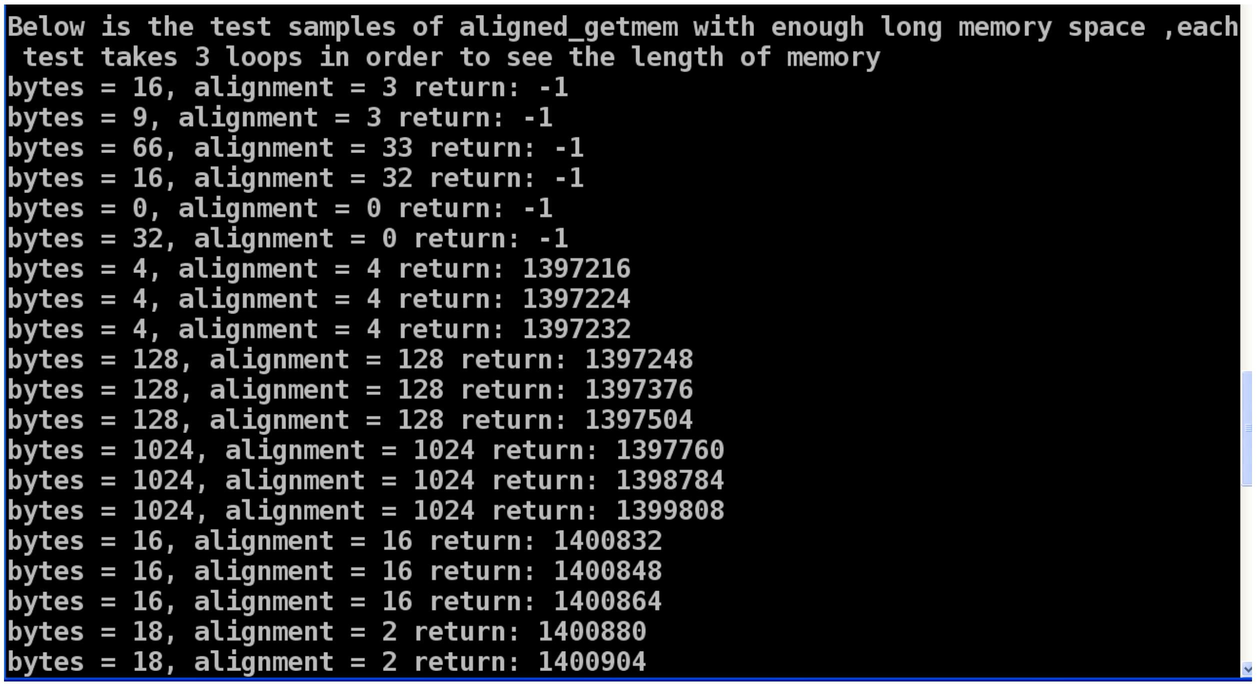
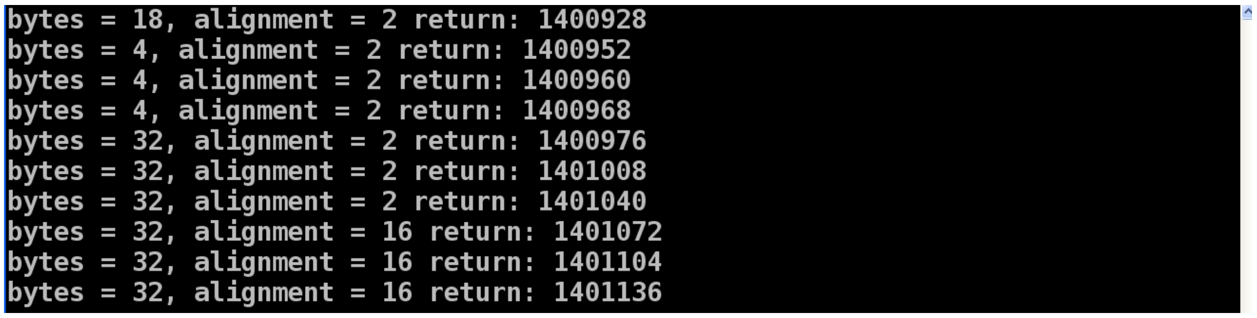
- How does your solution ensure allocation of memory in an aligned manner?

  I round up the memory address to the nearest aligned manner address. This may generate fragments for the first address, but will not generate more fragments after that.

- Describe your test cases. How to they ensure that your code correctly meets the requirements?



```
Below is the test samples of aligned_getmem with enough long memory space ,each
 test takes 3 loops in order to see the length of memory
bytes = 16, alignment = 3 return: -1
bytes = 9, alignment = 3 return: -1
bytes = 66, alignment = 33 return: -1
bytes = 16, alignment = 32 return: -1
bytes = 0, alignment = 0 return: -1
bytes = 32, alignment = 0 return: -1
bytes = 4, alignment = 4 return: 1397216
bytes = 4, alignment = 4 return: 1397224
bytes = 4, alignment = 4 return: 1397232
bytes = 128, alignment = 128 return: 1397248
bytes = 128, alignment = 128 return: 1397376
bytes = 128, alignment = 128 return: 1397504
bytes = 1024, alignment = 1024 return: 1397760
bytes = 1024, alignment = 1024 return: 1398784
bytes = 1024, alignment = 1024 return: 1399808
bytes = 16, alignment = 16 return: 1400832
bytes = 16, alignment = 16 return: 1400848
bytes = 16, alignment = 16 return: 1400864
bytes = 18, alignment = 2 return: 1400880
bytes = 18, alignment = 2 return: 1400904
```



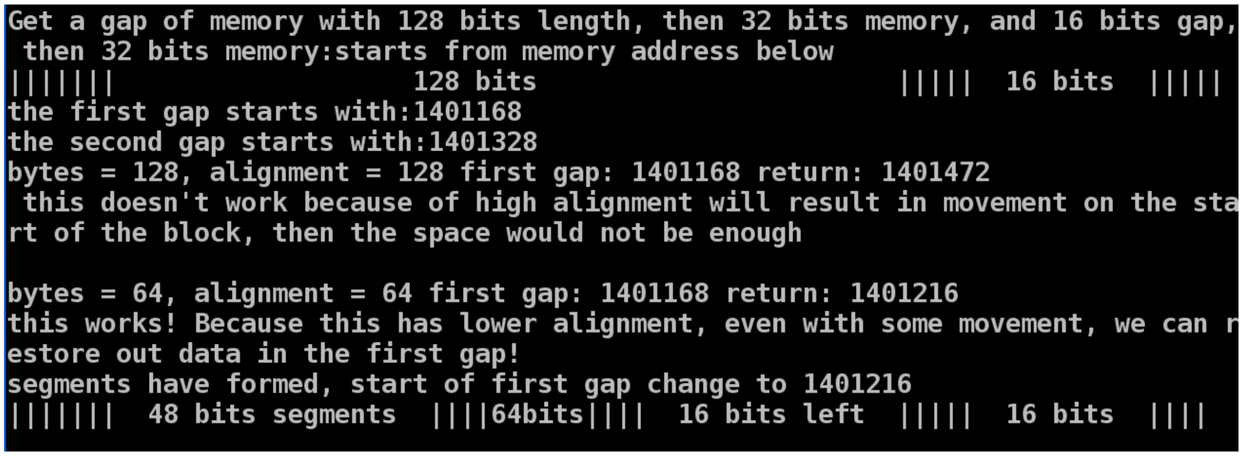
```
bytes = 18, alignment = 2 return: 1400928
bytes = 4, alignment = 2 return: 1400952
bytes = 4, alignment = 2 return: 1400960
bytes = 4, alignment = 2 return: 1400968
bytes = 32, alignment = 2 return: 1400976
bytes = 32, alignment = 2 return: 1401008
bytes = 32, alignment = 2 return: 1401040
bytes = 32, alignment = 16 return: 1401072
bytes = 32, alignment = 16 return: 1401104
bytes = 32, alignment = 16 return: 1401136
```

1. I generate 3 loops for each test cases to see clearly how long each memory address is. I used different combinations of bytes and alignment. When alignment is not order of 2 or bytes not multiple of alignment, it will return error. Alignment cannot be 0, it will also return error in this case.

   If alignment or bytes is less than 8, it will round up to be 8.

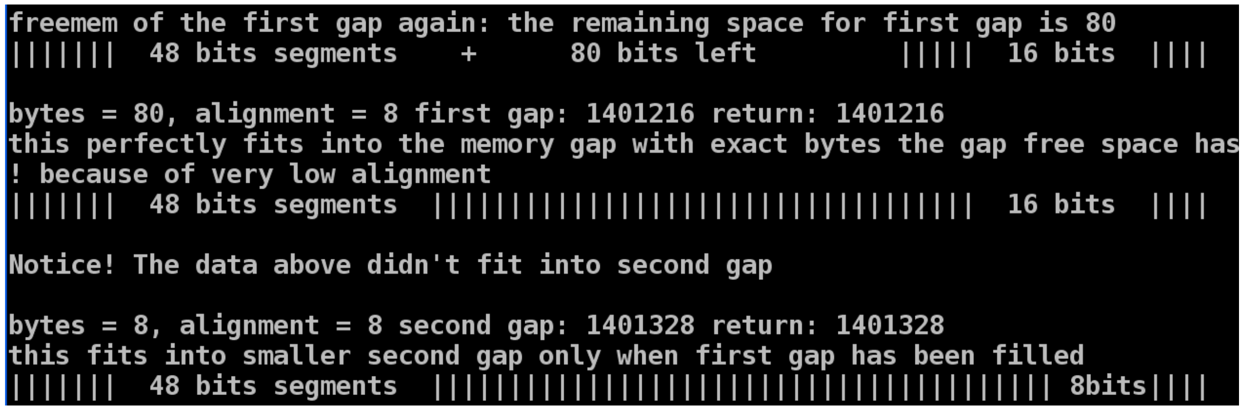
If alignment is very large, the address is not multiple of alignment, the address will round up to be the right address and leave a fragment here.

More details you can see tests above, the return value is the address pointer returned.



```
Get a gap of memory with 128 bits length, then 32 bits memory, and 16 bits gap,
 then 32 bits memory:starts from memory address below
|||||||                   128 bits                       |||||  16 bits  |||||
the first gap starts with:1401168
the second gap starts with:1401328
bytes = 128, alignment = 128 first gap: 1401168 return: 1401472
 this doesn't work because of high alignment will result in movement on the sta
rt of the block, then the space would not be enough

bytes = 64, alignment = 64 first gap: 1401168 return: 1401216
this works! Because this has lower alignment, even with some movement, we can r
estore out data in the first gap!
segments have formed, start of first gap change to 1401216
|||||||  48 bits segments  ||||64bits||||  16 bits left  |||||  16 bits  ||||
```



```
freemem of the first gap again: the remaining space for first gap is 80
|||||||  48 bits segments    +      80 bits left         |||||  16 bits  ||||

bytes = 80, alignment = 8 first gap: 1401216 return: 1401216
this perfectly fits into the memory gap with exact bytes the gap free space has
! because of very low alignment
|||||||  48 bits segments  ||||||||||||||||||||||||||||||||||  16 bits  ||||

Notice! The data above didn't fit into second gap

bytes = 8, alignment = 8 second gap: 1401328 return: 1401328
this fits into smaller second gap only when first gap has been filled
|||||||  48 bits segments  |||||||||||||||||||||||||||||||||||||||| 8bits||||
```

2. Above is the tests for allocate memory in a small gap, it is clearly illustrated in the figures. Firstly I initialized several alignment data larger than 16 bits to fit into the larger gap with 128 bits. The 128 alignment data doesn't fit into this gap because you will have to round up to the address and it will move the address and decrease the length of available free space, however 64 bits fit this gap perfectly.
   Then we delete the 64 bits data and the pointer is still at the 64 bits data next address, then we insert a 80 bits data, it perfectly fits into the gap, since the address is perfectly a multiple number of the alignment.
   Finally we use a little length data to fits into the second gap which has not been fitted by the data before.

- Does it make sense to allow an *alignment* value that is not a power of 2, of so why if not why not?

  I don't think it will work if the alignment is not a power of 2. For example if the alignment is a power of 3, then we have to round up every time if we use alignment_getmem and getmem together, and it will generate large piles of fragments in the memory. If we have to do like this, we have to modify the getmem too, which seems like complicated. Also this do not fits the bit operations, which we can easily find the power of 2 and may need more calculations to find power of 3. It’s also will lead to conflict to the hardware which belongs to a binary system and you use a 3 power or other number systems.

- The getmem system call is very similar to the POSIX malloc system call. In POSIX, there is a corresponding free system call which performs a very similar operation to the freemem system call in XINU, however the freemem system call in XINU requires the caller to specify not only the memory address to free, but also the size of memory to free (take a look at system/freemem.c). The POSIX free does not require the caller to specify the size (only the address). Why does XINU require the size to be specified when freemem is called? What modifications would need to be made to XINU to not require the caller to specify a size when the call freemem?

  I think the XINU do the coalescing work using the size of the data, when you free a memory you have to do coalescing and increase the size of the previous memory or next memory. Since the pervious or next memory length has been changed, this would help to find out whether the next freed memory is coalescing the surrounding memory and do a better work on freeing them. In a word, this is a good constraint of avoiding the freemem function to generate fragments in the memory.