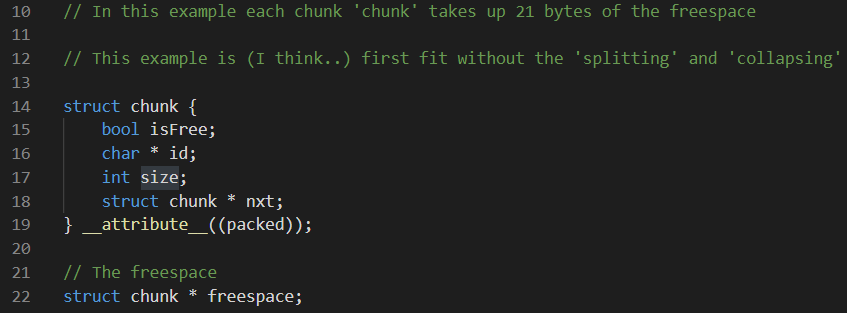
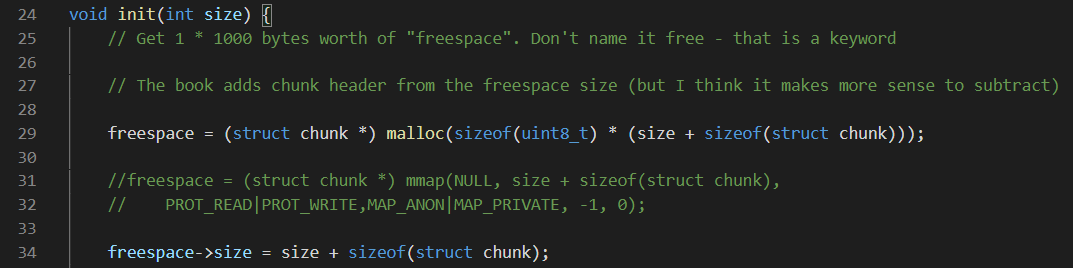
Walkthrough of **freespace.c**

Refer to C:\Users\Alifa\Desktop\**cygwin64-workspace**\mystuff\ostep\homework\freespace\**freespace.c**





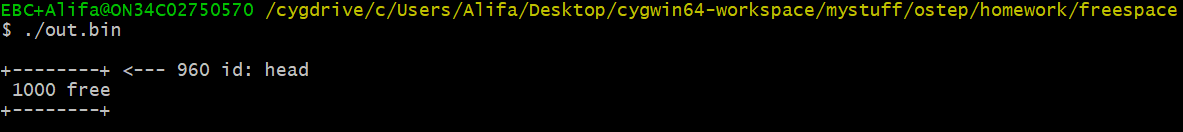
Notice I’m not using **mmap** function. TBH it won’t be available when writing up an OS anyway. I tested this will the following values.



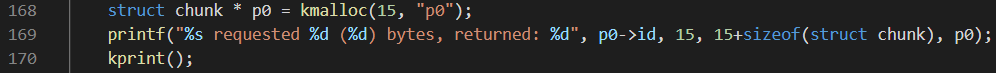
The first step is to initialize the freespace



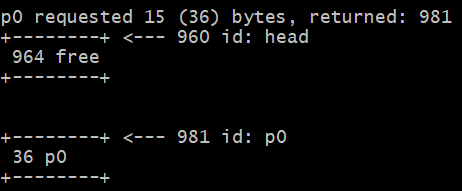
Where we get:



P0 requests 15 bytes (plus header of 21 bytes which we add to this)



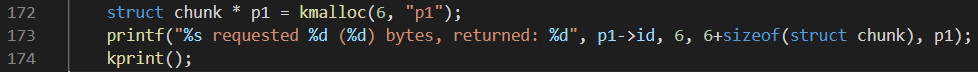
Which gives us:

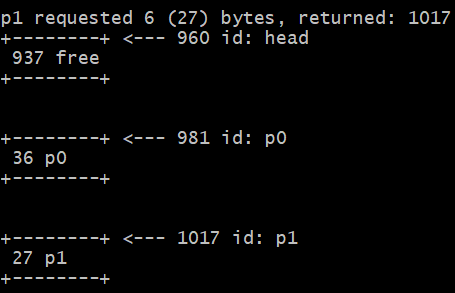


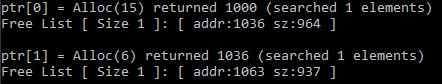
…and here is what OSTEP malloc.py returns for the same request



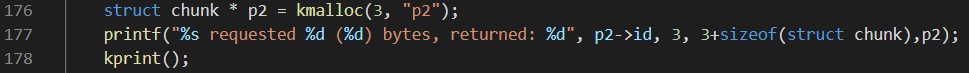
Next P1 asks for 6 bytes

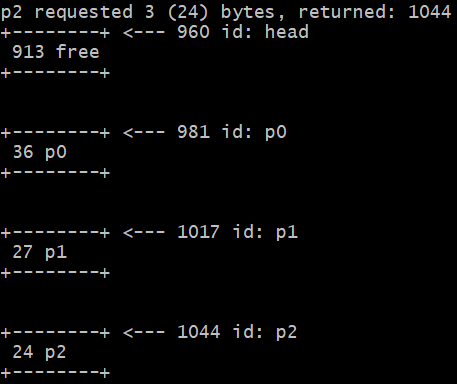


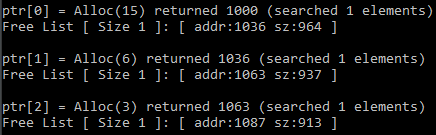




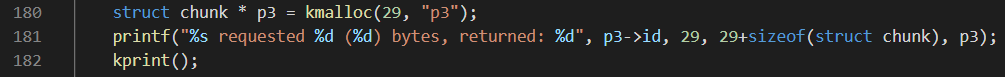
P2 asks for 3 bytes

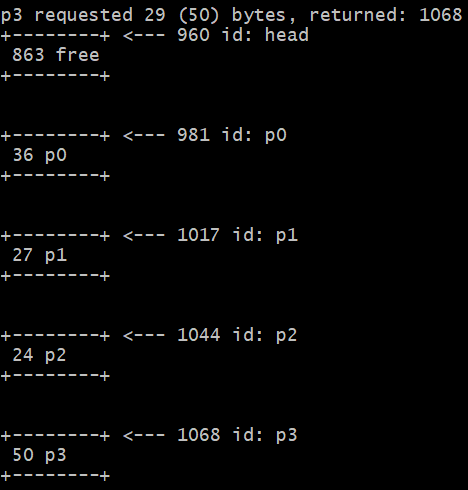


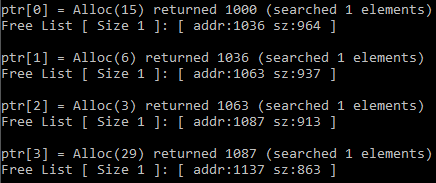




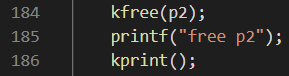
P3 asks for 29 bytes

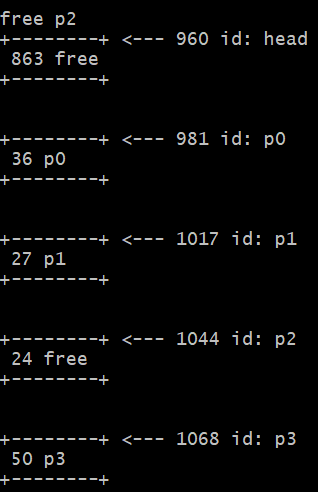


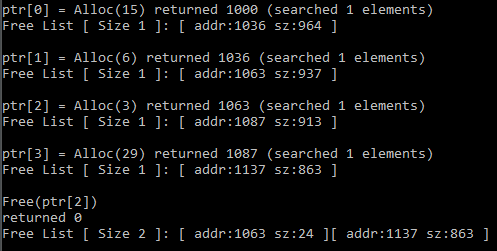




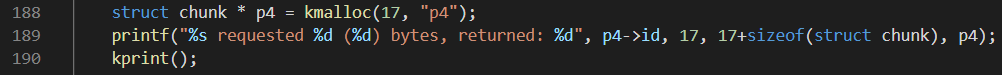
Next, we free p2

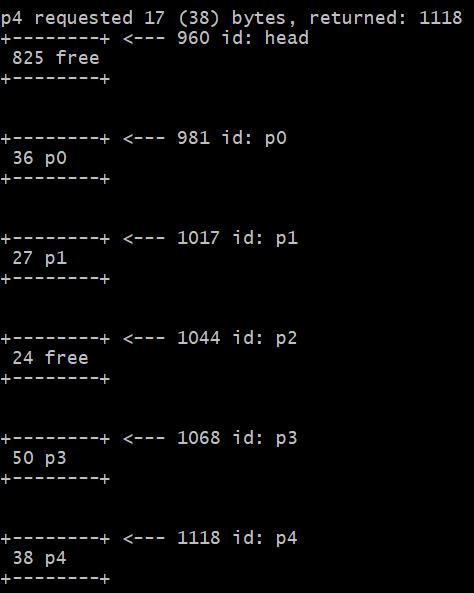


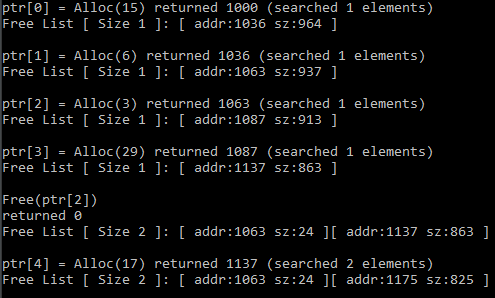




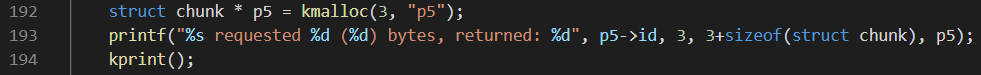
P4 asks for 17 bytes

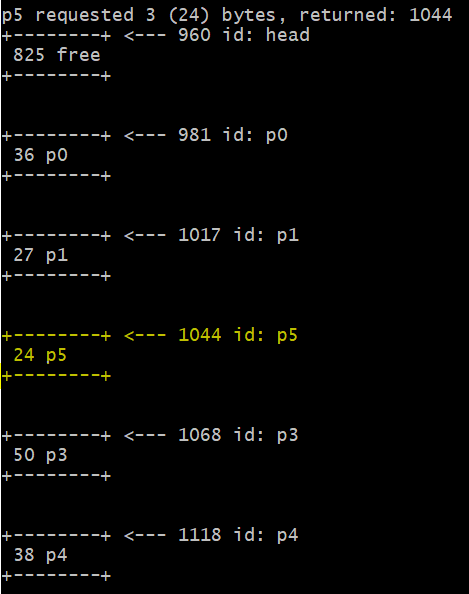


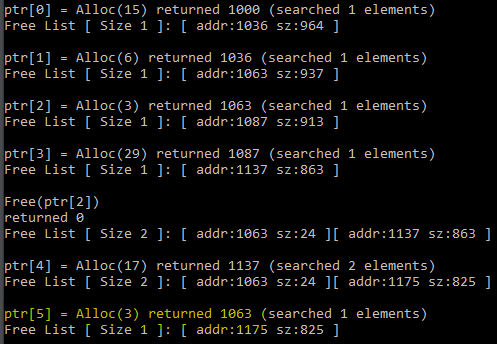




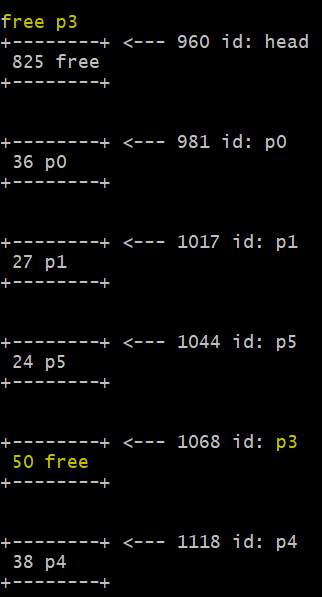
P5 requests for and gets 3 bytes that were previously occupied by p2

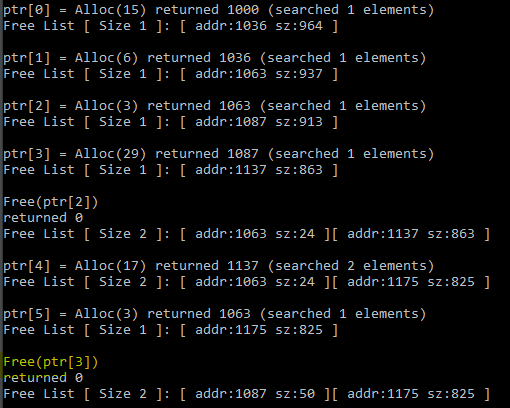




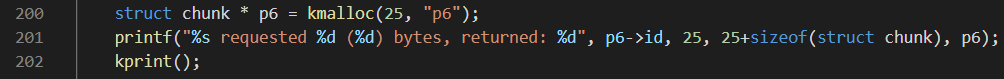


We free p3



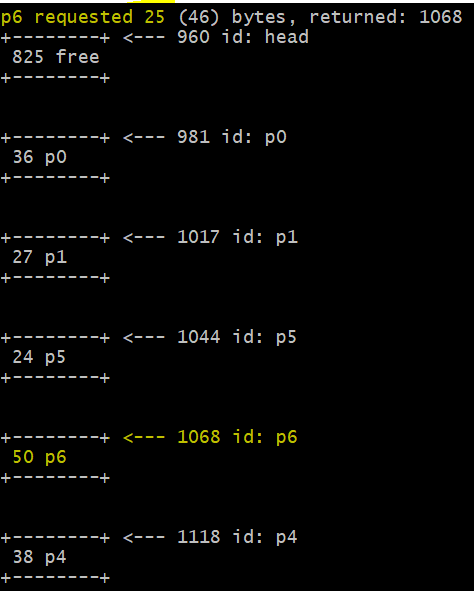


P6 requests 25 bytes. **Watch what happens here in my implementation**.

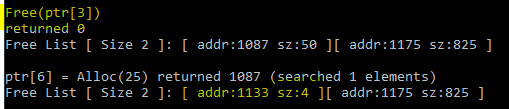


Remember we freed **p3** which had 29 bytes worth of space. P6 needs just 25 bytes so m implementation returns the same address that was used for p3. That leaves us with extra **4 bytes** that is “wasted” (29 – 25 = 4). Why the 4 bytes are wasted? **Because P6 gets the same space as P3**.

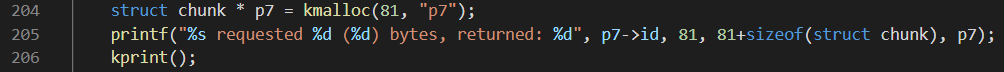


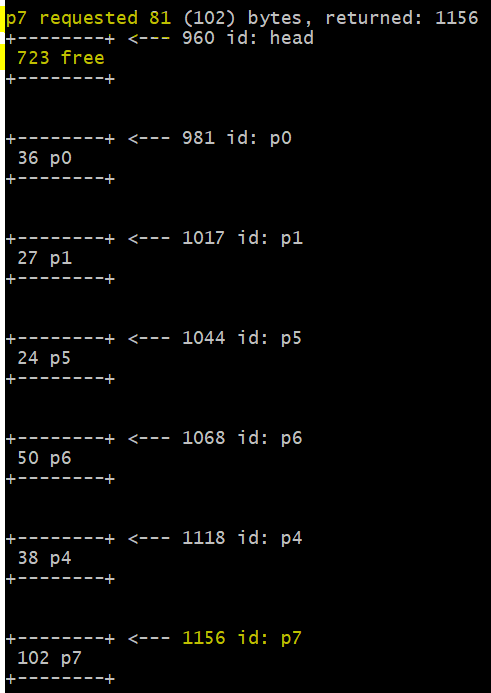


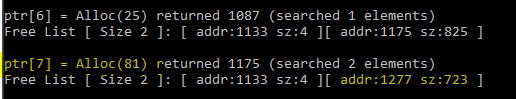
However, in OSTEP malloc.py ***splits*** the p3 chunk into two chunks: one with 25 bytes (for p6) and one with 4 bytes (**as free**). The problem I have with this approach is that those 4 bytes worth of free space are smaller than the header (21 bytes!) which means these 4 bytes will never get allocated 😊



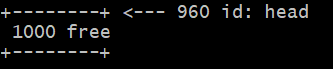
And finally, p7 asks for 81 bytes







You may have noticed that in my C implementation the head (large free chunk) maintains the same address each time a new chunk is requested. That is because the new allocated chunk is placed **below** the large free space upon each request, however, in the OSTEP book (chapter on ‘freespace’ page 7) you can see that when a new chunk is requested the location of the large free space is **moved** down so that the newly allocated chunk is placed **above** the free space (my **Java** implementation does this too).



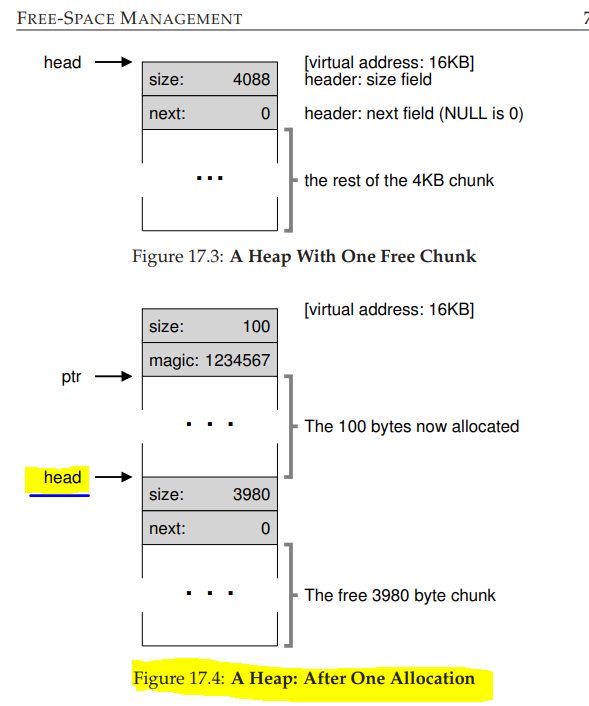
But I think their approach is incorrect because you can’t really move chunks of memory (bytes) within an entire free space. That would mean ***shifting*** bytes (and pointers) and based on your experience with osdev – qemu/bochs – this was a bad, bad idea!

Finally, the **qemu implementation**: Note that because the header size is different here, I had to use different set of params to match the output with malloc.py:

Dos>python malloc.py -S 1000 -b **1048576** -H **13** -p FIRST -s 0 -A +15,+6,+3,+29,-2,+17,+3,-3,+25,+81 -c

See C:\Users\Alifa\Desktop\osdev\batch\kernel\heap\kernelostep.c





4088 – 100 – 8 = 3980

“After running this code, the status of the list is that it has a single entry, of size 4088…(Fig 17.3). Now let’s imagine that a chunk of memory is requested, say of size 100 bytes. To service this request, the library will first find a chunk that is large enough to accommodate the request; because there is only one free chunk (size: 4088), this chunk will be chosen. \***Then, the chunk will be split into two: one chunk big enough to service the request (and header, as described above), and the remaining free chunk**. Assuming an 8-byte header (an integer size and an integer magic number), the space in the heap now looks like what you see in Figure 17.4.”

\*That’s exactly what my C implementation is doing except it does not shift the head/free space downwards.