My Malloc: Mylloc and Mhysa

Implement your own malloc()

Katariina Martikainen, Charlotta Spik, Fredrik Carlsson
With credit to Isabel Ghourchian who’s slide text from last year was used for this presentation
2018-11-14
KTH Kista, Stockholm

malloc(size)

- Used to allocate memory dynamically (at runtime) on the heap
- Needed when you do not know how much memory will be used until the program runs
- Return value:
  - a pointer (variable that holds an address) to the beginning of the allocated memory
  - or NULL if the memory can’t be allocated

malloc() - example

- Declaring an array with n number of elements
    - You do not know the number of elements from the beginning
    - It is given as an input from the user
    - The input n is taken at runtime and that much memory is allocated using malloc

sbrk(size)

- Ask for more memory on the heap by using the program break
- The program break points to the end of the heap
- Increments the program break by the defined size in bytes
- Return value:
  - the previous program break
  - or -1 on error
- sbrk(0) – returns current location of the program break

brk(address)

- Similar to sbrk() but takes an address instead
- Sets the program break to the specified address
- Return value:
    - 0 on success
    - or -1 on error

free(pointer)

- Deallocates the memory that malloc() points to
- Takes the pointer that was returned from malloc to find the memory
- It is the user’s job to deallocate the space when it is not needed anymore
- To know how much memory that is supposed to be freed, the memory block keeps the size in a header block that lies just before the actual memory

Free list

- Data structure used for dynamic memory allocation
- Contains blocks of the available free space on the heap
- There exist different strategies used to select the appropriate memory that is requested from the user: best fit, worst fit, first fit, next fit, buddy allocation

Splitting and coalescing

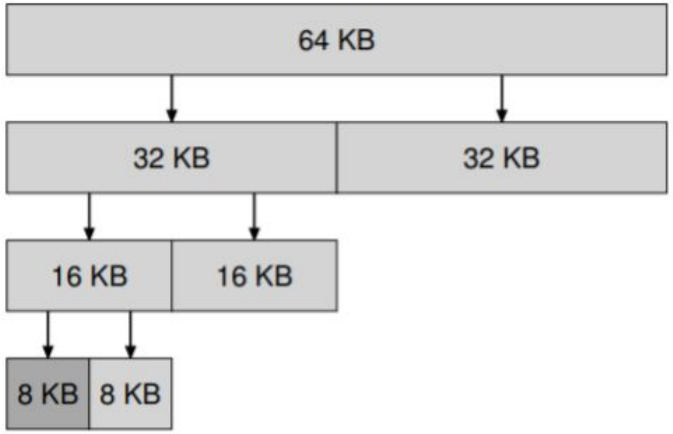
- Common techniques used in memory allocation
- Split a memory block in the free list
    - Return the needed amount to the user
    - Keep the rest
- When coalescing you merge two blocks of free memory that lies next to each other
    - Good to make sure that the heap is not divided into several small blocks of memory

Buddy allocation

- Used to make coalescing easier
- The free memory is seen as 2^n
- When a request for memory is made the memory is recursively splitted in two until an appropriate size is found
- Pros – Easy to determine if two blocks can be merged together
- Cons – Internal fragmentation as only blocks of size 2^n can be given

Buddy allocation - example

- 64 KB free space
- User request for 7 KB block of memory
- The memory block is divided in 2^n
- 8 KB is returned to the user



Taken from: Operating Systems: Three Easy Pieces by Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau

Best fit

- Searches through the free list and finds a memory block that is equal to or larger than the requested memory
- Returns the requested amount to the user and keeps the rest
- Pros – Reduces the amount of wasted memory
- Cons – A lot of performance is required to search through the whole list

Worst fit

- Searches through the free list and finds the largest amount of free memory
- Returns the requested amount to the user and keeps the rest
- Pros – Reduces the amount of small memory, leaves big pieces of memory free
- Cons – A lot of performance, the whole list needs to be searched through, takes away large chunks of memory that could be needed

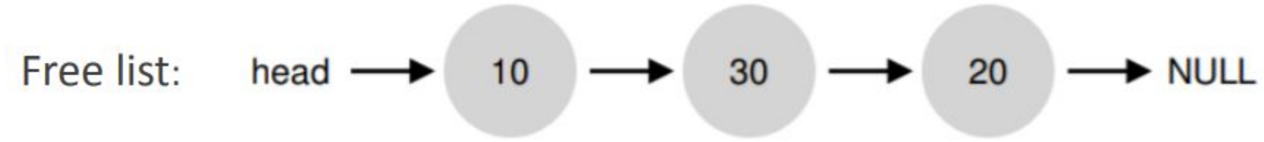
First fit

- Finds the first block of memory that is large enough
- Returns the requested amount to the user and keeps the rest
- Pros – Quick, no need to search through the whole list
- Cons – Could leave small blocks of memory in the beginning of the list

Next fit

- Similar to first fit but has an extra pointer that keeps track of where you were last time and begins the search from there next time
- Performance is similar to first fit

Free memory strategies - example



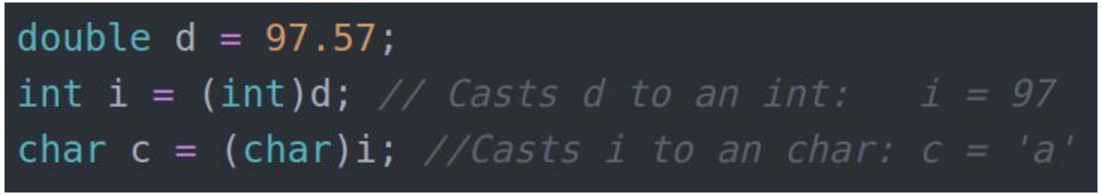
Which block will each strategy choose for a request of size 15?

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

Taken from: Operating Systems: Three Easy Pieces by Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau

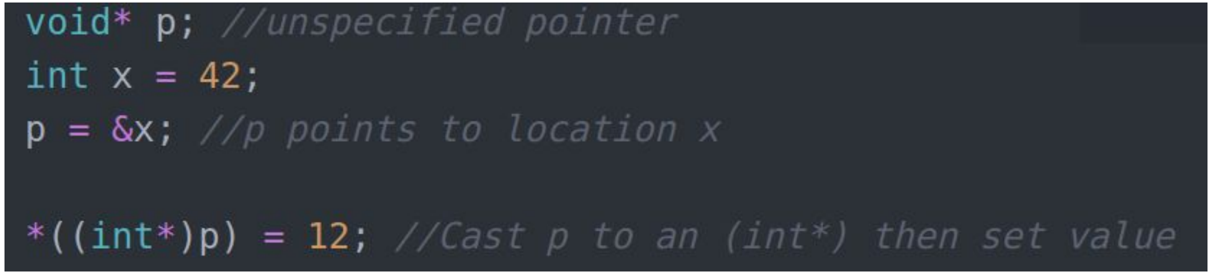
Casting

Since everything is just ***zeros*** or ***ones***, we can tell the computer how to interpret these…
In **C** this is done by writing: (<***type>****)<****expression>***



```c
double d = 97.57;
int i = (int)d; // Casts d to an int:   i = 97
char c = (char)i; //Casts i to an char: c = 'a'
```

Void pointers (***void****) can be thought of as unspecified pointers,
that we later specify using casting.

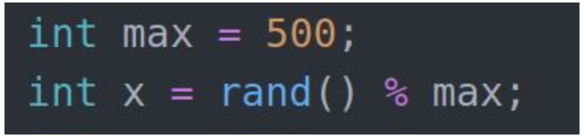


```c
void* p; //unspecified pointer
int x = 42;
p = &x; //p points to location x

*((int*)p) = 12; //Cast p to an (int*) then set value
```

Random Values in C

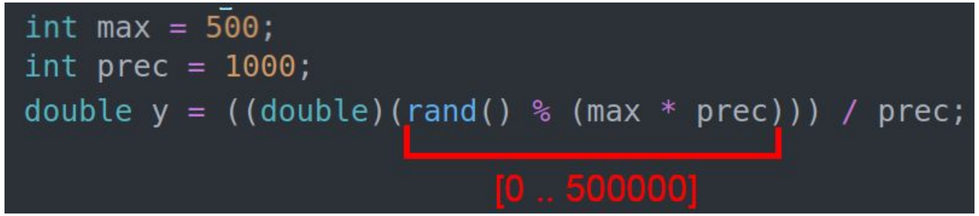
int **rand()** only returns positive integers in the interval ***0* - *RAND_MAX***

*Random **int** in range*
*[**0** .. **max**]*



```c
int max = 500;
int x = rand() % max;
```

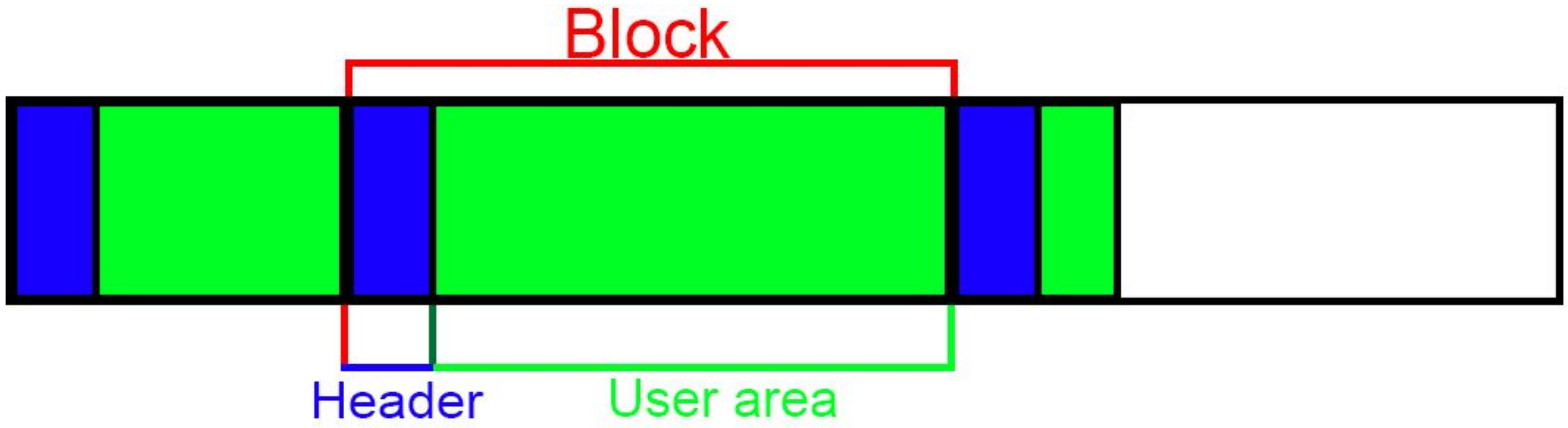
*Random **double** in range*
*[0 .. **max**]*



```c
int max = 500;
int prec = 1000;
double y = ((double)(rand() % (max * prec))) / prec;
```

Things to keep in mind

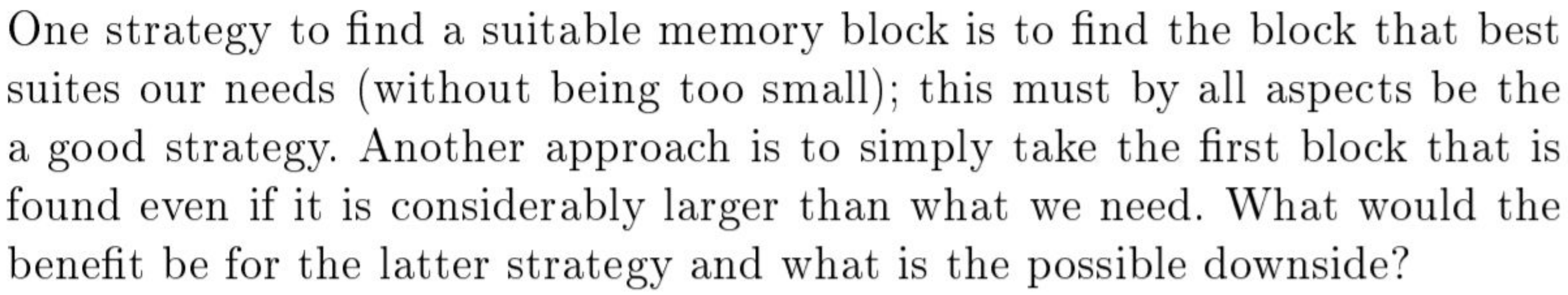
- *int* **atoi**(*string*) *// Converts a string to an int*
- *size_t* **sizeof**(*type*) *// Size of the given datatype, (useful for **structs**)*
- *size_t* *// Best fitting **unsigned** datatype,*
- *int* **rand()** *// Returns a random value between **0** & **RAND_MAX***



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()**

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

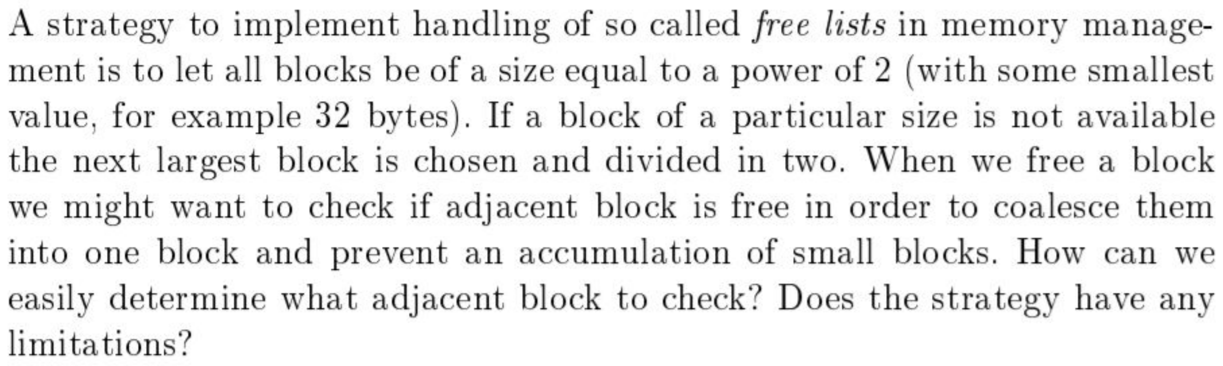
```
__thread chunk *free = NULL;

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.

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.