

Operating Systems

Memory Allocators

Giorgio Grisetti

`grisetti@diag.uniroma1.it`

Department of Computer Control and Management Engineering
Sapienza University of Rome

In this episode

We will see two types of allocators that can be used to manage a buffer of memory

They are extensively used within the kernel of the operating system, to manage the small objects required to implement data structures

Issues in managing memory

- fragmentation:

when we would have enough memory to satisfy a request, but the scattered allocations in the buffer make it impossible to find a contiguous chunk that is large enough

- time:

how long do i need to wait to get/release a memory block?

To manage memory we have to waste a little (for storing structures describing the memory layout)

SLAB allocator

Use it when

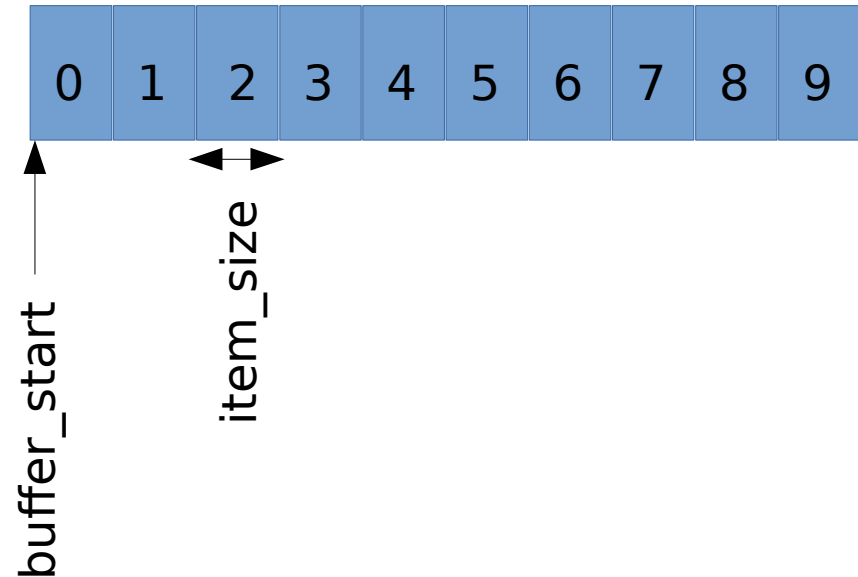
- you have a bunch of objects of fixed size (`item_size`)
- you know how many of these objects you can have (in the worst case)

A slab allocator divides the a memory in chunks of size `item_size`.

- If the memory starts at address `buffer_start`, the address of the `idx` block is
$$\text{ptr_block} = \text{buffer_start} + \text{idx} * \text{item_size}$$
- If i know an address how to get the index? (1st order equation: resolve the above by `idx`)

$$\text{idx} = (\text{ptr_block} - \text{buffer_start}) / \text{item_size}$$

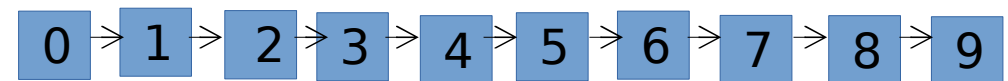
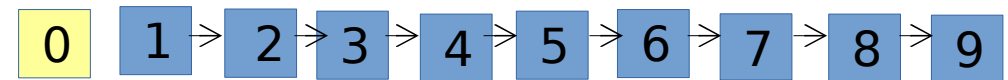
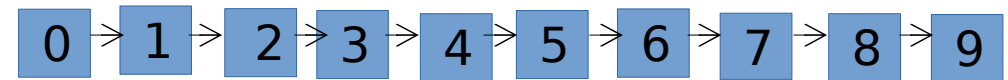
Not all addresses are good, only those aligned with the item size boundaries



SLAB aux structures

We can have a SLAB allocator capable of satisfying the requests in $O(1)$ by just keeping a list of structures representing free blocks

- At the beginning the list is populated with all blocks
- When a request comes, we return the block at the beginning of the list, and we remove it from the list
- When a block is deleted, we create the corresponding item and we put it back in the list.

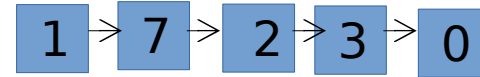


Issues:

- To implement a SLAB we need a list.
- To implement a list we need some sort of malloc
- how do we do without malloc?

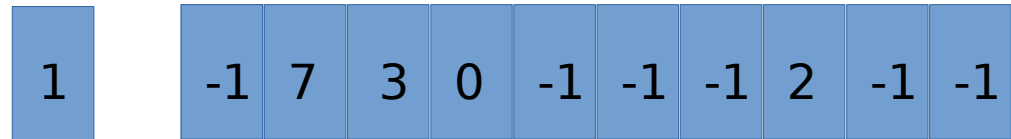
Array Lists

If we know the maximum size of a list, we can map a list on an array of `max_size` elements.



We need

- a start position
`int start_pos;`
- an array of int
`int array_list[max_size];`
- The cell at position `i` stores
 - the index of the successor element
 - -1 if not in the list, or null



SLAB interface

To manage a memory with a slab we need

- $N_{\max} \times \text{item_size}$ elements
- an array list of size N_{\max}

```
typedef enum {
    Success=0x0,
    NotEnoughMemory=-1,
    UnalignedFree=-2,
    OutOfRange=-3,
    DoubleFree=-4
} PoolAllocatorResult;

typedef struct PoolAllocator{

    //contiguous buffer managed by the system
    char* buffer;
    //list of linked objects
    int* free_list;
    //size of the buffer in bytes
    int buffer_size;

    //number of free blocks
    int size;
    //maximum number of blocks
    int size_max;
    //size of a block
    int item_size;
    //pointer to the first bucket
    int first_idx;
    // size of a bucket
    int bucket_size;
} PoolAllocator;
```

SLAB interface

To initialize a SLAB, we need to provide an external memory buffer

```
PoolAllocatorResult  
PoolAllocator_init(PoolAllocator* allocator,  
                  int item_size,  
                  int num_items,  
                  char* memory_block,  
                  int memory_size);
```

```
void* PoolAllocator_getBlock(  
    PoolAllocator* allocator);
```

```
PoolAllocatorResult  
PoolAllocator_releaseBlock(  
    PoolAllocator* allocator,  
    void* block);
```

```
// helper function that returns a string  
// from an error message  
const char* PoolAllocator_strerror  
    (PoolAllocatorResult result);
```

The buffer should have enough room to hold the enough elements and the array list.

Once initialized we can request a block to the slab, or return an already allocated block.

SLAB testing

```
// object size=4K
# define item_size 4096

// 16 blocks
#define num_items 16

// buffer should contain also bookkeeping information
#define buffer_size num_items*(item_size+sizeof(int))

// we allocate buffer in .bss
char buffer[buffer_size];

PoolAllocator allocator;

int main(int argc, char** argv) {
    printf("initializing... ");
    PoolAllocatorResult init_result=PoolAllocator_init(&allocator,
                                                    item_size,
                                                    num_items,
                                                    buffer,
                                                    buffer_size);
    printf("%s\n", PoolAllocator_strerror(init_result));

    // we allocate_all memory, and a bit more

    void* blocks[num_items+10];
    for (int i=0; i<num_items+10; ++i){
        void* block=PoolAllocator_getBlock(&allocator);
        blocks[i]=block;
        printf("allocation %d, block %p, size%d\n", i, block, allocator.size);
    }
}
```


SLAB testing

```
// we release all memory
for (int i=0; i<num_items+10; ++i){
    void* block=blocks[i];
    if (block){
        printf("releasing... idx: %d, block %p, free %d ... ",
            i, block, allocator.size);
        PoolAllocatorResult release_result=PoolAllocator_releaseBlock(&allocator, block);
        printf("%s\n", PoolAllocator_strerror(release_result));
    }
}

// we release all memory again (should get a bunch of errors)
for (int i=0; i<num_items+10; ++i){
    void* block=blocks[i];
    if (block){
        printf("releasing... idx: %d, block %p, free %d ... ",
            i, block, allocator.size);
        PoolAllocatorResult release_result=PoolAllocator_releaseBlock(&allocator, block);
        printf("%s\n", PoolAllocator_strerror(release_result));
    }
}

// we allocate half of the memory, and release it in reverse order
for (int i=0; i<num_items-5; ++i){
    void* block=PoolAllocator_getBlock(&allocator);
    blocks[i]=block;
    printf("allocation %d, block %p, size%d\n", i, block, allocator.size);
}
```

SLAB testing

```
for (int i=num_items-1; i>=0; --i){
void* block=blocks[i];
if (block){
    printf("releasing... idx: %d, block %p, free %d ... ",
        i, block, allocator.size);
    PoolAllocatorResult release_result=PoolAllocator_releaseBlock(&allocator, block);
    printf("%s\n", PoolAllocator_strerror(release_result));
}
}

// we allocate all memory,
// and release only even blocks, in reverse order
// release odd blocks in reverse order
for (int i=0; i<num_items; ++i){
    void* block=PoolAllocator_getBlock(&allocator);
    blocks[i]=block;
    printf("allocation %d, block %p, size%d\n", i, block, allocator.size);
}

for (int i=num_items-1; i>=0; i-=2){
    void* block=blocks[i];
    if (block){
        printf("releasing... idx: %d, block %p, free %d ... ",
            i, block, allocator.size);
        PoolAllocatorResult release_result=PoolAllocator_releaseBlock(&allocator, block);
        printf("%s\n", PoolAllocator_strerror(release_result));
    }
}
```

SLAB testing

```
for (int i=num_items-1; i>=0; --i){
void* block=blocks[i];
if (block){
    printf("releasing... idx: %d, block %p, free %d ... ",
        i, block, allocator.size);
    PoolAllocatorResult release_result=PoolAllocator_releaseBlock(&allocator, block);
    printf("%s\n", PoolAllocator_strerror(release_result));
}
}

// we allocate all memory, and release only even blocks, in reverse order
// release odd blocks in reverse order
for (int i=0; i<num_items; ++i){
    void* block=PoolAllocator_getBlock(&allocator);
    blocks[i]=block;
    printf("allocation %d, block %p, size%d\n", i, block, allocator.size);
}
for (int i=num_items-1; i>=0; i-=2){
    void* block=blocks[i];
    if (block){
        printf("releasing... idx: %d, block %p, free %d ... ",
            i, block, allocator.size);
        PoolAllocatorResult release_result=PoolAllocator_releaseBlock(&allocator, block);
        printf("%s\n", PoolAllocator_strerror(release_result));
    }
}
for (int i=num_items-2; i>=0; i-=2){
    void* block=blocks[i];
    if (block){
        printf("releasing... idx: %d, block %p, free %d ... ",
            i, block, allocator.size);
        PoolAllocatorResult release_result=PoolAllocator_releaseBlock(&allocator, block);
        printf("%s\n", PoolAllocator_strerror(release_result));
    }
}
}
```

Take home message

BE EVIL WHEN TESTING

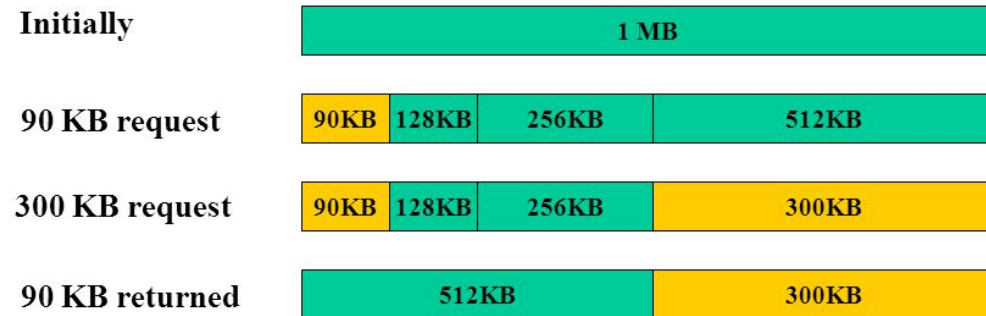
Buddy Allocator

When we have objects of arbitrary size

We recursively partition the memory in two, a maximum number of times.

A "buddy" of a memory block is the other region that is obtained by partitioning the "parent" region

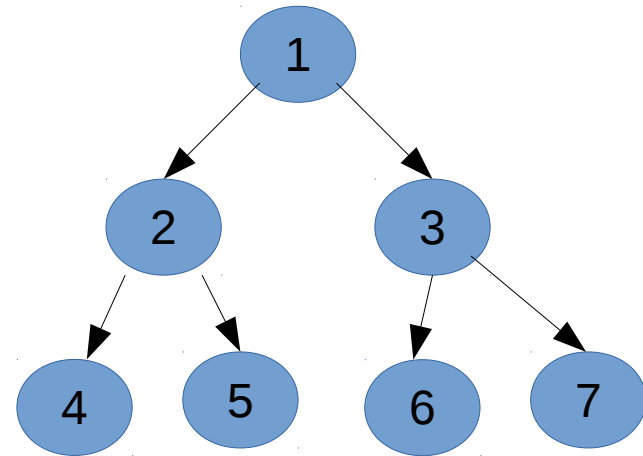
Allocating a block that is smaller than the smaller partition, wastes memory



Funny Binary Trees

Enumerating the nodes of a binary tree has some funny properties:

- level of node i
 - $\text{level}(\text{idx}) = \text{floor}(\log_2(\text{idx}))$
- idx of 1st node of a level i
 $\text{firstIdx}(i) = 1 \ll i \ // \{2^i\}$
- offset of node idx in his level
 $\text{idx} - \text{firstIdx}(\text{level}(\text{idx}))$
- index of the buddy of node i
 $\text{buddyIdx}(\text{idx}) = (i \% 2) ? \text{idx} - 1 : \text{idx} + 1;$
- parent of the node idx
 $\text{parentIdx}(\text{idx}) = \text{floor}(\text{idx} / 2);$
- each possible allocation in our system can be given an index in a binary tree
- from this index we can recover useful information, without storing anything else



```
int levelIdx(size_t idx){
    return (int)floor(log2(idx));
};

int buddyIdx(int idx){
    if (idx&0x1){
        return idx-1;
    }
    return idx+1;
}

int parentIdx(int idx){
    return idx/2;
}

int startIdx(int idx){
    return (idx-(1<<levelIdx(idx)));
}
```

Buddy Allocator

A block of memory is associated with a BuddyListItem, that stores also the tree structure.

In the allocator, we keep free lists for each level of the buddy system.

We use a SLAB allocator to manage the lists

What is the maximum capacity of the SLAB?

```
typedef struct BuddyListItem {
    ListItem list;
    int idx;    // tree index
    int level; // level for the buddy
    char* start; // start of memory
    int size;
    struct BuddyListItem* buddy_ptr;
    struct BuddyListItem* parent_ptr;
} BuddyListItem;
```

```
typedef struct {
    ListHead free[MAX_LEVELS];
    ListHead occupied[MAX_LEVELS];
    int num_levels;
    PoolAllocator list_allocator;
    // the memory area to be managed
    char* memory; //
    // the minimum page of RAM that can be returned
    int min_bucket_size;
} BuddyAllocator;
```

Buddy Allocator

We store each level in a list of "free blocks"

When memory is requested:

- the size+8 is rounded up to the size of the smallest partition capable to contain it
- if the partition is in the free list of that level, we return the partition and we remove from the free list
- if a partition at that level is not available, we ask for a partition to the higher level and we split it in two (recursively) the region returned, populating the free list accordingly

```
//allocates memory
void* BuddyAllocator_malloc(BuddyAllocator* alloc,
                           int size) {
    // calculate max mem
    int mem_size=
        (1<<alloc->num_levels)*alloc->min_bucket_size;

    //calculate level for page
    int level=floor(log2(mem_size/(size+8)));

    // if the level is too small, we pad it to max
    if (level>alloc->num_levels)
        level=alloc->num_levels;

    printf("requested: %d bytes, level %d \n",
           size, level);

    // we get a buddy of that size;
    BuddyListItem* buddy=
        BuddyAllocator_getBuddy(alloc, level);
    if (! buddy)
        return 0;

    // we write in the memory
    // region managed the buddy address
    BuddyListItem** target=
        (BuddyListItem**)(buddy->start);
    *target=buddy;
    return buddy->start+8;
}
```

note that we return the start address+8

Buddy Allocator

We store each level in a list of "free blocks"

When memory is requested:

- the size is rounded up to the size of the smallest partition capable to contain it
- if the partition is in the free list of that level, we return the partition and we remove from the free list
- if a partition at that level is not available, we ask for a partition to the higher level and we split it in two (recursively) the region returned, populating the free list accordingly

```
BuddyListItem* BuddyAllocator_getBuddy(
    BuddyAllocator* alloc, int level){
    if (level<0)
        return 0;
    if (! alloc->free[level].size ) {
        // no buddies on this level
        BuddyListItem* parent_ptr=
            BuddyAllocator_getBuddy(alloc, level-1);
        if (! parent_ptr)
            return 0;

        // parent already detached from free list
        int left_idx=parent_ptr->idx<<1;
        int right_idx=left_idx+1;

        BuddyListItem* left_ptr=
            BuddyAllocator_createListItem(alloc,
                                           left_idx,
                                           parent_ptr);
        BuddyListItem* right_ptr=
            BuddyAllocator_createListItem(alloc,
                                           right_idx,
                                           parent_ptr);
        // we need to update the buddy ptrs
        left_ptr->buddy_ptr=right_ptr;
        right_ptr->buddy_ptr=left_ptr;
    }
    // we detach the first
    if(alloc->free[level].size) {
        BuddyListItem* item=
            (BuddyListItem*)
            List_popFront(alloc->free+level);
        return item;
    }
    return 0;
}
```

Buddy Allocator

When memory is released we

- identify the "region" that was associated to the released block (see how in the next slide)
- if the buddy of the region is **not** in the free list of the level, we add the element to the list and we terminate
- if the buddy is in the free list, we
 - merge the two buddies, by deleting them from the free list
 - we insert a new entry in the upper level, corresponding to the merged elements (recursively)

This mechanism is capable to handle a certain level of fragmentation

The max operations are $O(\text{levels})$

```
void BuddyAllocator_releaseBuddy(
    BuddyAllocator* alloc,
    BuddyListItem* item){

    BuddyListItem* parent_ptr=item->parent_ptr;
    BuddyListItem *buddy_ptr=item->buddy_ptr;

    // buddy back in the free list of its level
    List_pushFront(&alloc->free[item->level],
        (ListItem*)item);

    // if on top of the chain, do nothing
    if (! parent_ptr)
        return;

    // if the buddy of this item is not free,
    // we do nothing
    if (buddy_ptr->list.prev==0 &&
        buddy_ptr->list.next==0)
        return;

    //join
    //1. we destroy the two buddies in the free list;
    printf("merge %d\n", item->level);
    BuddyAllocator_destroyListItem(alloc, item);
    BuddyAllocator_destroyListItem(alloc, buddy_ptr);
    //2. we release the parent
    BuddyAllocator_releaseBuddy(alloc, parent_ptr);
}
```

Buddy Allocator

How to determine the buddy of a memory area when it is freed?

- Option 1:

We seek for a list item whose "start" field address matches with the returned region (slow)

- Option 2:

- We "store" the address of the list item (that is in a "detached" status, in the first bytes of the memory partition.

- the returned memory address will be

beginning_of_region+address_size

- We can retrieve the address of the list element by decrementing the address of <address_size> bytes

```
//releases allocated memory
void BuddyAllocator_free(BuddyAllocator* alloc,
                        void* mem) {
    printf("freeing %p", mem);
    // we retrieve the buddy from the system
    char* p=(char*) mem;
    p=p-8;
    BuddyListItem** buddy_ptr=(BuddyListItem**)p;
    BuddyListItem* buddy=*buddy_ptr;
    //printf("level %d", buddy->level);
    // sanity check;
    assert(buddy->start==p);
    BuddyAllocator_releaseBuddy(alloc, buddy);
}
```

Buddy Allocator Interface

- The interface is similar to the pool allocator

```
// initializes the buddy allocator,  
// and checks that the buffer is large enough  
void BuddyAllocator_init(BuddyAllocator* alloc,  
                        int num_levels,  
                        char* buffer,  
                        int buffer_size,  
                        char* memory,  
                        int min_bucket_size);
```

- need to pass two buffers:

```
//allocates memory  
void* BuddyAllocator_malloc(BuddyAllocator* alloc,  
                           int size);
```

- one for the internal pool allocator that stores the list

```
//releases allocated memory  
void BuddyAllocator_free(BuddyAllocator* alloc,  
                        void* mem);
```

- one for the managed memory
 - the smallest leaf of the buddy
- no need to specify the block size

Buddy Testing

```
#include "buddy_allocator.h"
#include <stdio.h>

#define BUFFER_SIZE 102400
#define BUDDY_LEVELS 9
#define MEMORY_SIZE (1024*1024)
#define MIN_BUCKET_SIZE (MEMORY_SIZE>>(BUDDY_LEVELS))

char buffer[BUFFER_SIZE]; // 100 Kb buffer to handle memory should be enough
char memory[MEMORY_SIZE];

BuddyAllocator alloc;
int main(int argc, char** argv) {

    //1 we see if we have enough memory for the buffers
    int req_size=BuddyAllocator_calcSize(BUDDY_LEVELS);
    printf("size requested for initialization: %d/BUFFER_SIZE\n", req_size);

    //2 we initialize the allocator
    printf("init... ");
    BuddyAllocator_init(&alloc, BUDDY_LEVELS,
                        buffer,
                        BUFFER_SIZE,
                        memory,
                        MIN_BUCKET_SIZE);

    printf("DONE\n");

    void* p1=BuddyAllocator_malloc(&alloc, 100);
    void* p2=BuddyAllocator_malloc(&alloc, 100);
    void* p3=BuddyAllocator_malloc(&alloc, 100000);
    BuddyAllocator_free(&alloc, p1);
    BuddyAllocator_free(&alloc, p2);
    BuddyAllocator_free(&alloc, p3);
}
```

Buddy Issues

Issues

- Lots of space wasted to store the tree
- Recursion is good for compilers, should be avoided inside an OS
 - in this case it can easily be avoided

Solutions

- Tree is stored in a bitmap
- No free lists, the items are found through brutal bitwise checks
- Modern machines do this 64 bytes at a time
- Asymptotically, worse, better in practice (cache issues etc)

Exercises

SLAB

- Implement the exercise on the polymorphic list using a SLAB allocator instead malloc/free

Buddy

- modify the init function of the buddy allocator so that it takes
 - a memory buffer
 - the size of the memory area to manage
 - the number of levels
- The internal buffer for the SLAB allocator should be "taken" from the single buffer passed, and allocated at the beginning of it