www.jamesmolloy.co.uk

Home » JamesM's kernel development tutorials

7. The Heap

In order to be responsive to situations that you didn't envisage at the design stage, and to cut down the size of your kernel, you will need some kind of dynamic memory allocation. The current memory allocation system (allocation by placement address) is absolutely fine, and is in fact optimal for both time and space for allocations. The problem occurs when you try to free some memory, and want to reclaim it (this must happen eventually, otherwise you will run out!). The placement mechanism has absolutely no way to do this, and is thus not viable for the majority of kernel allocations.

As a sidepoint of general terminology, *any* data structure that provides both allocation and deallocation of contiguous memory can be referred to as a heap (or a pool). There is, as such, no standard 'heap algorithm' - Different algorithms are used depending on time/space/efficiency requirements. *Our* requirements are:

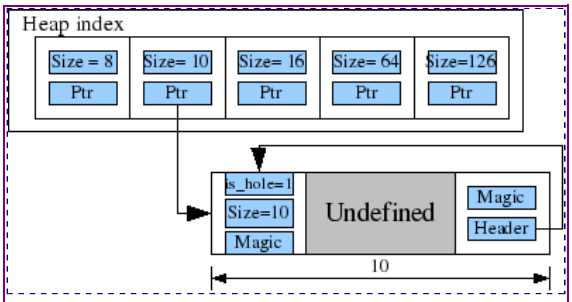
- (Relatively) simple to implement.
- Able to check consistency - debugging memory overwrites in a kernel is about ten times more difficult than in normal apps!

The algorithm and data structures presented here are ones which I developed myself. They are so simple however, that I am sure others will have used it first. It is similar to (though more simple than) Doug Lea's malloc which is used in the GNU C library.

7.1. Data structure description

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

For every hole there is a corresponding descriptor in an index table. The index table is always ordered ascending by the size of the hole pointed to.



The index table with pointers to holes

Blocks and holes each contain descriptive data - a header and a footer. The header contains the most information about the block - the footer merely contains a pointer to the header (the reason for the footer will become apparent soon). Pseudocode:

```
typedef struct
{
  u32int magic;  // Magic number, used for error checking and identification.
  u8int is_hole; // 1 if this is a hole, 0 if this is a block.
  u32int size;   // Size of the block, including this and the footer.
} header_t;

typedef struct
{
  u32int magic;     // Magic number, same as in header_t.
  header_t *header; // Pointer to the block header.
} footer_t;
```

Notice that each also has a 'magic number' field. This is for error checking, and later will play a part in our 'free' algorithm. This is just a sentinel number - an unusual number that will stand out from others - much like 0xdeadbaba that we used in chapter 2. In the sample code I've gone for 0x123890AB arbitrarily.

Note also that within this tutorial I will refer to the *size* of a block being the number of bytes from the start of the header to the end of the footer - so within a block of size *x*, there will be x - sizeof(header_t) - sizeof(footer_t) user-useable bytes.

7.2. Algorithm description

7.2.1. Allocation

Allocation is straightforward, if a little long-winded. Most of the steps are error-checking and creating new holes to minimise memory leaks.

1. Search the index table to find the smallest hole that will fit the requested size. As the table is ordered, this just entails iterating through until we find a hole which will fit.
    - If we didn't find a hole large enough, then:
        1. Expand the heap.
        2. If the index table is empty (no holes have been recorded) then add a new entry to it.
        3. Else, adjust the last header's size member and rewrite the footer.
        4. To ease the number of control-flow statements, we can just recurse and call the allocation function again, trusting that this time there will be a hole large enough.
2. Decide if the hole should be split into two parts. This will normally be the case - we usually will want much less space than is available in the hole. The only time this will not happen is if there is less free space after allocating the block than the header/footer takes up. In this case we can just increase the block size and reclaim it all afterwards.
3. If the block should be page-aligned, we must alter the block starting address so that it is and create a new hole in the new unused area.
    - If it is not, we can just delete the hole from the index.
4. Write the new block's header and footer.
5. If the hole was to be split into two parts, do it now and write a new hole into the index.
6. Return the address of the block + sizeof(header_t) to the user.

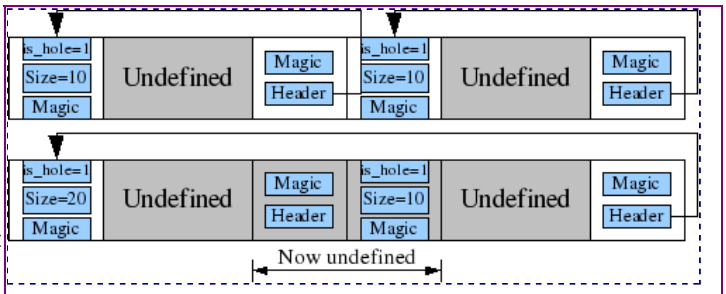
7.2.2. Deallocation

Deallocation (freeing) is a little more tricky. As mentioned earlier, this is where the efficiency of a memory-management algorithm is really tested. The problem is effective reclaimation of memory. The naive solution would be to change the given block to a hole and enter it back into the hole index. However, if I do this:

```
int a = kmalloc(8); // Allocate 8 bytes: returns 0xC0080000 for sake of argument
int b = kmalloc(8); // Allocate another 8 bytes: returns 0xC0080008.
kfree(a);           // Release a
kfree(b);           // Release b
int c = kmalloc(16);// What will this allocation return?
```

Note that in this example the space required for headers and footers have been purposely omitted for readability

Here we have allocated space for 8 bytes, twice. We then release both of those allocations. With the naive release algorithm we would then end up with two 8-byte sized holes in the index. When the next allocation (for 16 bytes) comes along, neither of those holes can fit it, so the kmalloc() call will return 0xC0080010. This is suboptimal. There are 16 bytes of space free at 0xC0080000, so we *should* be reallocating that!



Unifying the two allocations in the top diagram into one in the lower diagram.

The solution to this problem in most cases is a varation on a simple algorithm that I call *unification* - That is, converting two adjacent holes into one. (Please note that this coining of a term is not from a sense of self-importance, merely from the absence of a standardised name).

It works thus: When free()ing a block, look at what is immediately to the left (assuming 0-4GB left-to-right) of the header. If that is a footer, which can be discovered from the value of the magic number, then follow the pointer to it's header and query whether it is a hole or a block. If it is a hole, we can modify it's header's size attribute to take into account both it's size and ours, then point *our* footer to *it's* header.

We have thus amalgamated both holes into one (and in this case there is no need to do an expensive *insert* operation on the index).

That is what I call *unifying left*. There is also *unifying right*, which should be performed on free() as well. Here we look at what is directly after the footer. If we find a header there, again identified by it's magic number, we check if it is a hole. We can then use it's size attribute to find it's footer. We rewrite the footer's pointer to point to *our* header. Then, all that needs to be done is to remove it's old entry from the hole index, and add our own.

Note also that in the name of reclaiming space, if we are free()ing the last block in the heap (there are no holes or blocks after us), then we can contract the size of the heap. To avoid this happening constantly, in my implementation I have defined a minimum heap size, below which it will not contract.

7.2.2.1. Pseudocode

1. Find the header by taking the given pointer and subtracting the sizeof(header_t).
2. Sanity checks. Assert that the header and footer's magic numbers remain in tact.
3. Set the is_hole flag in our header to 1.
4. If the thing immediately to our left is a footer:
    - Unify left. In this case, at the end of the algorithm we shouldn't add our header to the hole index (the header we are unifying with is already there!) so set a flag which the algorithm checks later.
5. If the thing immediately to our right is a header:
    - Unify right.
6. If the footer is the last in the heap (footer_location+sizeof(footer_t) == end_address):
    - Contract.
7. Insert the header into the hole array unless the flag described in *Unify left* is set.

7.3. Implementing an ordered list

So now we come to the implementation. As usual I'm going to try and explain the utility datatypes and functions first, and finish up with the allocation/free functions themselves.

The first datatype we need it an implementation of an ordered list. This concept will be used multiple times in your kernel (it is a common requirement) so it is probably a good idea to abstract it, so it can be used again.

7.3.1. ordered_array.h

```
// ordered_array.h -- Interface for creating, inserting and deleting
// from ordered arrays.
// Written for JamesM's kernel development tutorials.

#ifndef ORDERED_ARRAY_H
#define ORDERED_ARRAY_H

#include "common.h"

/**
  This array is insertion sorted - it always remains in a sorted state (between calls).
  It can store anything that can be cast to a void* -- so a u32int, or any pointer.
**/
typedef void* type_t;
/**
  A predicate should return nonzero if the first argument is less than the second. Else
  it should return zero.
**/
typedef s8int (*lessthan_predicate_t)(type_t,type_t);
typedef struct
{
   type_t *array;
   u32int size;
   u32int max_size;
   lessthan_predicate_t less_than;
} ordered_array_t;

/**
  A standard less than predicate.
**/
```

```
s8int standard_lessthan_predicate(type_t a, type_t b);

/**
  Create an ordered array.
**/
ordered_array_t create_ordered_array(u32int max_size, lessthan_predicate_t less_than);
ordered_array_t place_ordered_array(void *addr, u32int max_size, lessthan_predicate_t less_than);

/**
  Destroy an ordered array.
**/
void destroy_ordered_array(ordered_array_t *array);

/**
  Add an item into the array.
**/
void insert_ordered_array(type_t item, ordered_array_t *array);

/**
  Lookup the item at index i.
**/
type_t lookup_ordered_array(u32int i, ordered_array_t *array);

/**
  Deletes the item at location i from the array.
**/
void remove_ordered_array(u32int i, ordered_array_t *array);

#endif // ORDERED_ARRAY_H
```

Notice that in the name of abstraction we have made the 'less than' function user-defineable. We will use this in the heap implementation to order by size and not pointer address. Note also we have two methods of defining an ordered_array. *create_ordered_array* will use kmalloc() to get some space. *place_ordered_array* will use the given start location. As we want to put our heap in a specific place (and because kmalloc isn't yet working!) we use place_ordered_array in our heap code.

7.3.2. ordered_map.c

```
// ordered_array.c -- Implementation for creating, inserting and deleting
// from ordered arrays.
// Written for JamesM's kernel development tutorials.

#include "ordered_array.h"

s8int standard_lessthan_predicate(type_t a, type_t b)
{
   return (a<b)?1:0;
}

ordered_array_t create_ordered_array(u32int max_size, lessthan_predicate_t less_than)
{
   ordered_array_t to_ret;
   to_ret.array = (void*)kmalloc(max_size*sizeof(type_t));
   memset(to_ret.array, 0, max_size*sizeof(type_t));
   to_ret.size = 0;
   to_ret.max_size = max_size;
   to_ret.less_than = less_than;
   return to_ret;
}

ordered_array_t place_ordered_array(void *addr, u32int max_size, lessthan_predicate_t less_than)
{
   ordered_array_t to_ret;
   to_ret.array = (type_t*)addr;
   memset(to_ret.array, 0, max_size*sizeof(type_t));
   to_ret.size = 0;
   to_ret.max_size = max_size;
   to_ret.less_than = less_than;
   return to_ret;
}
```

```
void destroy_ordered_array(ordered_array_t *array)
{
// kfree(array->array);
}

void insert_ordered_array(type_t item, ordered_array_t *array)
{
   ASSERT(array->less_than);
   u32int iterator = 0;
   while (iterator < array->size && array->less_than(array->array[iterator], item))
       iterator++;
   if (iterator == array->size) // just add at the end of the array.
       array->array[array->size++] = item;
   else
   {
       type_t tmp = array->array[iterator];
       array->array[iterator] = item;
       while (iterator < array->size)
       {
           iterator++;
           type_t tmp2 = array->array[iterator];
           array->array[iterator] = tmp;
           tmp = tmp2;
       }
       array->size++;
   }
}

type_t lookup_ordered_array(u32int i, ordered_array_t *array)
{
   ASSERT(i < array->size);
   return array->array[i];
}

void remove_ordered_array(u32int i, ordered_array_t *array)
{
   while (i < array->size)
   {
       array->array[i] = array->array[i+1];
       i++;
   }
   array->size--;
}
```

Hopefully nothing there should surprise you. On *insert* the item is placed at the correct position and all larger other items shifted up one position. As always with these satellite datatypes, any implementation will work. There **are** better implementations of ordered arrays than this (c.f. heap-ordering, binary search trees), but I decided to go with a simple one for teaching purposes.

7.4. The heap itself

7.4.1. kheap.h

Some #defines and function prototypes are useful:

```
#define KHEAP_START         0xC0000000
#define KHEAP_INITIAL_SIZE  0x100000
#define HEAP_INDEX_SIZE   0x20000
#define HEAP_MAGIC        0x123890AB
#define HEAP_MIN_SIZE     0x70000

/**
  Size information for a hole/block
**/
typedef struct
{
   u32int magic;   // Magic number, used for error checking and identification.
   u8int is_hole;   // 1 if this is a hole. 0 if this is a block.
```

```
   u32int size;    // size of the block, including the end footer.
} header_t;

typedef struct
{
   u32int magic;     // Magic number, same as in header_t.
   header_t *header; // Pointer to the block header.
} footer_t;

typedef struct
{
   ordered_array_t index;
   u32int start_address; // The start of our allocated space.
   u32int end_address;   // The end of our allocated space. May be expanded up to max_address.
   u32int max_address;   // The maximum address the heap can be expanded to.
   u8int supervisor;     // Should extra pages requested by us be mapped as supervisor-only?
   u8int readonly;       // Should extra pages requested by us be mapped as read-only?
} heap_t;

/**
  Create a new heap.
**/
heap_t *create_heap(u32int start, u32int end, u32int max, u8int supervisor, u8int readonly);
/**
  Allocates a contiguous region of memory 'size' in size. If page_align==1, it creates that block starting
  on a page boundary.
**/
void *alloc(u32int size, u8int page_align, heap_t *heap);
/**
  Releases a block allocated with 'alloc'.
**/
void free(void *p, heap_t *heap);
```

I have decided, arbitrarily, to put the kernel heap at 0xC0000000, give it's index a size of 0x20000 bytes, and give it a minimum size of 0x70000 bytes. The header and footer structures are the same as those given at the top of the chapter. We can actually have more than one heap (in my own kernel I have a user-mode heap as well), so for ease of portability I have decided to implement the heap as a datatype itself. heap_t keeps track of the heap's index, start/end/max addresses and the modifiers to give alloc_page when requesting more memory.

7.4.2. kheap.c

Finding the smallest hole that will fit a certain number of bytes is a common task that gets called on every allocation. It would therefore be nice to wrap this up in a function:

```
static s32int find_smallest_hole(u32int size, u8int page_align, heap_t *heap)
{
   // Find the smallest hole that will fit.
   u32int iterator = 0;
   while (iterator < heap->index.size)
   {
       header_t *header = (header_t *)lookup_ordered_array(iterator, &heap->index);
       // If the user has requested the memory be page-aligned
       if (page_align > 0)
       {
           // Page-align the starting point of this header.
           u32int location = (u32int)header;
           s32int offset = 0;
           if ((location+sizeof(header_t)) & 0xFFFFF000 != 0)
               offset = 0x1000 /* page size */  - (location+sizeof(header_t))%0x1000;
           s32int hole_size = (s32int)header->size - offset;
           // Can we fit now?
           if (hole_size >= (s32int)size)
               break;
       }
       else if (header->size >= size)
           break;
       iterator++;
   }
   // Why did the loop exit?
```

```
   if (iterator == heap->index.size)
       return -1; // We got to the end and didn't find anything.
   else
       return iterator;
}
```

I feel I should explain two lines:

```
if ((location+sizeof(header_t)) & 0xFFFFF000 != 0)
  offset = 0x1000 /* page size */  - (location+sizeof(header_t))%0x1000;
```

It's important to note that when the user requests memory to be page-aligned, he is requesting the memory *that he has access to* to be page-aligned. That means that the header address will actually *not* be page-aligned. The address that we want to fall on a boundary is location + sizeof(header_t).

Creating a heap is a simple procedure. The only part worthy of note is that we set aside the first HEAP_INDEX_SIZE*sizeof(type_t) bytes as the index. The index is put there using place_ordered_array, and the effective start address is shifted forwards. That is why, when testing your kernel, you will see allocations starting at 0xC0080000 instead of the more obvious 0xC0000000. Also note that we create a custom less_than function for the index array. This is because with the standard less_than function the array would be sorted by *pointer address*, instead of by size.

```
static s8int header_t_less_than(void*a, void *b)
{
   return (((header_t*)a)->size < ((header_t*)b)->size)?1:0;
}

heap_t *create_heap(u32int start, u32int end_addr, u32int max, u8int supervisor, u8int readonly)
{
   heap_t *heap = (heap_t*)kmalloc(sizeof(heap_t));

   // All our assumptions are made on startAddress and endAddress being page-aligned.
   ASSERT(start%0x1000 == 0);
   ASSERT(end_addr%0x1000 == 0);

   // Initialise the index.
   heap->index = place_ordered_array( (void*)start, HEAP_INDEX_SIZE, &header_t_less_than);

   // Shift the start address forward to resemble where we can start putting data.
   start += sizeof(type_t)*HEAP_INDEX_SIZE;

   // Make sure the start address is page-aligned.
   if (start & 0xFFFFF000 != 0)
   {
       start &= 0xFFFFF000;
       start += 0x1000;
   }
   // Write the start, end and max addresses into the heap structure.
   heap->start_address = start;
   heap->end_address = end_addr;
   heap->max_address = max;
   heap->supervisor = supervisor;
   heap->readonly = readonly;

   // We start off with one large hole in the index.
   header_t *hole = (header_t *)start;
   hole->size = end_addr-start;
   hole->magic = HEAP_MAGIC;
   hole->is_hole = 1;
   insert_ordered_array((void*)hole, &heap->index);

   return heap;
}
```

7.4.2.1. Expansion and contraction

At points we will need to alter the size of our heap. If we run out of space, we will need more. If we reclaim space, we may need less.

```
static void expand(u32int new_size, heap_t *heap)
{
   // Sanity check.
   ASSERT(new_size > heap->end_address - heap->start_address);
   // Get the nearest following page boundary.
   if (new_size&0xFFFFF000 != 0)
   {
       new_size &= 0xFFFFF000;
       new_size += 0x1000;
   }
   // Make sure we are not overreaching ourselves.
   ASSERT(heap->start_address+new_size <= heap->max_address);

   // This should always be on a page boundary.
   u32int old_size = heap->end_address-heap->start_address;
   u32int i = old_size;
   while (i < new_size)
   {
       alloc_frame( get_page(heap->start_address+i, 1, kernel_directory),
                    (heap->supervisor)?1:0, (heap->readonly)?0:1);
       i += 0x1000 /* page size */;
   }
   heap->end_address = heap->start_address+new_size;
}
```

I think that code is self-explanatory. A few assertions are made, and the new_size parameter is changed so that it falls on a page boundary. Frames are then allocated one-by-one according to the parameters given when creating the heap (supervisor mode enabled?, read only access?).

```
static u32int contract(u32int new_size, heap_t *heap)
{
   // Sanity check.
   ASSERT(new_size < heap->end_address-heap->start_address);
   // Get the nearest following page boundary.
   if (new_size&0x1000)
   {
       new_size &= 0x1000;
       new_size += 0x1000;
   }
   // Don't contract too far!
   if (new_size < HEAP_MIN_SIZE)
       new_size = HEAP_MIN_SIZE;
   u32int old_size = heap->end_address-heap->start_address;
   u32int i = old_size - 0x1000;
   while (new_size < i)
   {
       free_frame(get_page(heap->start_address+i, 0, kernel_directory));
       i -= 0x1000;
   }
   heap->end_address = heap->start_address + new_size;
   return new_size;
}
```

Similarly to expand, new_size is adjusted so it sits on a page boundary. We then check that we're not trying to contract past our minimum size, and free each frame in turn until we reach the desired size.

7.4.2.2. Allocation

We'll talk through the allocation function in parts.

```
void *alloc(u32int size, u8int page_align, heap_t *heap)
{

   // Make sure we take the size of header/footer into account.
   u32int new_size = size + sizeof(header_t) + sizeof(footer_t);
   // Find the smallest hole that will fit.
   s32int iterator = find_smallest_hole(new_size, page_align, heap);

   if (iterator == -1) // If we didn't find a suitable hole
```

```
   {
     ... // Will be filled in in a second.
   }
```

Here we adjust the requested block size to account for the size of the header and footer. We then request the smallest hole available that will fit using our find_smallest_hole function. If we couldn't find one (find_smallest_hole() == -1), we go into some error-handling code. It's a bit beefy, so I'll come back to this to explain it.

```
   header_t *orig_hole_header = (header_t *)lookup_ordered_array(iterator, &heap->index);
   u32int orig_hole_pos = (u32int)orig_hole_header;
   u32int orig_hole_size = orig_hole_header->size;
   // Here we work out if we should split the hole we found into two parts.
   // Is the original hole size - requested hole size less than the overhead for adding a new hole?
   if (orig_hole_size-new_size < sizeof(header_t)+sizeof(footer_t))
   {
       // Then just increase the requested size to the size of the hole we found.
       size += orig_hole_size-new_size;
       new_size = orig_hole_size;
   }
```

Here we get the header pointer from the index given us by find_smallest_hole. We then save the address and size of this header in case we need to overwrite it later. After this, we decide if it is worth splitting the hole in two (that is, will the free space be able to fit another hole into it?) If not, we increase the requested size to the hole size, so it isn't split.

```
   // If we need to page-align the data, do it now and make a new hole in front of our block.
   if (page_align && orig_hole_pos&0xFFFFF000)
   {
       u32int new_location   = orig_hole_pos + 0x1000 /* page size */ - (orig_hole_pos&
0xFFF) - sizeof(header_t);
       header_t *hole_header = (header_t *)orig_hole_pos;
       hole_header->size     = 0x1000 /* page size */ - (orig_hole_pos&0xFFF) - sizeof(header_t);
       hole_header->magic    = HEAP_MAGIC;
       hole_header->is_hole  = 1;
       footer_t *hole_footer = (footer_t *) ( (u32int)new_location - sizeof(footer_t) );
       hole_footer->magic    = HEAP_MAGIC;
       hole_footer->header   = hole_header;
       orig_hole_pos         = new_location;
       orig_hole_size        = orig_hole_size - hole_header->size;
   }
   else
   {
       // Else we don't need this hole any more, delete it from the index.
       remove_ordered_array(iterator, &heap->index);
   }
```

If the user wants his memory to be page-aligned, we facilitate that here. The new location for the header to be placed at is calculated by going to the next page boundary then subtracting the size of a header. The attributes of the new hole's header are then filled in, along with the footer. Note that because we are creating a new hole at the old hole's address, we are essentially reusing the old hole, so there is no need to remove it from the hole index.

```
   // Overwrite the original header...
   header_t *block_header  = (header_t *)orig_hole_pos;
   block_header->magic     = HEAP_MAGIC;
   block_header->is_hole   = 0;
   block_header->size      = new_size;
   // ...And the footer
   footer_t *block_footer  = (footer_t *) (orig_hole_pos + sizeof(header_t) + size);
   block_footer->magic     = HEAP_MAGIC;
   block_footer->header    = block_header;
```

This should be self-explanatory - we make sure all the header and footer attributes are correct, along with magic numbers.

```
   // We may need to write a new hole after the allocated block.
```

```
   // We do this only if the new hole would have positive size...
   if (orig_hole_size - new_size > 0)
   {
       header_t *hole_header = (header_t *) (orig_hole_pos + sizeof(header_t) + size + sizeof(footer_t));
       hole_header->magic    = HEAP_MAGIC;
       hole_header->is_hole  = 1;
       hole_header->size     = orig_hole_size - new_size;
       footer_t *hole_footer = (footer_t *) ( (u32int)hole_header + orig_hole_size - new_size - sizeof(footer_t) );
       if ((u32int)hole_footer < heap->end_address)
       {
           hole_footer->magic = HEAP_MAGIC;
           hole_footer->header = hole_header;
       }
       // Put the new hole in the index;
       insert_ordered_array((void*)hole_header, &heap->index);
   }
```

If we wanted to split our hole in two, we do it here, creating a new hole.

```
   // ...And we're done!
   return (void *) ( (u32int)block_header+sizeof(header_t) );
}
```

... And that's our allocation function! The only thing left to do is fill in the error-checking code we missed out earlier:

```
   if (iterator == -1) // If we didn't find a suitable hole
   {
       // Save some previous data.
       u32int old_length = heap->end_address - heap->start_address;
       u32int old_end_address = heap->end_address;

       // We need to allocate some more space.
       expand(old_length+new_size, heap);
       u32int new_length = heap->end_address-heap->start_address;

       // Find the endmost header. (Not endmost in size, but in location).
       iterator = 0;
       // Vars to hold the index of, and value of, the endmost header found so far.
       u32int idx = -1; u32int value = 0x0;
       while (iterator < heap->index.size)
       {
           u32int tmp = (u32int)lookup_ordered_array(iterator, &heap->index);
           if (tmp > value)
           {
               value = tmp;
               idx = iterator;
           }
           iterator++;
       }

       // If we didn't find ANY headers, we need to add one.
       if (idx == -1)
       {
           header_t *header = (header_t *)old_end_address;
           header->magic = HEAP_MAGIC;
           header->size = new_length - old_length;
           header->is_hole = 1;
           footer_t *footer = (footer_t *) (old_end_address + header->size - sizeof(footer_t));
           footer->magic = HEAP_MAGIC;
           footer->header = header;
           insert_ordered_array((void*)header, &heap->index);
       }
       else
       {
           // The last header needs adjusting.
           header_t *header = lookup_ordered_array(idx, &heap->index);
           header->size += new_length - old_length;
           // Rewrite the footer.
           footer_t *footer = (footer_t *) ( (u32int)header + header->size - sizeof(footer_t) );
```

```
            footer->header = header;
            footer->magic = HEAP_MAGIC;
        }
        // We now have enough space. Recurse, and call the function again.
        return alloc(size, page_align, heap);
    }
```

This code quite simple in function but verbose in code. If a hole big enough couldn't be found (iterator == -1), we must expand the size of the heap (by calling the *expand* function). We then have to account for this expansion in the index. The normal way to do this is to find the endmost hole in the index and adjust it's size. The only time this won't work is when there aren't any holes in the index at all (an unlikely but possible case). In this case we must make one to fill the gap.

7.4.2.3. Freeing

Again, I'll go through this step-by-step.

```
void free(void *p, heap_t *heap)
{
   // Exit gracefully for null pointers.
   if (p == 0)
       return;

   // Get the header and footer associated with this pointer.
   header_t *header = (header_t*) ( (u32int)p - sizeof(header_t) );
   footer_t *footer = (footer_t*) ( (u32int)header + header->size - sizeof(footer_t) );

   // Sanity checks.
   ASSERT(header->magic == HEAP_MAGIC);
   ASSERT(footer->magic == HEAP_MAGIC);
```

Initially we find the header by subtracting sizeof(header_t) from *p*, then use this to find the footer. Sanity checks are always a good idea, as they provide an early indication if your code has overwritten crucial data.

```
   // Make us a hole.
   header->is_hole = 1;

   // Do we want to add this header into the 'free holes' index?
   char do_add = 1;
```

This block is being deallocated and so is now a hole. We also create a variable to hold whether we should add the header to the hole index (see algorithm description).

```
   // Unify left
   // If the thing immediately to the left of us is a footer...
   footer_t *test_footer = (footer_t*) ( (u32int)header - sizeof(footer_t) );
   if (test_footer->magic == HEAP_MAGIC &&
       test_footer->header->is_hole == 1)
   {
       u32int cache_size = header->size; // Cache our current size.
       header = test_footer->header;     // Rewrite our header with the new one.
       footer->header = header;          // Rewrite our footer to point to the new header.
       header->size += cache_size;       // Change the size.
       do_add = 0;                       // Since this header is already in the index, we don't want to add it
again.
   }
```

This piece of code performs our *left unification*. By subtracting sizeof(header_t) from the header address, we can get a pointer to a footer. We check if this is actually a valid footer by checking it's magic number, and that it is a hole (not allocated!). If so, we rewrite our footer to point to the test footer's header, change our size, and instruct the algorithm not to add an entry to the hole index (as the header we just unified with was already in the index!).

```
   // Unify right
   // If the thing immediately to the right of us is a header...
   header_t *test_header = (header_t*) ( (u32int)footer + sizeof(footer_t) );
   if (test_header->magic == HEAP_MAGIC &&
```

```
        test_header->is_hole)
    {
        header->size += test_header->size; // Increase our size.
        test_footer = (footer_t*) ( (u32int)test_header + // Rewrite it's footer to point to our header.
                                    test_header->size - sizeof(footer_t) );
        footer = test_footer;
        // Find and remove this header from the index.
        u32int iterator = 0;
        while ( (iterator < heap->index.size) &&
                (lookup_ordered_array(iterator, &heap->index) != (void*)test_header) )
            iterator++;

        // Make sure we actually found the item.
        ASSERT(iterator < heap->index.size);
        // Remove it.
        remove_ordered_array(iterator, &heap->index);
    }
```

Similarly, this code performs our *right unification*. Again, we test if the header immediately to our right is valid, and is a hole. We rewrite it's footer to point to our header, then remove it's header from the hole index.

```
    // If the footer location is the end address, we can contract.
    if ( (u32int)footer+sizeof(footer_t) == heap->end_address)
    {
        u32int old_length = heap->end_address-heap->start_address;
        u32int new_length = contract( (u32int)header - heap->start_address, heap);
        // Check how big we will be after resizing.
        if (header->size - (old_length-new_length) > 0)
        {
            // We will still exist, so resize us.
            header->size -= old_length-new_length;
            footer = (footer_t*) ( (u32int)header + header->size - sizeof(footer_t) );
            footer->magic = HEAP_MAGIC;
            footer->header = header;
        }
        else
        {
            // We will no longer exist :(. Remove us from the index.
            u32int iterator = 0;
            while ( (iterator < heap->index.size) &&
                    (lookup_ordered_array(iterator, &heap->index) != (void*)test_header) )
                iterator++;
            // If we didn't find ourselves, we have nothing to remove.
            if (iterator < heap->index.size)
                remove_ordered_array(iterator, &heap->index);
        }
    }
```

This is almost the last snippet (I promise!). If we are releasing the last hole in the index (that is, the one closest to the end of memory), then we can contract the heap size. We keep note of the old heap size, then call contract. One of two things can happen here. Either the contract() command will shrink the heap so our hole no longer exists (the 'else' case), or it will either partially contract or not contract at all. In which case our hole still exists, but we need to resize it. We rewrite it's footer with the new size, and exit. If the hole has been removed, we just look ourselves up in the heap index and delete ourselves.

A small one to finish off with:

```
if (do_add == 1)
  insert_ordered_array((void*) header, &heap->index);
```

If we are suppposed to add ourselves into the index, do it here. And that's it! The next thing to do is initialise the heap when paging is initialised :)

7.4.2.4. paging.c

```
extern heap_t *kheap;
```

We declare the variable *kheap* as our kernel heap. We define this in kheap.c (you can do that yourself) and reference it here.

```
// Map some pages in the kernel heap area.
// Here we call get_page but not alloc_frame. This causes page_table_t's
// to be created where necessary. We can't allocate frames yet because they
// they need to be identity mapped first below, and yet we can't increase
// placement_address between identity mapping and enabling the heap!
int i = 0;
for (i = KHEAP_START; i < KHEAP_START+KHEAP_INITIAL_SIZE; i += 0x1000)
    get_page(i, 1, kernel_directory);
```

This goes in *initialise_paging*, before we identity map from 0-placement_addr. There is a reason for this code. As the kernel heap is up at 0xC0000000, when we write to it some page tables will need to be created (beacause nothing near that area has been accessed before). However, after we finish the identity-mapping loop allocating everything up to placement_address, we can't use kmalloc any more until our heap is active! So, we need to force the tables to be created **before** we freeze the placement address. That's what this code does.

```
// Now allocate those pages we mapped earlier.
for (i = KHEAP_START; i < KHEAP_START+KHEAP_INITIAL_SIZE; i += 0x1000)
    alloc_frame( get_page(i, 1, kernel_directory), 0, 0);

// Before we enable paging, we must register our page fault handler.
register_interrupt_handler(14, page_fault);

// Now, enable paging!
switch_page_directory(kernel_directory);

// Initialise the kernel heap.
kheap = create_heap(KHEAP_START, KHEAP_START+KHEAP_INITIAL_SIZE, 0xCFFFF000, 0, 0);
```

Et voila! you are complete! A nice thing to do (which I have done in my sample code) is to get kmalloc/kfree to pass calls straight through to alloc/free if kheap != 0. I'll leave that to you to do ;)

```

```

7.5. Testing

main.c

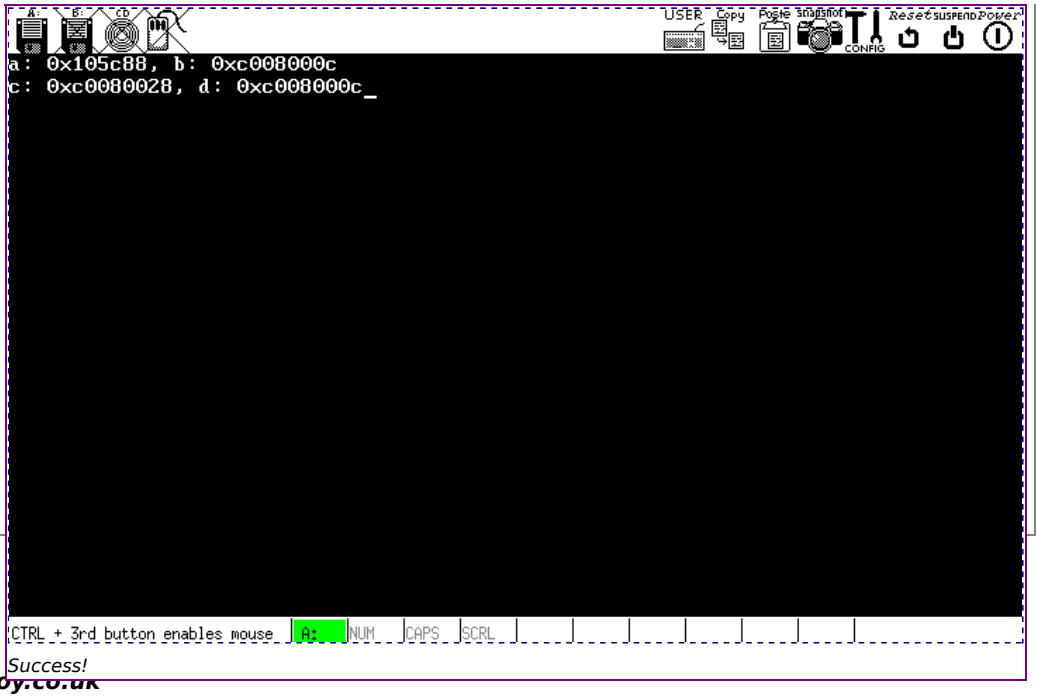
```
u32int a = kmalloc(8);
initialise_paging();
u32int b = kmalloc(8);
u32int c = kmalloc(8);
monitor_write("a: ");
monitor_write_hex(a);
monitor_write(", b: ");
monitor_write_hex(b);
monitor_write("\nc: ");
monitor_write_hex(c);

kfree(c);
kfree(b);
u32int d = kmalloc(12);
monitor_write(", d: ");
monitor_write_hex(d);
```

You can, of course, experiment with the order of allocations and frees here. The code above will allocate one variable, *a*, before initialise_paging is called, so it'll be allocated via placement address. *b* and *c* both get allocated on the heap, and printed out. They are then both freed and another variable, *d*, created. If the address of *d* is the same as the address of *b*, then the space reclaimed by *b* and *c* has been successfully unified and all is good!

7.6. Summary

Dynamic memory allocation is one of the few things that it is very difficult to do without. Without it, you would have to specify an absolute maximum number of processes running (static array of pids), you would have to statically give the size of every buffer - Generally making your OS lacklustre and woefully inefficient.

Sample code, as ever, can be found here.



james<at>jamesmolloy.co.uk