

EPITA - Practical Programming



Memory

Overview

1. Multi-task And Memory Management
 - a. Issues ?
 - b. Virtual Memory
2. Processus Memory Organization
3. malloc(3)
 - a. Heap Management ?
 - b. A Simple First Fit Allocator
4. Other Kind Of Allocators

Multi-Task And Memory

Issues ?

- Sharing one memory with several processes
- Each process wants contiguous memory blocks
- We need confinement between processes
- How to deal with dynamic memory
- Binary program may not be relocatable
- ...

Solutions

- Fixed size memory partitions: *too much constraints*
- Full swapping: *way too slow ...*
- Dynamics partitioning: *continuity issues and probably no possible confinement ...*
- Relocatable code: *complex and risky*

Can we do better ?

Virtual Memory

*Virtual Memory maps memory addresses used by a program, called **virtual addresses**, into **physical addresses** in computer memory.*

Virtual Memory

- All memory accesses are rewritten on the fly
- Each process has its own virtual address space
- Rewriting is done by the MMU (hardware)
- Rewriting tables are managed by the Kernel

Benefits

- Memory is always contiguous for each process
- Memory has a fixed layout for each process
- No swapping are required
- Each process is alone in its virtual address space

Virtual Memory provides a complete solution.

Mapping ?

- Can we do a one-to-one mapping ?
- A translation table needs 8 bytes (64bits) per translated address, let's do some math ...
- For n bytes of virtual memory, we need $8n$ bytes

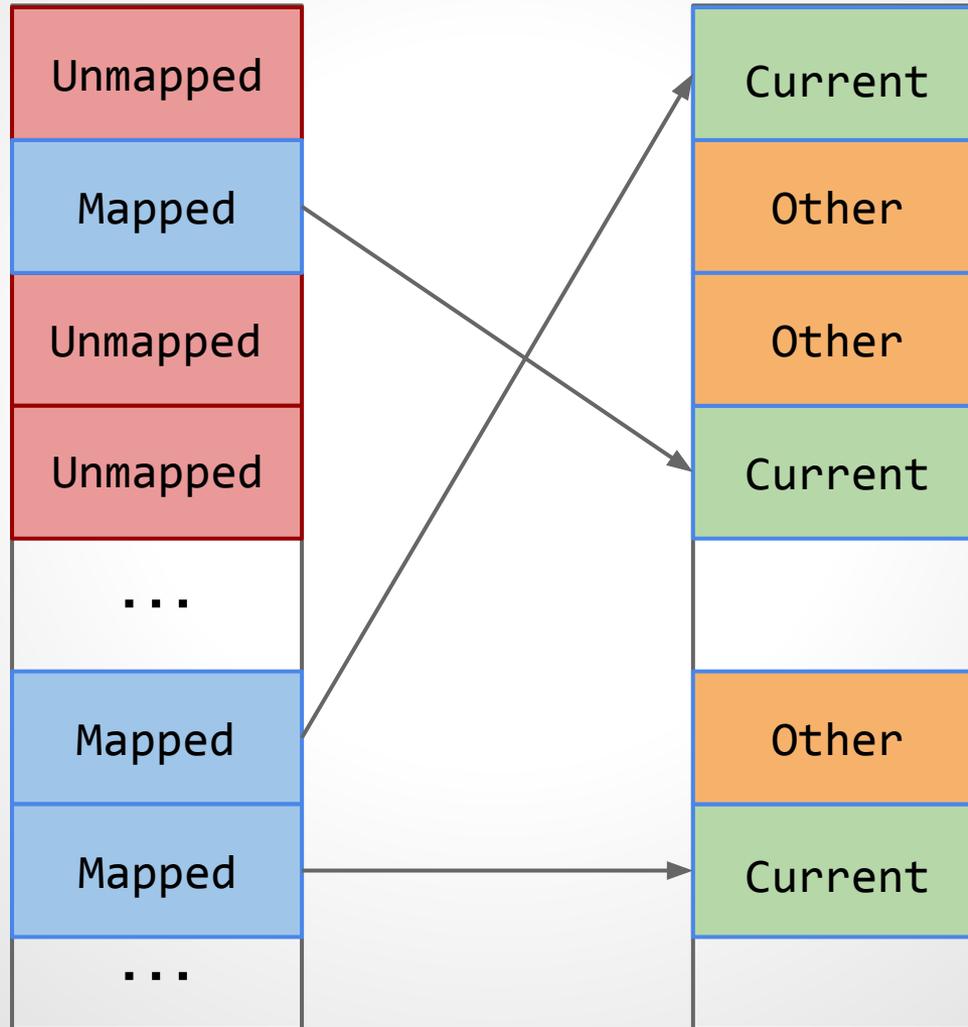
To map the memory we need 8 times its size !

Frames and Pages

- Memory is splitted in blocks
- Block in physical memory is called **frame**
- Block in virtual memory is called **page**
- We only translate page base into fram base
- Page/Frame base addresses are shorter
- Usually, page/frame have fixed size

Virtual Memory

Virtual
Memory



Physical
Memory

Pages Translation Example

Intel ia32 (32bits) schema:

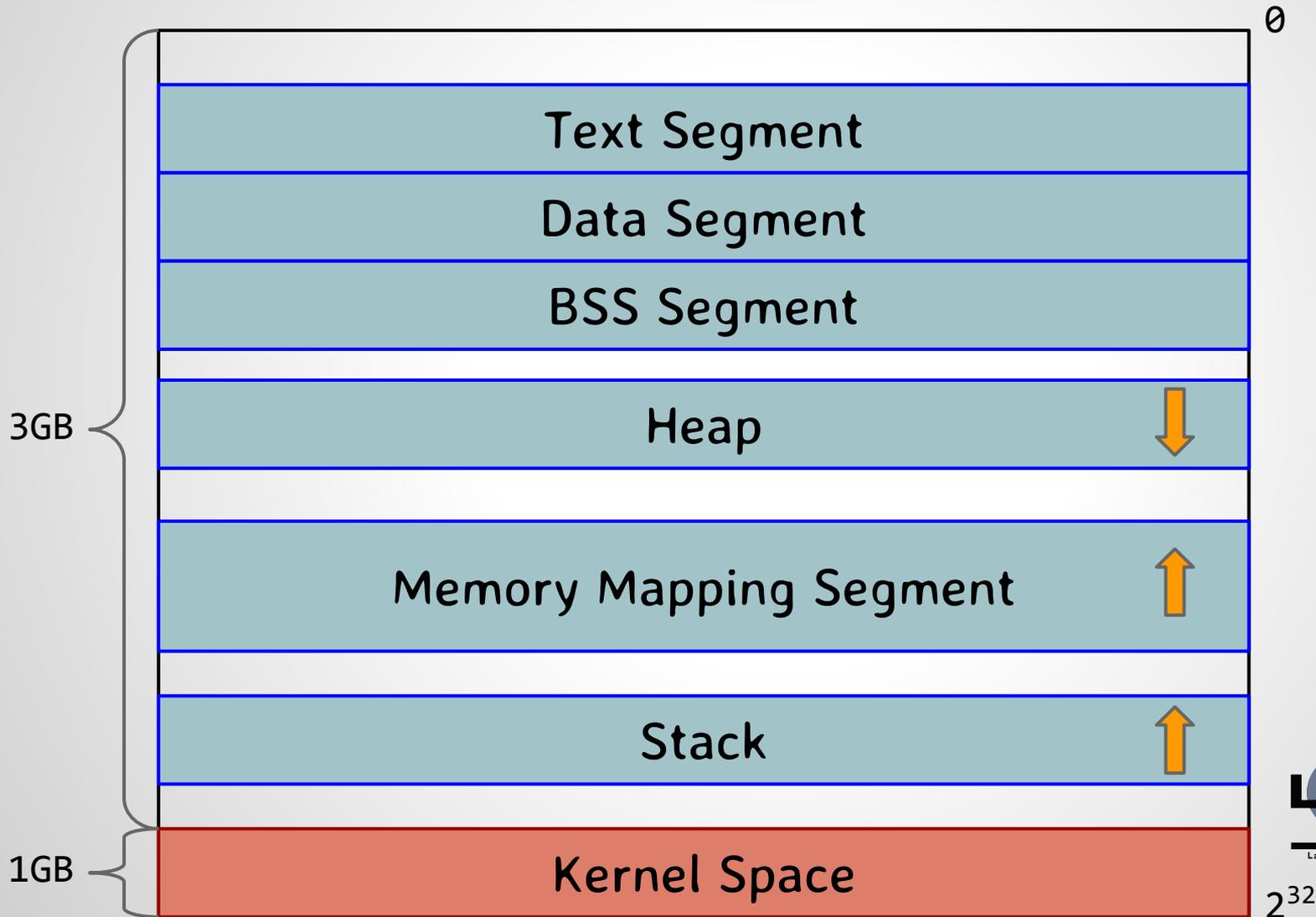
- We use 4KB pages (12bits)
- Pages are grouped in page directory
- Translation is done in two step:
 1. Translate the page directory base address
 2. Translate the page base address
- The page directory table contains 1024 entry of 4 bytes each pointing to page table of the same size. A full mapping takes at most 4MB for 4GB.

Process Memory

Memory Organization

- Each process has its own address space
- The memory is divided in *segments*:
 - Text Segment: the process image (code)
 - Data Segment: static initialized variables
 - BSS Segment: static uninitialized variables (fill with 0)
 - Heap: where dynamic allocation should take place
 - Memory Mapping Segment: mapped files, libs ...
 - Stack: process (main thread) stack
- The kernel is also mapped, in the higher part of the memory.

Process Layout (Linux 32bits)



Where're My Variables ?

- Global variables belongs to the BSS Segment
- Global constants belongs to the Data Segment
- Local variables are on the stack
- Dynamic allocation is done on the heap
- Parameters use registers, if possible, and the stack.

Where're My Variables

```
void f(int x) {  
    // x is in a register but has  
    // an address on the stack  
    int y; // on the stack  
    int tab[8]; // on the stack  
    char *s1 = "a constant string";  
    // s1 is on the stack but points  
    // to the Data segment  
    char *s2 = malloc(32);  
    // s2 is on the stack but points  
    // to the heap  
    char s3[] = "a local string";  
    // the string is on the stack
```

```
    // Let's print some addresses  
    printf("x : %p\n",&x);  
    printf("y : %p\n",&y);  
    printf("tab : %p\n",tab);  
    printf("&tab: %p\n",&tab);  
    printf("s1 : %p\n",s1);  
    printf("s2 : %p\n",s2);  
    printf("s3 : %p\n",s3);  
    printf("&s3 : %p\n",&s3);  
}
```

Where're My Variables

```
x      : 0x7fffffffffd5ec
y      : 0x7fffffffffd5e8
tab    : 0x7fffffffffd5c0
&tab  : 0x7fffffffffd5c0
s1     : 0x400732
s2     : 0x800c07040
s3     : 0x7fffffffffd5b0
&s3   : 0x7fffffffffd5b0
```

Static Arrays

```
int    tab[8];
```

- Static arrays always verify: (tab == &tab)
- The address is not stored
- Address usage are replaced at compile time
- Static local string literals are similar

Mind Variables Lifetime

```
struct list {
    struct list *next;
    int         value;
};

struct list *
add(struct list *l, int x) {
    struct list *r, tmp;
    tmp.next = l;
    tmp.value = x;
    r = &tmp;
    return r;
}
```

```
int main() {
    struct list *l = NULL;
    for (int i=0; i<10; ++i)
        l = add(l, i);
    assert(l != l->next);
    return 0;
}
```

Assertion failed:
(l != l->next), function
main, file stupid_list.c,
line 23.

malloc(3)

malloc(3)

➤ C defines allocation functions: malloc(3)

The malloc() function allocates size bytes of uninitialized memory. The allocated space is suitably aligned (after possible pointer coercion) for storage of any type of object.

...

The malloc() and calloc() functions return a pointer to the allocated memory if successful; otherwise a NULL pointer is returned and errno is set to ENOMEM.

FreeBSD man page

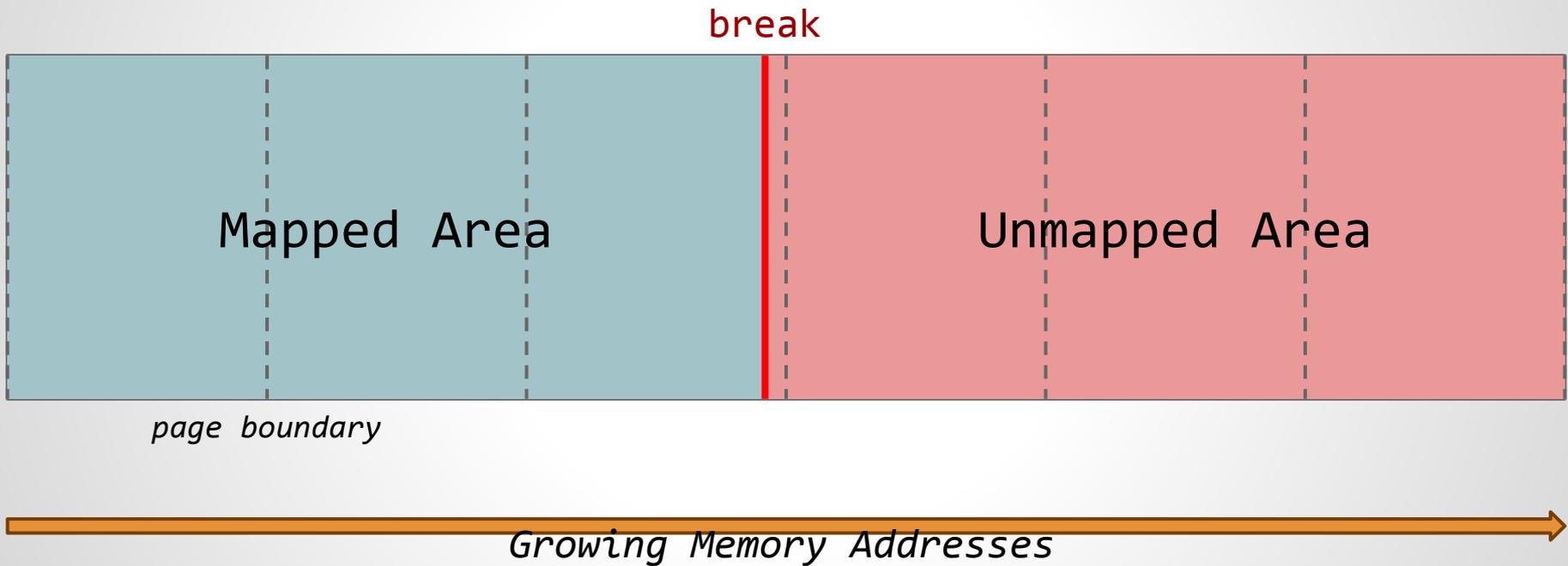
malloc(3)

- The returned pointer points to a contiguous memory zone with a size superior or equal to the asked size.
- No further invocation of malloc(3) will return a pointer to an overlapping area.
- Memory acquire with malloc(3) can be release using free(3).

Managing The Heap

- Dynamic allocation goes to the heap
- The heap start after the end of the BSS
- The actual end of the heap is called the **break**
- The break can be moved using `brk(2)` and `sbrk(2)`

The Heap



brk(2) and sbrk(2)

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

- `brk(addr)` moves the break at address `addr`
 - returns `0` if successful
 - returns `-1` and set `errno` otherwise
- `sbrk(incr)` moves the break by `incr` bytes
 - returns the address of the old break when successful
 - returns `(void*)(-1)` and set `errno` otherwise

Simplest Allocator

```
void *malloc(size_t s) {  
    void *r;  
    if ( (r = sbrk(s)) == (void*)(-1) )  
        return NULL;  
    return r;  
}
```

Comments

Pros:

- Minimal cost (just the cost of `sbrk(2)`)
- No waste space and no overhead

Cons:

- Can't implement `free(3)`
- `realloc(3)` is unsafe

What Do We Need ?

- We need information about memory chunk:
 - start of the chunk
 - end of the chunk
 - availability (freed or not)
- We need to store those data
- We need to find available chunks of a given size

We need meta-data !

Strategy

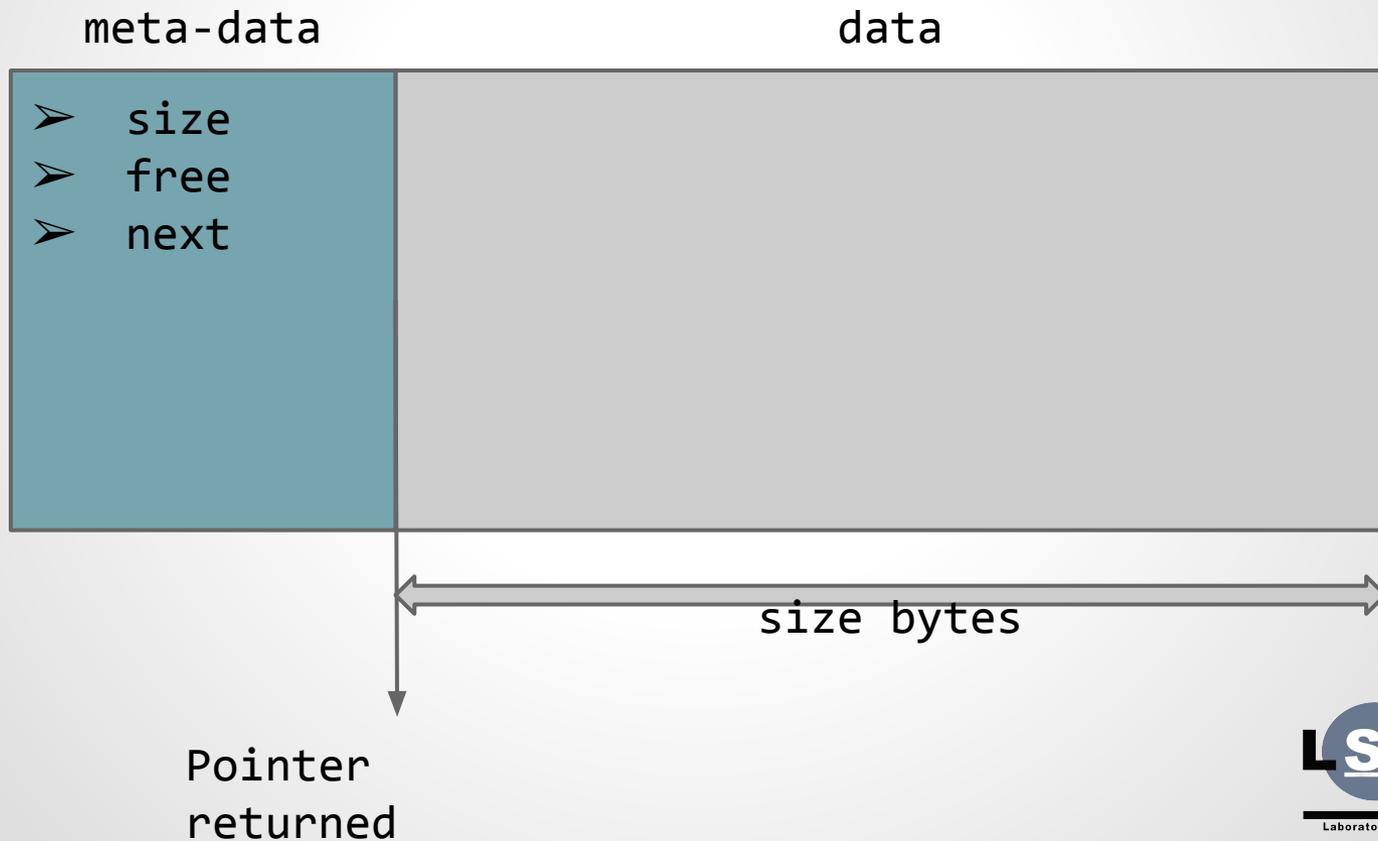
- **First-Fit:** when searching for a given size returns the first chunk sufficiently large.
- **Best-Fit:** when searching for a given size returns the smallest chunk sufficiently large.
- Simple linear organization (linked list)
- More advanced structures (buddies, slab ...)

A First Fit Allocator

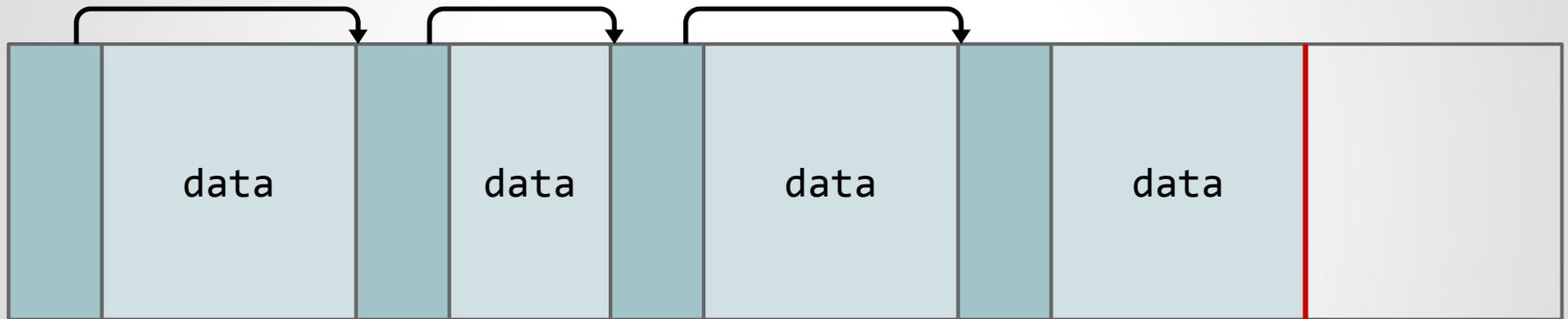
Meta-Data ?

- We add meta-data to each chunk
- we add the meta-data before the chunk
- We link chunk like a long linked list

Meta-data



Structured Heap



Meta-Data

```
struct chunk {  
    struct chunk *next;  
    size_t      size;  
    int         free;  
};
```

- next is the list pointer
- we keep data size only
- free indicate chunk availability
- we may add information later

Construction Of malloc

- Manage the first call for init
- Enforce aligned addresses: align sizes.
- When calling malloc with size s :
 - align the size (eventually force a minimal size)
 - scan the list of chunk for a free chunk sufficiently large
 - if none are available, extend the heap with `sbrk(2)`
 - mark the chunk as used and return a pointer just after the meta-data struct.
 - if it goes wrong, return NULL

Heap Init

```
void *base() {
    static struct chunk *b = NULL;
    if (!b) {
        /* Initial call */
        b = sbrk(sizeof (struct chunk));
        if (b == (void*)(-1)) {
            /* error management */
            _exit(71 /* EX_OSERR */);
        }
        /* Build a sentinel */
        b->next = NULL;
        b->size = 0;
        b->free = 0;
    }
    return b;
}
```

- Get the base of the heap
- if not initialized:
 - add place for an empty chunk
 - save the old break
 - build a sentinel
- return the saved base
- if something goes wrong, die.
- We should probably test the initial address of the break for alignment

Searching For A Free Chunk

```
struct chunk* find_chunk(size_t s)
{
    struct chunk      *cur = get_base();
    for (; cur->next; cur = cur->next) {
        if (cur->next->free && cur->next->size >= s)
            break;
    }
    return cur;
}
```

- Basic list search
- Returns the pointer to the element before the free block

Adding New Chunk

```
struct chunk *new_chunk(size_t s,
                        struct chunk *prev)
{
    struct chunk *b;
    b = sbrk(s + sizeof (struct chunk));
    if (b == (void*)(-1)) {
        if (errno == ENOMEM)
            return NULL;
        /* error management */
        _exit(71 /* EX_OSERR */);
    }
    prev->next = b;
    b->next = NULL;
    b->size = s;
    b->free = 1;
    return b;
}
```

- Used when no chunk are available
- Move the break of asked size plus meta-data size
- Old break points to the new chunk
- Init the new meta-data block
- In case of error:
 - If not enough mem: return NULL
 - Otherwise: die !

free ?

free(p)

- First we need to verify p:
 - it must be between the base of the heap and its end
 - it must be aligned on `sizeof (void*)`
- Then we must access the meta-data:
 - they lie (`sizeof (struct chunk)`) bytes before p
 - we just have to set the free field to 1

Valid call to free ?

- Is p a pointer returned by malloc ?
- We've done only the minimal tests
- Can we do better ?
 - traverse the list of chunk
 - add a magic number to the meta-data
 - available data can be verified some how ...

expensive
not sure

Tricks

- First add data address in the meta-data

```
struct chunk {  
    struct chunk *next;  
    size_t      size;  
    int        free;  
    void       *data; // pointer to data  
};
```

- Then check for the address in free

```
struct chunk *b = p;  
b -= 1;  
if (b->data == p)  
    b->free = 1;
```

More ...

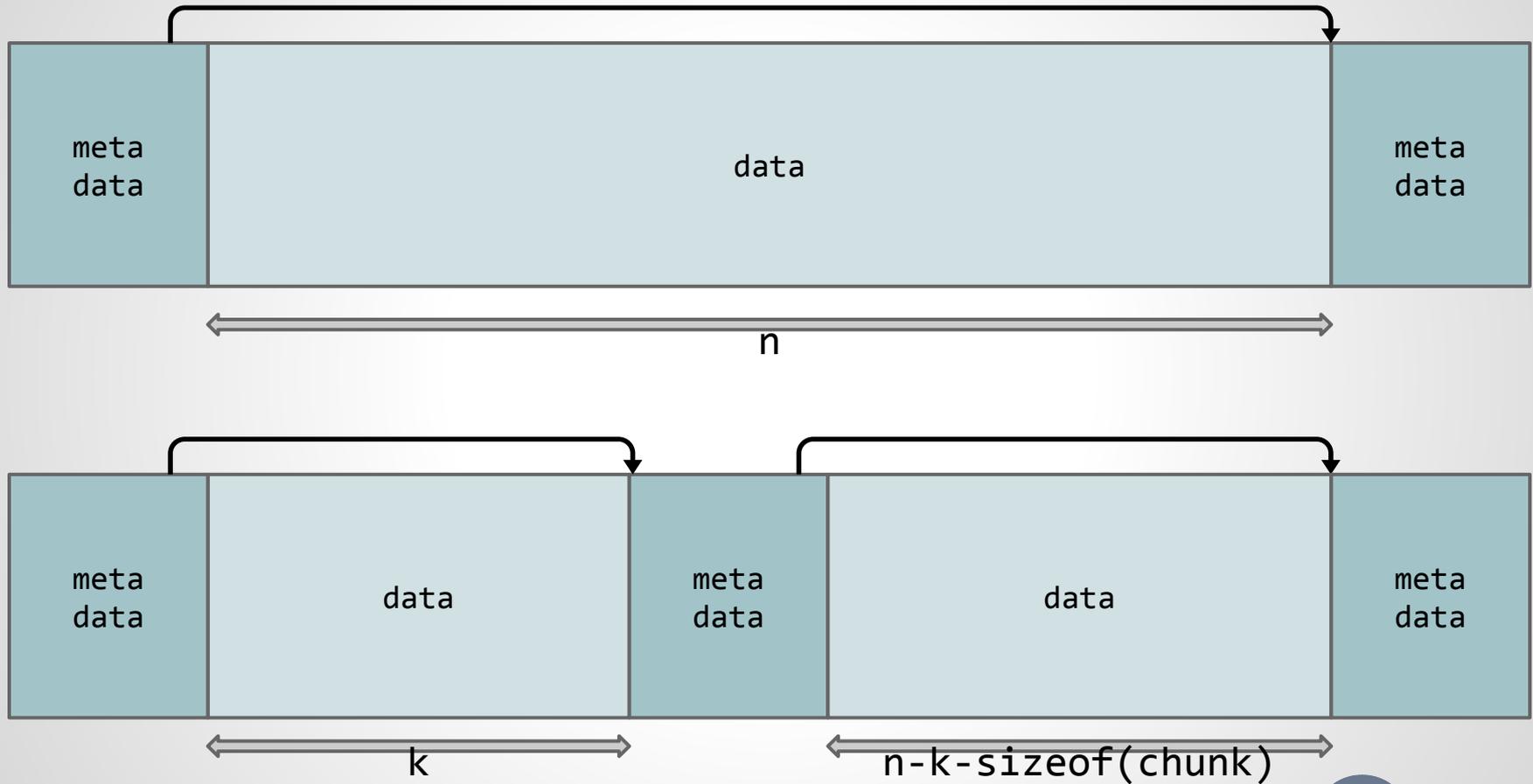
- `calloc(3)` : checks for integer overflow
- `realloc(3)` :
 - simple version: `malloc`, `copy`, `free`
 - try to reuse current block
 - Beware of special cases
- Special cases:
 - `malloc(0)` : must return a valid pointer
 - `realloc(NULL, size)` : alias for `malloc(size)`
 - `realloc(ptr, 0)` : alias for `free(ptr)`
 - when `realloc(3)` fails, it returns `NULL` and does not free

Wasted Space And Fragmentation

Wasted Space

- first-fit may choose ridiculously too large chunk
- best-fit requires clever storage for decent perf
- we should split chunk when they are too large:
 - if the founded check is larger than required
 - if the extra amount is enough to store meta-data
 - force a minimal size to avoid too small chunk
 - update the size of the chunk and populate meta-data

Split Chunk



Fragmentation

- **Internal Fragmentation:** wasted space → solved
- **External Fragmentation:** too many small chunks
 - after some round of malloc/free, all chunks are small
 - sum of free space is sufficient for large allocation
 - but no suitable chunk can be found
 - a lot of space is lost in meta-data

Merge !

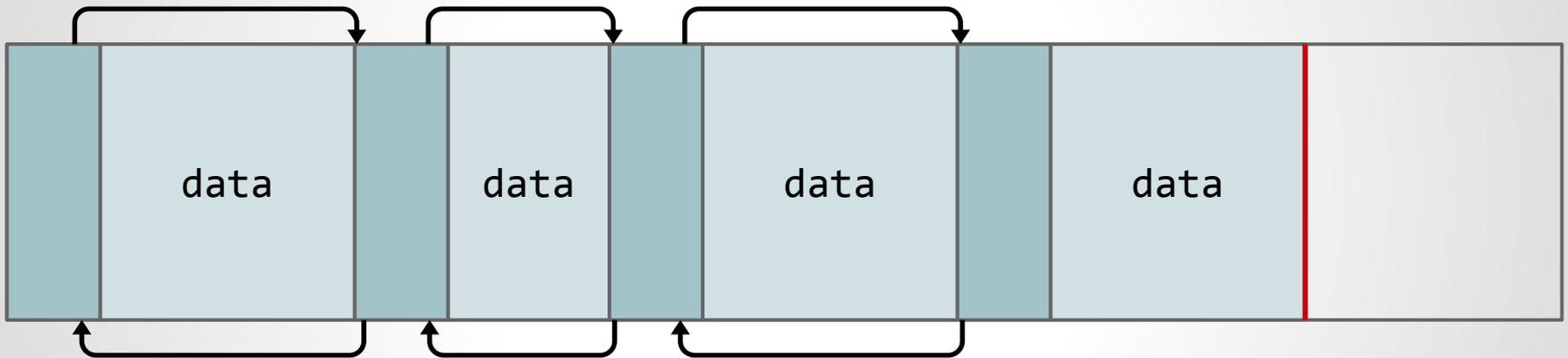
- The solution is to merge chunks
- When performing free:
 - if the next chunk is also free we can merge with it
 - we just update the size
 - and make next point to the next of the merged chunk
- Is it enough ?

Merge More ?

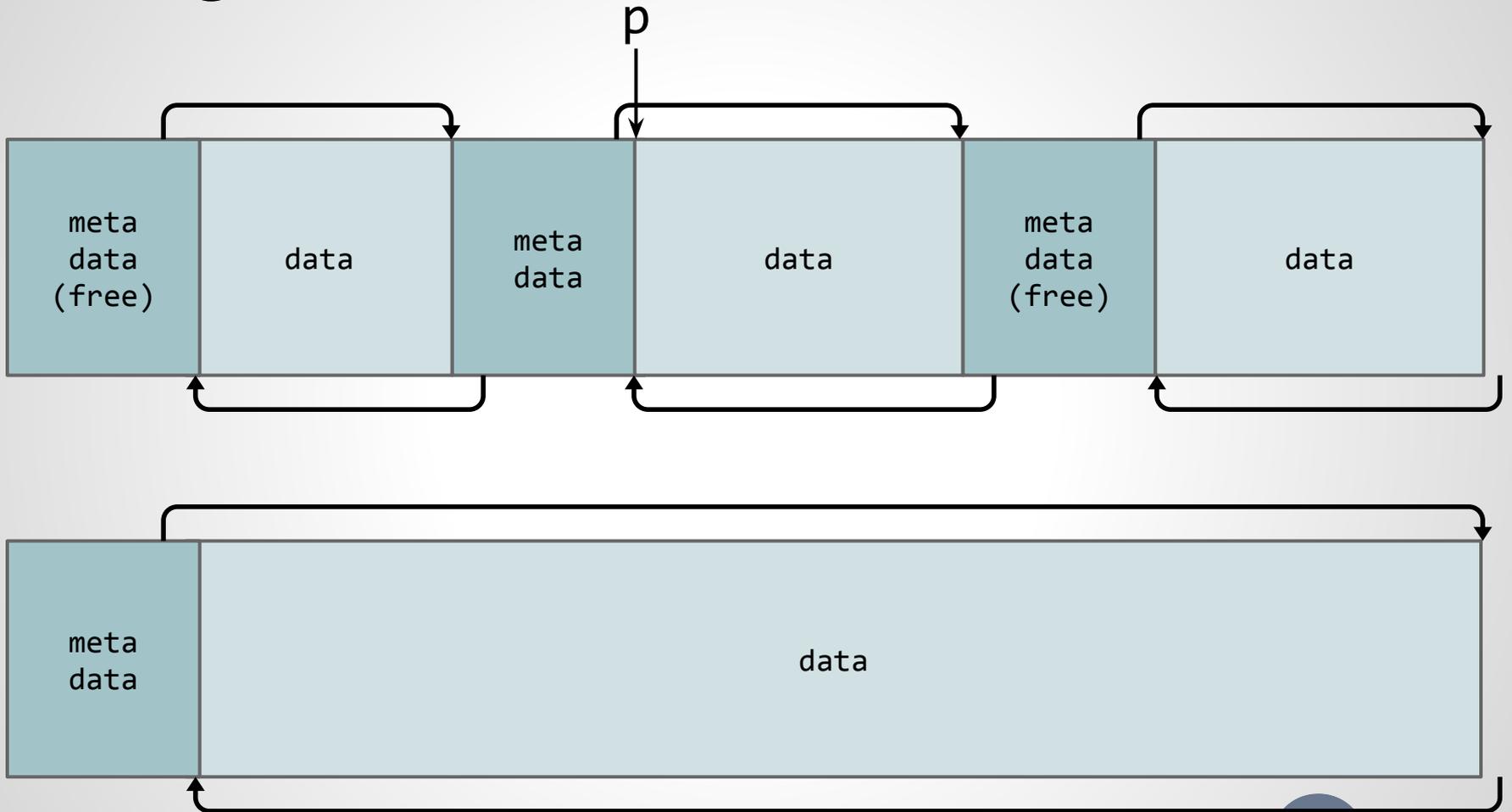
Sure that all adjacent free chunks are merged ?

- The next of the next ? No, too expensive !
- The previous chunk ? Yes
- If each time a chunk is freed we merge its previous and next chunk, we win !
- We need to access the previous chunk !

Double Link !



Merge



Doing Better ?

- Only link free chunks
- Embedded meta-data inside the chunk
- Using a best-fit ?
 - Finding the best chunk is expensive
 - Free chunk can be linked with chunks of the same size
 - Buddy Algorithm: chunks are split/merge by pair
 - Slab Allocator: pools of chunk of the same size
- Use `mmap(2)` with or instead of `sbrk(2)`

More Allocators !

More Allocators ?

`malloc(3)` is highly constrained:

- Allocate any possible size
- Free chunk independently

For specific usage we may have better strategy

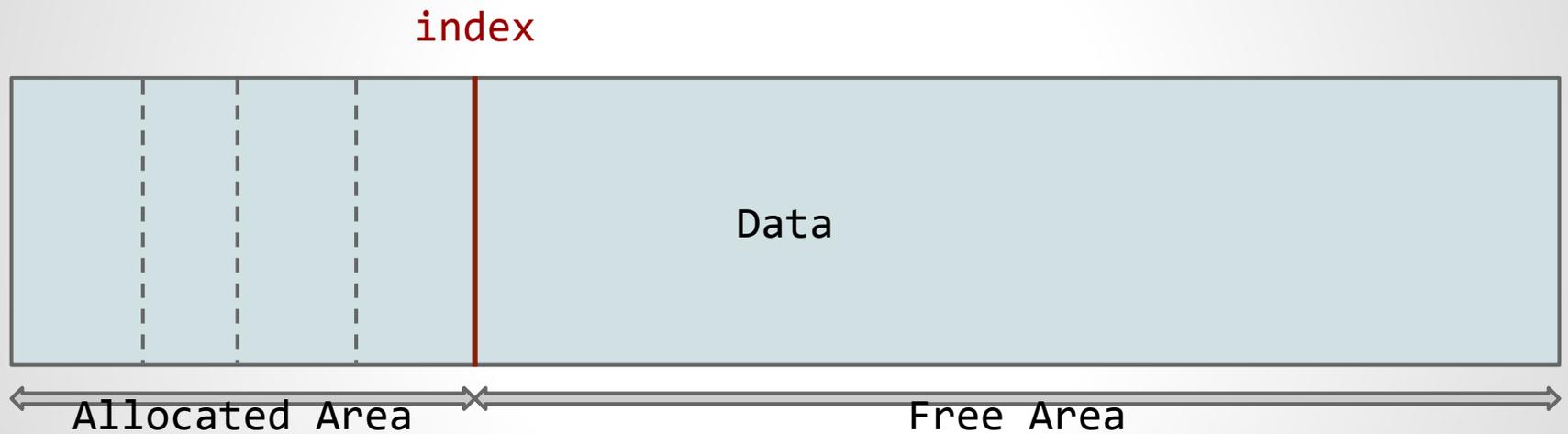
Allocate Many, Free Once

- Many programs can be sliced into stages
- Memory will be freed all at once at end of stage
- `malloc(3)` is not adapted to this case
 - we uselessly pay the cost of fine grain free
 - Freeing one by one can be difficult and expensive
- For that we can use a pool allocator !

Pool Allocator

- Allocate once and for all a huge single block
- Maintain an index of the currently used space
- Give chunk piece by piece as required
- Release memory by simply set the index to 0
- Allocation and release are in $O(1)$
- No overhead, no wasted space !

Pool Allocator



Pool Allocator

```
typedef struct pool {
    size_t      poolsize, index;
    char        *data;
}              *mempool;

void reset_pool(mempool mem) {
    mem->index = 0;
}

void *alloc_pool(mempool mem, size_t s) {
    void *r = NULL;
    if (mem->index + s < mem->poolsize) {
        r = mem->data + mem->index;
        mem->index += s;
    }
    return r;
}
```

Recycling Pool

- We have a bunch of allocations of the same size
- We often allocate/free blocks
- We can use a pool for allocation
- Rather than really freed blocks, we reuse them !
- Again we have $O(1)$ allocate and free !

Free List

- To provide easy reuse we add a free list
- We keep a list of free block
- When freeing a block:
 - we use the first word to store a pointer (our next field)
 - we bind the block to the free list (first place add)
 - we replace the entry of the free list by the block
- When allocate:
 - try to take the first element of the free list
 - take another chunk of the pool

Recycler

```
typedef struct rpool {
    size_t      poolsize, bsize, index;
    char        *data;
    void        **free_list;
}              *recycler;

void *alloc_rpool(recycler mem) {
    void        *r = NULL;
    if (mem->free_list) {
        r = mem->free_list;
        mem->free_list = *(mem->free_list);
    } else if (mem->index + mem->bsize < mem->poolsize) {
        r = mem->data + mem->index;
        mem->index += mem->bsize;
    }
    return r;
}

void free_rpool(recycler mem, void *p_) {
    void        **p = p_;
    *p = mem->free_list;
    mem->free_list = p;
}
```