The Heap

In order to be responsive to situations that weren’t envisaged at the design stage, and to cut down the size of the kernel, we need some kind of dynamic memory allocation. The algorithm that Pro-Type uses is rather simple, it has two main goals:

* (Relatively) simple to implement.
* Ability to check consistency.

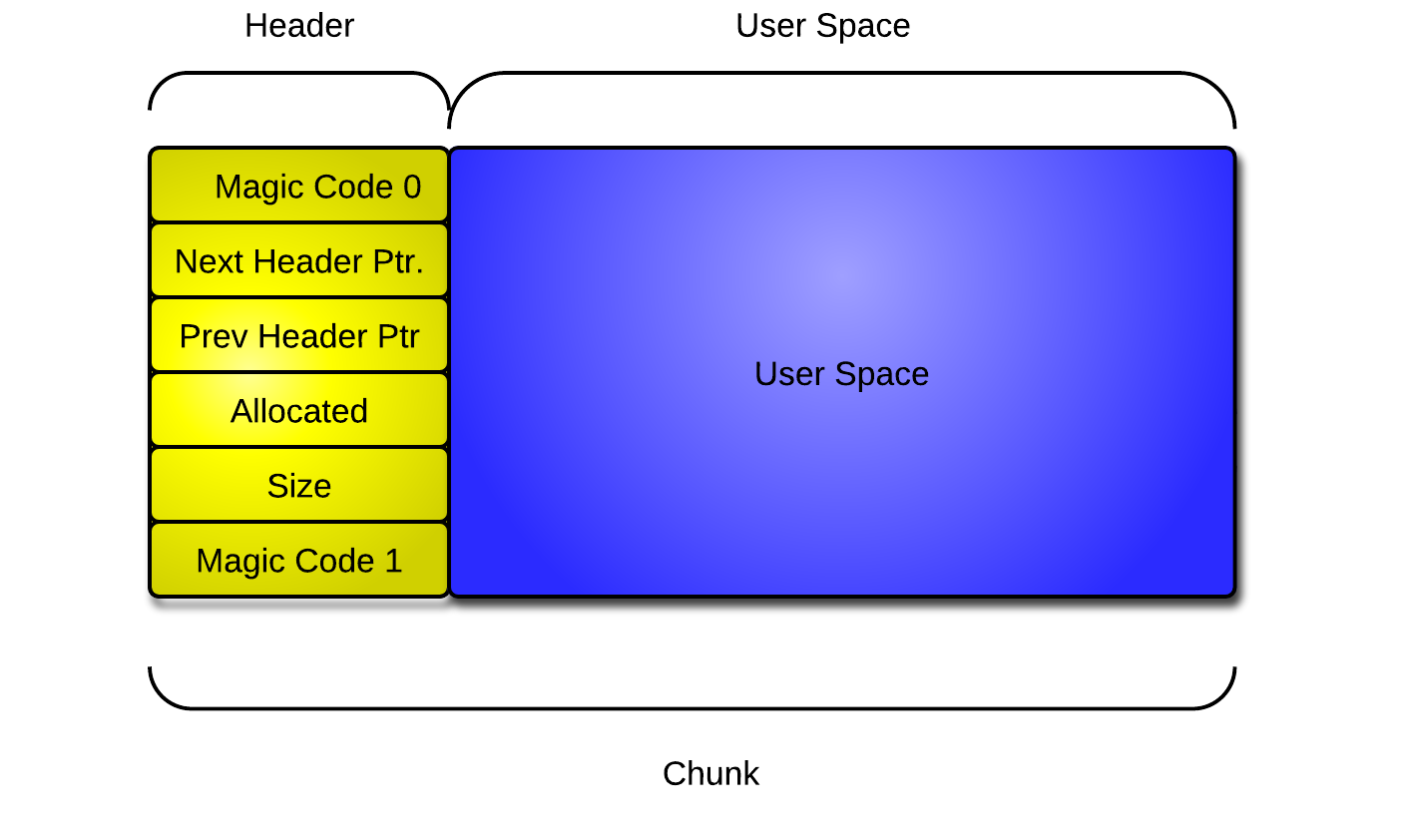
The algorithm and data structures presented here come from a basic idea that a lot of hobby programmers have used because of its simplicity. However it’s not very stable, because of its high dependence on the virtual memory manager, it can crash easily if there is a bug in that manager. It can also crash if the process doesn’t respect the boundaries of the allocated space.

# The Algorithm

The algorithm uses two concepts: allocated blocks and holes. Blocks are contiguous areas of memory containing user data currently in use (i.e. allocated but not freed). Holes are unallocated blocks, so their contents are not in use. So initially by this concept the entire area of heap space is one large hole.

We will keep every block and hole in a linked list, so in order to find free space we can just iterate over it and check the entries. Blocks and holes each contain a header filled with descriptive data. The header contains information about the length of a block, whether it is a hole or not and links to the next and previous items in the list. The header and the piece of memory together form a chunk of memory. The header is protected by two double words containing a specified value: the magic code. Every allocation these get checked to make sure the heap isn’t corrupted.

The headers of every block and hole form that linked list through a series of pointers. Here you can see a representation of a chunk:



# Algorithm description

Below I will describe the algorithm that is used to allocate and to free a chunk of the heap. These are simple algorithms that are fast and simple to implement and relatively efficient. Below you can also find flowcharts describing the same algorithms.

### Allocation

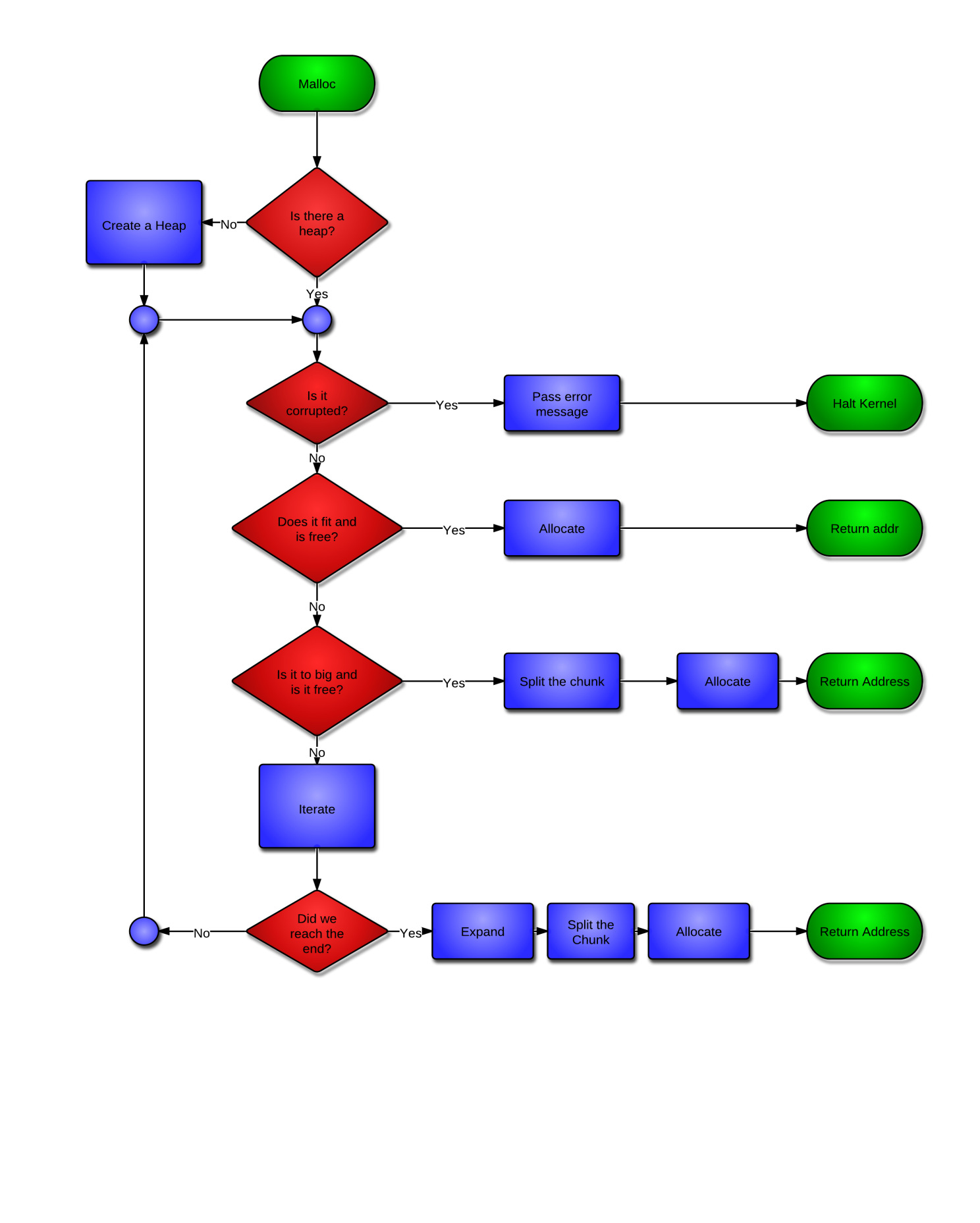
Allocation is pretty straightforward:

* Iterate the list to find the first hole that fits our request.
* If the hole fits:
* Adjust the new block's header to be allocated and return the address + sizeof(header\_t).
* If the hole’s size is larger than we need:
* Split the chunk.
* Adjust the new block's header to be allocated and return the address + sizeof(header\_t).
* If we didn't find a hole large enough, then:
* Expand the heap.
* Split the new chunk.
* Adjust the new block's header to be allocated and return the address + sizeof(header\_t).

### Deallocation

* Find the header by taking the given pointer and subtracting the sizeof(header\_t).
* Set the allocated flag in our header to 0.
* If the next header is a hole:
* Merge the next chunk and our chunk.
* If the previous header is a hole:
* Merge the previous chunk and our chunk.
* If our hole is the last in the heap (header->next == 0 ):
* Unmap that address, free the physical page.

# Malloc: Flowchart



# Free: Flowchart

