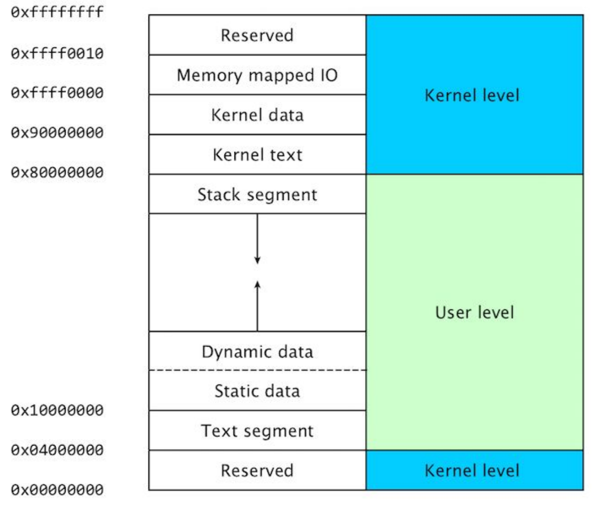
My Malloc: Mylloc and Mhysa

Implement your own malloc()

malloc(size)

- Allocate memory dynamically on the heap (std lib)
- Returns a pointer to the start of the memory area
- May return NULL
- Has to be deallocated by calling free()
- Calls something called sbrk() internally



sbrk(size)

- The almost raw way of asking for memory
- Increments the program break by the defined size in bytes
- Returns the previous address of the program break
- sbrk(0) – returns current address of the program break

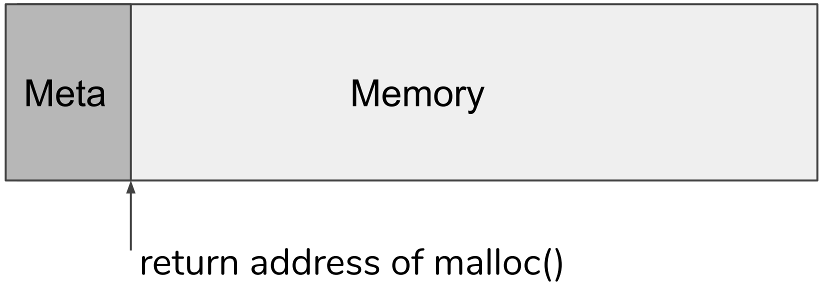
brk(address)

- The raw way of asking for memory
- Sets the program break to the specified address
- Returns 0 on success

free(pointer)

Going back to malloc...

- Deallocates the memory that malloc() pointed to
- Meta data is kept in the memory header to allow it to be freed (and maybe other stuff)
- Also calls sbrk() internally



Allocation Strategies

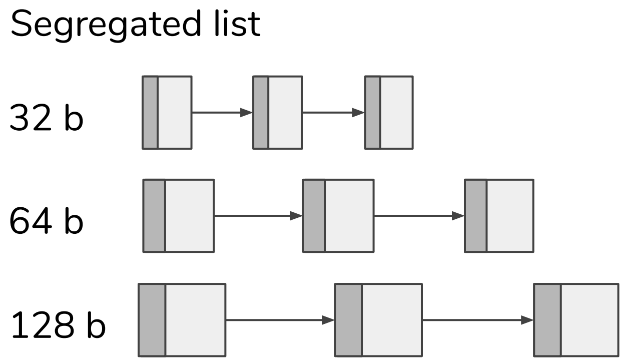
The problem of how to get and return memory: How to split and coalesce

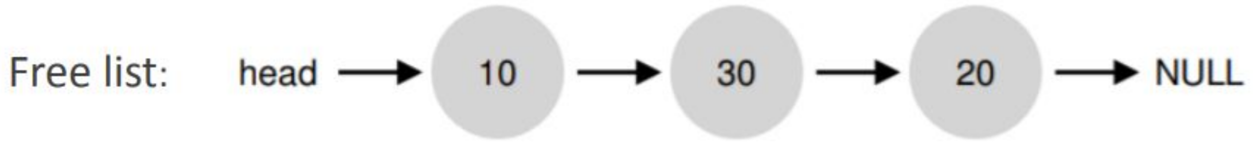
- List based allocation
  - Best fit
    - slow
  - Worst fit
    - low external fragmentation, high internal fragmentation
  - First fit / next fit
    - quick but high external fragmentation

- Buddy allocation

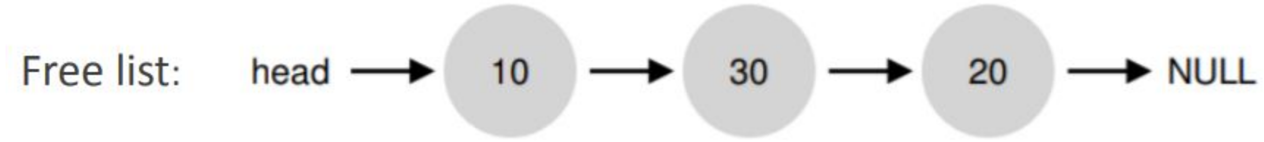
List based approach

- Contains blocks of the available free space on the heap
- Naïve: one list for all blocks
  - Different selection strategies exists
- One list for each size: **segregated list**
  - Similar structure used in malloc





Selection strategies for lists

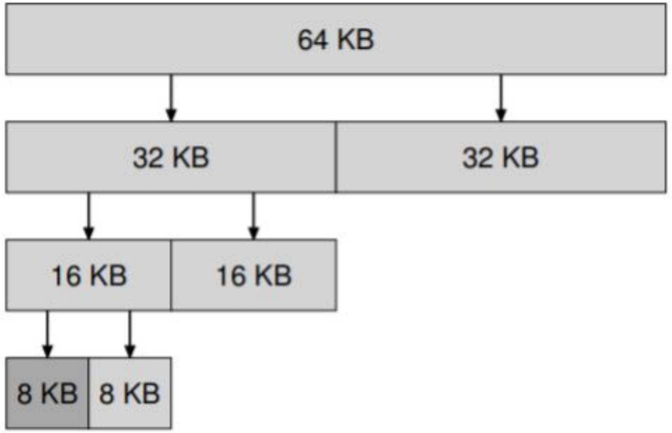


Which block will each strategy choose for a request of size 15?

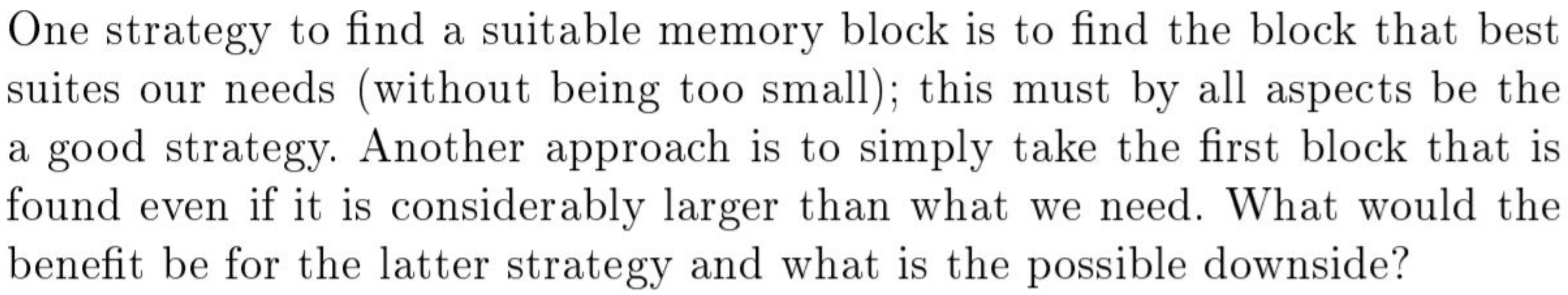
- Best fit: 20
- Worst fit: 30
- First/Next fit: 30

Buddy allocation

- Similar structure used on OS level
- Easy to coalesce
- Memory is recursively splitted into two to appropriate size
- Cons – Internal fragmentation with only size 2^n



2017-08-21



One strategy to find a suitable memory block is to find the block that best suites our needs (without being too small); this must by all aspects be the a good strategy. Another approach is to simply take the first block that is found even if it is considerably larger than what we need. What would the benefit be for the latter strategy and what is the possible downside?

2017-08-21

One strategy to find a suitable memory block is to find the block that best suites our needs (without being too small); this must by all aspects be the a good strategy. Another approach is to simply take the first block that is found even if it is considerably larger than what we need. What would the benefit be for the latter strategy and what is the possible downside?

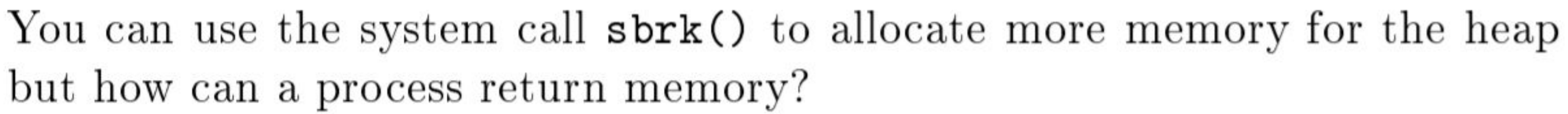
Answers:
One of the obvious upsides of this approach would be that we don't have to search through "**all**" of the blocks. Decreasing the time it takes to find a free block.

If we were to split the selected block we also get less external fragmentation than best fit.
However if we don't split the selected block we end up with a lot of internal fragmentation.

2017-08-21

You can use the system call `sbrk()` to allocate more memory for the heap but how can a process return memory?

2017-08-21



You can use the system call **`sbrk()`** to allocate more memory for the heap but how can a process return memory?

Answers:
By explicitly setting the top of the heap using the **brk()** or by giving **sbrk()** a negative value

2017-12-18

Assume that we implement a memory manager (alloc/free) where the free list is handled using the construct below. Which advantages and possible disadvantages would this give us?

```
__thread  chunk *free = NULL;

void free(void *memory) {
  if(memory != NULL) {
    struct chunk *cnk = (struct chunk*)((struct chunk*)memory - 1);
    cnk->next = free;
    free = cnk;
  }
  return;
}
```

2017-12-18

```
__thread  chunk *free = NULL;

void free(void *memory) {
  if(memory != NULL) {
    struct chunk *cnk = (struct chunk*)((struct chunk*)memory - 1);
    cnk->next = free;
    free = cnk;
  }
  return;
}
```

Answers:
Every thread has its own list of free blocks, meaning that it’s safe operate on the list of free blocks without semaphores and locks. This maximises the chances of good cache performance on the individual thread.
However this creates the risk of creating an imbalance between the amount of free blocks on the threads. Hence we could end up with a thread that’s out of free space, and the free space on the other threads can’t be used.

1.4 the size of the block? [2 points]

When implementing *free()* we need to know the size of the block that should be freed. How do we know the size? Draw and explain in the figure below what an implementation could look like.

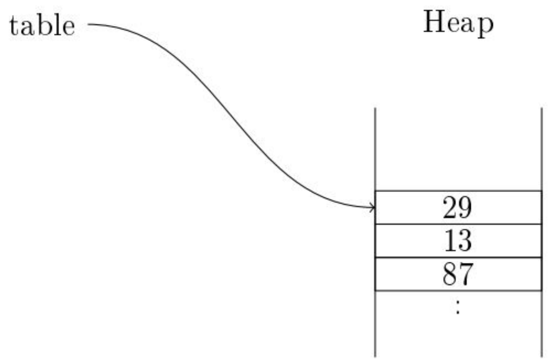
1.4 the size of the block? [2 points]

When implementing *free()* we need to know the size of the block that should be freed. How do we know the size? Draw and explain in the figure below what an implementation could look like.

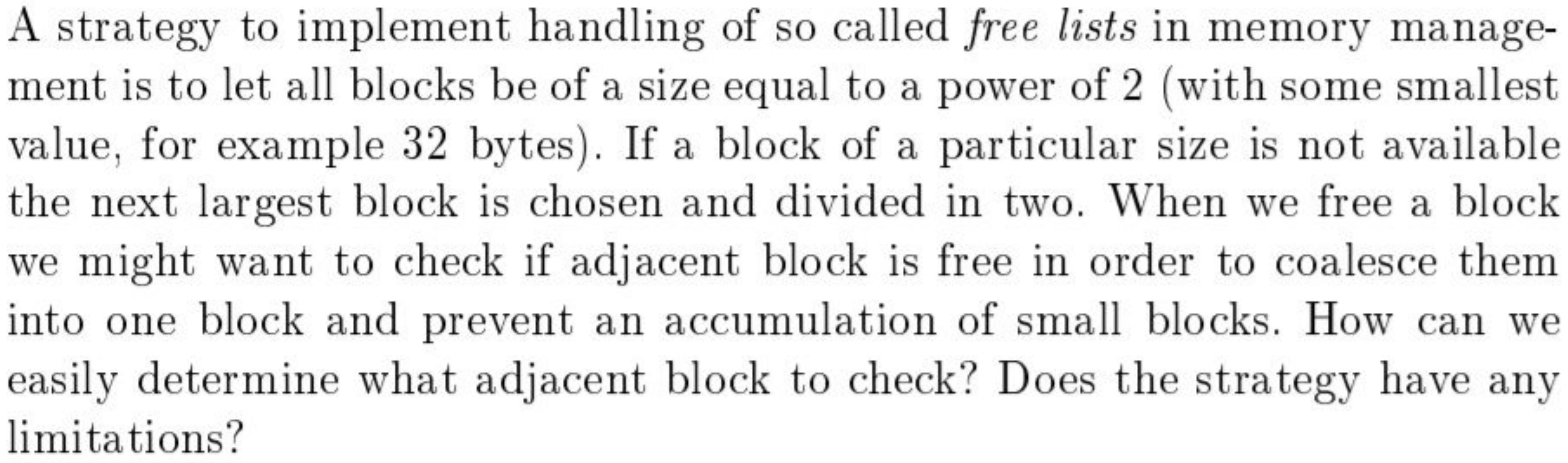
Answer: You can hide a header before the allocated block where the size of the block is recorded.

```
int *new_table(int elements) {
  return (int*)malloc(sizeof(int)*elements);
}

int main() {}
  int *table = new_table(24);
   :
   table[0] = 29;
   table[1] = 13;
   table[2] = 87;
   :
  free(table);
  return 0;
}
```

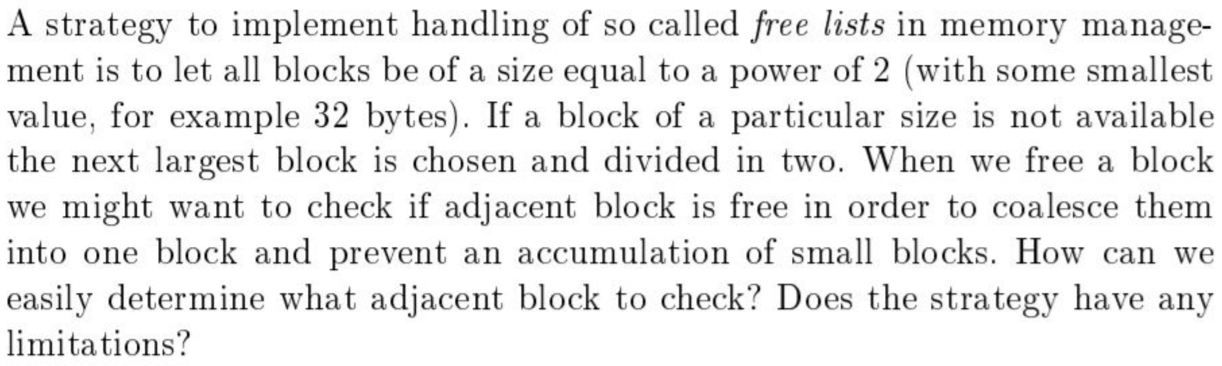


2017-06-07



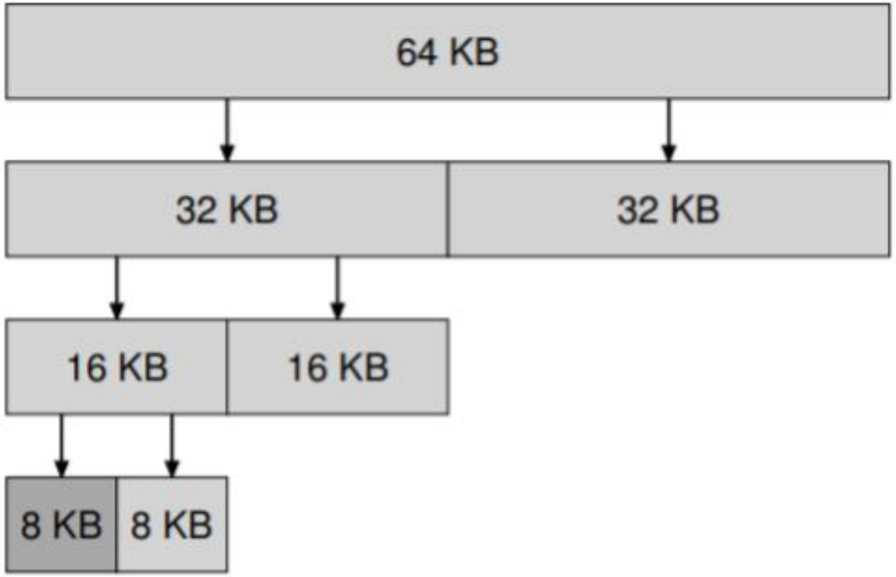
A strategy to implement handling of so called *free lists* in memory management is to let all blocks be of a size equal to a power of 2 (with some smallest value, for example 32 bytes). If a block of a particular size is not available the next largest block is chosen and divided in two. When we free a block we might want to check if adjacent block is free in order to coalesce them into one block and prevent an accumulation of small blocks. How can we easily determine what adjacent block to check? Does the strategy have any limitations?

2017-06-07



A strategy to implement handling of so called *free lists* in memory management is to let all blocks be of a size equal to a power of 2 (with some smallest value, for example 32 bytes). If a block of a particular size is not available the next largest block is chosen and divided in two. When we free a block we might want to check if adjacent block is free in order to coalesce them into one block and prevent an accumulation of small blocks. How can we easily determine what adjacent block to check? Does the strategy have any limitations?

Answers:
We can use the buddy-algorithm strategy to simply toggle a bit to get the address of the "*Buddy*", which is a VERY quick operation.
A limitation of this algorithm however is that we can't merge two blocks unless they're buddies, even though they're side by side.
Also we can only allocate these fixed sizes leading to internal fragmentation


64 KB
32 KB
32 KB
16 KB
16 KB
8 KB
8 KB

4.2 implement the buddy algorithm [2 points]

Assume that you should implment the buddy-algorithm for memory management. To help you a function that locates the buddy of a block of size k is given. Assume that all blocks are taged as either free ortaken and have a field that gives the size of the block. Free blocks also have two pointers that links the block in a double linked list of free blocks of its size.

Assume that you should free a block and have found its buddy - what do you have to check before you can coalece the block with it's buddy? If the blocks can me coaleced, what are the operations that you need to perform.

Assume that you should implment the buddy-algorithm for memory management. To help you a function that locates the buddy of a block of size k is given. Assume that all blocks are taged as either free ortaken and have a field that gives the size of the block. Free blocks also have two pointers that links the block in a double linked list of free blocks of its size.

Assume that you should free a block and have found its buddy - what do you have to check before you can coalece the block with it's buddy? If the blocks can me coaleced, what are the operations that you need to perform.

Answer: You have to check if the buddy is **free** and of the **same size**. If this is the case they can be coalesced. This is done by **removing the buddy** from its free-list, determine which block is the mayor one, change the size of this block. One then must free the coalesced block **recursively**. When no free buddy is found the block is inserted first in the list of its size.