Operating Systems and Architecture (CO527) Spring 2016

Dynamic Memory

Fred Barnes and Radu Grigore

High Level Interface: malloc and free

The functions malloc and free are typically implemented in user space; for example, in glibc. Nevertheless, we will study them because (1) they are part of POSIX, (2) they are simple, and (3) similar ideas are used in later lectures. If your preferred language is Java, then think of x=(T*)malloc(n) as C's counterpart of x=new T[n]: that's how we allocate memory for n objects of type T. There is no counterpart for free(x), however, because Java has garbage collection. In POSIX, you need to collect your own garbage.

Here is a simple program that uses malloc and free:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5    int n; scanf("%d", &n);
6    int *x = malloc(sizeof(int) * n);
7    for (int i = 0; i < n; ++i) scanf("%d",&x[i]);
8    int j; scanf("%d", &j);
9    printf("%d\n", x[j]);
10   free(x);
11 }
```

The important lines are 6 and 10. On line 6 we request enough space to store n integers; on line 10 we release that space. Note that the value of n is unknown when we write the program: we have to ask the user (on line 5) what the value of n is. In general, dynamic memory allocation is used when we know only at runtime how much memory is needed.

To use dynamically allocated memory, you only need to understand two functions: malloc and free.

Policies: First Fit, Best Fit, Buddy System

The high level malloc/free interface must be implemented in terms of a lower level interface. This lower level interface is an array of bytes. Moreover, we are not allowed to use any other storage space but this array. In this section, we tackle a simplified problem. We will just worry about which portions of the array we want to allocate, without worrying about which storage space we use for our bookkeeping. We'll worry about bookkeeping in the next section.

The first policy to know is *first fit*: always allocate a block of memory at the smallest address possible. Here is an example:

Call	Return	0	1	2	3	4	5	6	7	8	9	Free	Available	Fragmentation
												10	10	0.00
malloc(3)	0	🟥	🟥	🟥								7	7	0.00
malloc(2)	3	🟥	🟥	🟥	🟥	🟥						5	5	0.00
malloc(1)	5	🟥	🟥	🟥	🟥	🟥	🟥					4	4	0.00
free(3)		🟥	🟥	🟥			🟥					6	4	0.33
malloc(3)	6	🟥	🟥	🟥			🟥	🟥	🟥	🟥		3	2	0.33
malloc(1)	3	🟥	🟥	🟥	🟥		🟥	🟥	🟥	🟥		2	1	0.50
free(3)		🟥	🟥	🟥			🟥	🟥	🟥	🟥		3	2	0.33
free(5)		🟥	🟥	🟥				🟥	🟥	🟥		4	3	0.25
free(0)								🟥	🟥	🟥		7	6	0.17

The call free(x) refers to the block at address x, which was allocated by the last malloc that returned x.

The next policy to know is next fit, but we skip it because it doesn't work that well in practice. After that comes *best fit*: put a block in the smallest gap where it fits. If there are several smallest gaps, then the one at the smallest address is chosen. In the example from above, best fit would only differ in how it handles the last malloc:

Call	Return	0	1	2	3	4	5	6	7	8	9	Free	Available	Fragmentation
		🟥	🟥	🟥			🟥	🟥	🟥	🟥		3	2	0.33
malloc(1)	9	🟥	🟥	🟥			🟥	🟥	🟥	🟥	🟥	2	2	0.00

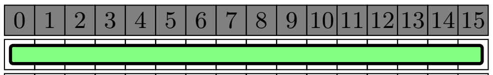
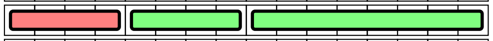
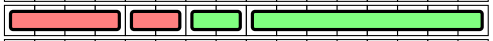
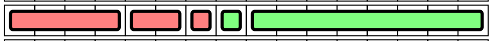
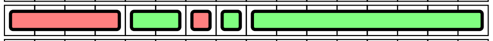
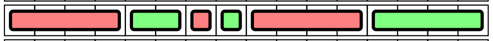
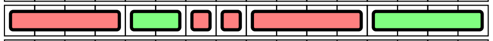
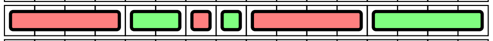
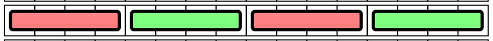
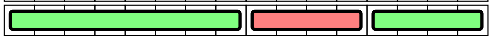
First fit, next fit, best fit and their cousins suffer from *external fragmentation*: The largest block that can be allocated is smaller than the total amount of free memory.

$$\text{external fragmentation} = 1 - \frac{\text{biggest available block}}{\text{total free memory}}$$

The next policy, (binary) *buddy system*, tends to have much lower external fragmentation. The buddy system works when the size of the memory is a power of two, and it allocates only blocks whose size is a power of two and are aligned. For example, if the memory has size 2^4, then *only* the following blocks can be allocated:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Order
🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	0
🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	1
🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	2
🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	3
🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	4

Here is how the buddy system handles the use case from before:

Call	Return	Memory	Fragmentation	
			Internal	External
			0.00	0.00
malloc(3)	0		0.33	0.33
malloc(2)	4		0.20	0.20
malloc(1)	6		0.17	0.11
free(4)			0.25	0.27
malloc(3)	8		0.29	0.43
malloc(1)	7		0.25	0.33
free(7)			0.29	0.43
free(6)			0.33	0.50
free(0)			0.33	0.33

It turns out that in this particular example the external fragmentation is not smaller than for first fit. In addition, we also have *internal fragmentation*: there is more allocated memory than was requested. For example, if 3 cells are requested, then 4 are allocated, because the size of a block must be a power of two. In the table above, internal fragmentation is computed as follows:

$$\text{internal fragmentation} = \frac{\text{allocated memory}}{\text{requested memory}} - 1$$

When free(7) is called, two gluing operations take place: first, the block [6] is glued with its buddy [7] to form the block [6, 7]; second, the block [4, 5] is glued with its buddy [6, 7] to form the block [4, 5, 6, 7]. In general, each block of size 2^{k+1} that starts at address $n \cdot 2^{k+1}$ is made out of two *buddies* of size 2^k, one starting at address $2n \cdot 2^k$ and one starting at address $(2n + 1) \cdot 2^k$. Blocks are split into buddies when smaller blocks are needed. There are never two free buddies — they get glued into a bigger block.

Observe that the policies from above work for arrays of x for all x, not only for x = bytes. When the type x is irrelevant, we talk about an 'array of cells', as opposed to 'array of bytes' or 'array of integers'.

Implementation

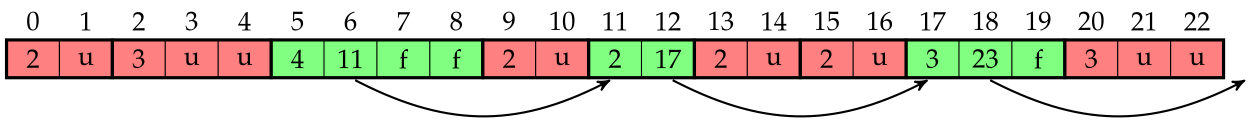
To implement the policies from the previous section, we have some bookkeeping to do. How do we know which cells are allocated and which are free? When we are given the start address of a block, how do we know how many cells are we supposed to free? How do we find the free cells?

Before reading on, you may want to try coming up with a solution yourself. You have to keep in mind that the only storage space you have is the given array. This means that it's OK to ignore the need for a constant number of variables, but not OK to ignore the need for a variable amount of space. Why? Well, if you need, say, 10 integers, then you just reserve enough space at the end of the given array, and then pretend the array is smaller. However if the extra space is not a fixed number like 10, then you need to think carefully about where you put that information. In particular, you can't assume you have something like malloc (or new), because you don't.

The data structure most often to solve this problem is a free list. We will take a look at its structure, and then we'll revisit the policies we saw. The presentation makes the simplifying assumption that each array cell holds and address. (On the one hand, this is not quite true: memory looks more like an array of bytes, and one needs ~ 8 bytes to store an address. On the other hand, removing this simplifying assumption doesn't involve any interesting idea.)

Free List

A free list is just what the name says: a list of free blocks of memory. For example:



In the picture, we see an array of size 23, whose cells are indexed by $0, 1, 2, \dots, 22$. The free list starts at index 5: we need to store this information somewhere, separately. The free block starting at index 5 has size 4: we find the 4 written in the cell with index 5. The next free block starts at index 11: we find the 11 written in the cell with index $5 + 1$. Similarly, at index 11 we have the size of a free block, and at index $11 + 1$ we have the index where the next free block starts. And so on, until we find the index 23, which is just after the end of the array.

We will restrict the size of *any* block to be ≥ 2. Notice that free blocks need 2 cells, because their first cell contains the size of the block and their second cell contains the index of the next free block. Technically, used blocks only need 1 cell, to store their own size; however, allowing used blocks to be smaller than free blocks would make it cumbersome to transform a used block into a free one.

First Fit, Best Fit

When malloc(n) is called, we begin traversing the free list to find a free block. Let m be the size of the free block. (We will see in a moment how big m should be.) We split the free block in two: a used block of size $n + 1$, and a free block of size $m - (n + 1)$. The used block starts with a cell that stores the value $n + 1$, and continues with the n cells we reserved for the user. Of course, we need to satisfy the constraints we have on the sizes of blocks: $n + 1 \geq 2$ and $m - (n + 1) \geq 2$. We satisfy the first constraint by requesting users to call malloc(n) only with $n \geq 1$. We satisfy the second constraint by finding a free block of size $m \geq n + 3$. The *first fit* strategy finds the first block in the free list whose size is at least $n + 3$. The *best fit* strategy finds the smallest block in the free list whose size is at least $n + 3$. Of course, in both cases, a big enough free block might not exist, in which case malloc fails.

When free(j) is called, we take the used block starting at index $j - 1$ and insert it into the free list. How should this insertion be done? One option would be to simply make the new block be the first block in the free list. This can be done quickly because it does not require traversing the free list. Most implementations, however, enforce the invariant that all arrows of the free list point from left to right. One reason is simplicity: With this convention we can speak of 'the previous free block' without worrying whether we mean previous in the array or previous in the list. To keep all arrows from left to right, we must traverse the free list until we find two blocks that sandwich $j - 1$. Then we can insert the freed block in the free list in its proper place.

There are two other reasons for keeping all arrows pointing right: (1) it makes the traversal of the list more cache friendly, and (2) it simplifies gluing free blocks. We will speak about caches in the next lecture. Now, let us speak about gluing blocks.

Why would we ever need to glue blocks? Well, with the algorithm described so far, malloc splits blocks into smaller ones, and free does not change the size or the number of blocks. Thus, after a long sequence of malloc and free operations we will have smaller and smaller blocks, making it more and more likely that we will not have a big enough block to satisfy the next malloc. To counteract such external fragmentation, we glue free blocks that touch each-other. In other words, we enforce the invariant that free blocks cannot touch.

Observe that malloc and free are rather complicated. They are not constant time operations, as most developers assume. You can find a C implementation for first fit, as described above, in the accompanying code.

Buddy System

In the buddy system block sizes can only be a power of two. So, for example, if the memory has size 2^{32}, then the blocks can have sizes $2^0, 2^1, 2^2, \dots, 2^{32}$. A block of size 2^k is said to have *order* k. All the free blocks of order 0 are chained in the free list 0; all the free blocks of order 1 are chained in the free list 1; and so on. Whenever a block of order $k + 1$ is split into two blocks of order k, we need to remove it from the free list $k + 1$ and to insert the two buddies in the free list k. Similarly, when gluing two buddies of order k, we need to appropriately update the free lists k and $k+1$. For each free list, we typically enforce the same invariants that we had for first-fit and best-fit.

Garbage Collection*

With garbage collection, the user interface is even simpler: instead of two functions (one for allocating memory and one for deallocating memory), there is only one function, for allocating memory.

This topic is optional because few operating systems implement garbage collection. Linux uses garbage collection internally in specific places only; for example, see net/unix/garbage.c. However, this example doesn't really count because the garbage collector is used internally, not exposed to apps. The Singularity operating system does implement several garbage collectors, *and* allows apps to use them. However, this example doesn't really count either because Singularity is a research prototype only. Still, many high-level languages do implement garbage collection, and it is conceivable that this functionality will eventually migrate in the operating system. So, you may want to know a bit about garbage collection.

Let us look at a couple of differences between garbage collection and the malloc/free method.

A first difference is that programmers need not worry about when to call free. Despite what it may seem, such a simplification makes programming significantly easier. The flip side is that the programmer doesn't get to determine when memory is deallocated. To emphasize this difference, the malloc/free method is sometimes called *deterministic* memory management.

A second difference is that external fragmentation is less of a problem with garbage collection. Most high-level languages that implement garbage collection distinguish between object references and memory addresses. As a result, the garbage collector is allowed to change the memory address without changing the object reference. In other words, the garbage collector gets to move objects around in memory, without the programmer ever knowing that this happens! In the *compaction* phase, the garbage collector moves objects so they occupy a contiguous region of memory.

Finally, programmers do need to know how their garbage collector works, occasionally. For example, if the garbage collector interrupts the execution of the program for compaction, then you may have trouble implementing a real-time system, which must obey hard deadlines. Other garbage collectors, however, may run concurrently, in a way that never interrupts the main program for long. So, occasionally, programmers may need to explicitly choose which garbage collector to use.

This was quite enough about garbage collection. If you want to know more, go and ask Richard Jones. You are lucky to have him in Canterbury – he is **the** garbage collection person.

Exercises

1. ► Run 'man malloc', and read.
2. Look at the code example that uses malloc and free. Can you explain what each statement does?
3. Write a C program that reverses a list of integers. The input is formed of $n+1$ whitespace separated integers: '$n\ x_1 \ldots x_n$'. The output is formed of n space separated integers: '$x_n \ldots x_1$'. You will need to use malloc to allocate space for integers. Make sure you clean up after yourself. (Use valgrind to check you did so.)
4. The functions malloc and free are in fact implemented on top of sbrk and mmap. The low level interface is not one big array of bytes, but rather several big array of bytes, called *arenas*. To get a new arena, you use mmap; to enlarge an arena, you use sbrk. (Both operations may fail.) Take a look at man sbrk and at man mmap.
5. ► If we want to enforce that all arrows of a free list point to the right, free(j) must find two free blocks that sandwich $j-1$. What if $j-1$ comes after the last free block? What if $j-1$ comes before the first free block? Can you handle the latter case by initialising the array with some special blocks?
6. Implement best-fit memory allocation, in C.

7. Download the source code of glibc, the **G**NU implementation of the standard **lib**rary of **C**. Then look at the real implementation of malloc, in malloc/malloc.c.

8. Implement the buddy system, in C.

References

[1] Wilson et al., *Dynamic Storage Allocation: A Survey and Critical Review*, 1995

[2] Knuth, *The Art of Computer Programming*, Volume 1, Section 2.5 (*Dynamic Storage Allocation*), 1997

[3] Hunt, Larus, *Singularity: Rethinking the Software Stack*, 2007