b->next->size) >= s)
            {
              fusion(b);
              if (b->size - s >= (BLOCK_SIZE + 4))
                split_block(b,s);
            }
```

```
            else
              {
                /* good old realloc with a new block */
                newp = malloc(s);
                if (!newp)
                  /* we're doomed ! */
                  return (NULL);
                /* I assume this work ! */
                new = get_block(newp);
                /* Copy data */
                copy_block(b,new);
                /* free the old one */
                free(p);
                return (newp);
              }
          }
        return (p);
      }
  return (NULL);
}
```

Vài phiên bản khác nhé:

```
/* The reallocf
/* See reallocf(3)
void *reallocf(void *p, size_t size)
{
  void *newp;
  newp = realloc(p,size);
  if (!newp)
    free(p);
  return (newp);
}
```

Next:

```c
/* block struct
struct s_block {
  size_t             size;
  struct s_block    *next;
  struct s_block    *prev;
  int                free;
  void              *ptr;
  /* A pointer to the allocated block */
  char                          data[1];
};
typedef struct s_block *t_block;

/* Define the block size since the sizeof will be wrong
#define BLOCK_SIZE 20

/* Split block according to size.
/* The b block must exist.
void split_block(t_block b, size_t s)
{
  t_block       new;
  new       = (t_block)(b->data + s);
  new->size = b->size - s - BLOCK_SIZE;
  new->next = b->next;
  new->prev = b;
  new->free = 1;
  new->ptr  = new->data;
  b->size   = s;
  b->next   = new;
  if (new->next)
    new->next->prev = new;
}
```

```c
/* Add a new block at the of heap
/* return NULL if things go wrong
t_block extend_heap(t_block last, size_t s)
{
  int            sb;
  t_block        b;
  b        = sbrk(0);
  sb       = (int)sbrk(BLOCK_SIZE + s);
  if (sb < 0)
    return (NULL);
  b->size = s;
  b->next = NULL;
  b->prev = last;
  b->ptr  = b->data;
  if (last)
    last->next = b;
  b->free = 0;
  return (b);
}
```