Dynamic memory allocator homework report

SHENGXIN QIAN

1. The General thinking about realizing my own malloc() and free() function

Use system call function ***void *sbrk(intptr_t increment)*** to make the heap space accessible and use pre-defined structure to manage every users' malloce space. In every pre-defined structure, we would store metadata like size of block, the flag of whether this structure is free or not and we would use free block linked list to manage every free block which would make the block allocation process faster. When our the heap space we apply for is not enough, we could call sbrk() again to let OS allocate more heap space.

2. Different choices when we realizing our allocator

2.1. Alignment Problem

When we apply for 4 bytes block, the most UNIX system would actually return a larger memory block and the size of it would be a multiple of some fixed number (the power of 2). This is called alignment. The benefit of it is make the program more compatible for different platform and easy for split a larger memory block. And also, could be accelerated by some structure in CPU Data path (could use shift for multiple).

2.2. How to organize our metadata? (32bit)

We could use 4B at front (head part) to store the size and flag of whether this block is used or not. We could use the next 4B to store the free list pointer which points to the previous free block and next 4B to store the pointer pointing to the next free block. At the end of block, we could still use 4B as foot part to store the same information as the head part. The foot part is for coalescing. But all those 4 parts of metadata is for free block. **When we allocate this block of memory, we could erase the foot part and two pointers.** We only need to keep the front part for tracking. It could decrease the number of internal fragments.

2.3. How to manage free block

Of course we could use implicit allocator to manage all block. Which means the list for locating free block by traverse every block with real heap address. This would be O(n) complexity search, the n is the size of all blocks. This could make our internal fragment minimal but the time cost is too much. That is why we trade some space for time. We could store two more pointers to link our free block. Which will make our malloc process become O(nf) complexity search, the nf is the size of free blocks. The tradeoff is the internal fragmentation problem.

2.4. How to choose the free block when we go through the free lists

There are many optional block in our free linked list to return, we could choose the first suitable block (first fit), the smallest optional block (best fit), the largest

suitable block (worst fit) and the last time we split (next fit). Different placement policy could make different allocation time and utilize ratio.

2.5. How to coalesce two free block which is side by side.

We could create the metadata like foot part and head part to track the previous and next block on heap and check whether it is free or not.

2.6. How to split a large memory block for small application?

Just split the large memory block and reinsert the rest free block

3. The diagram of my realization of dynamic allocator (only compatible to 32bit)

3.1. Some general information and term explanation

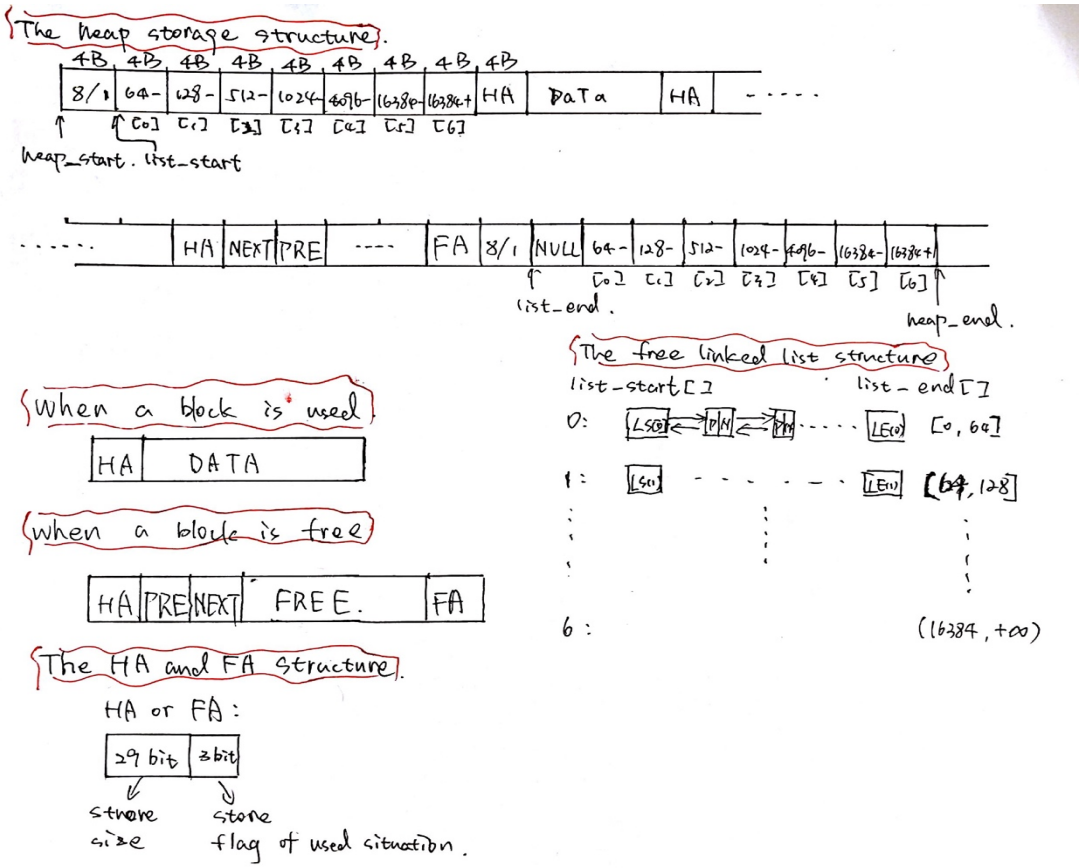
3.1.1. Compile requirement

The test case makefile must add –m32 at CFLAGS part.

3.1.2. Some possible question

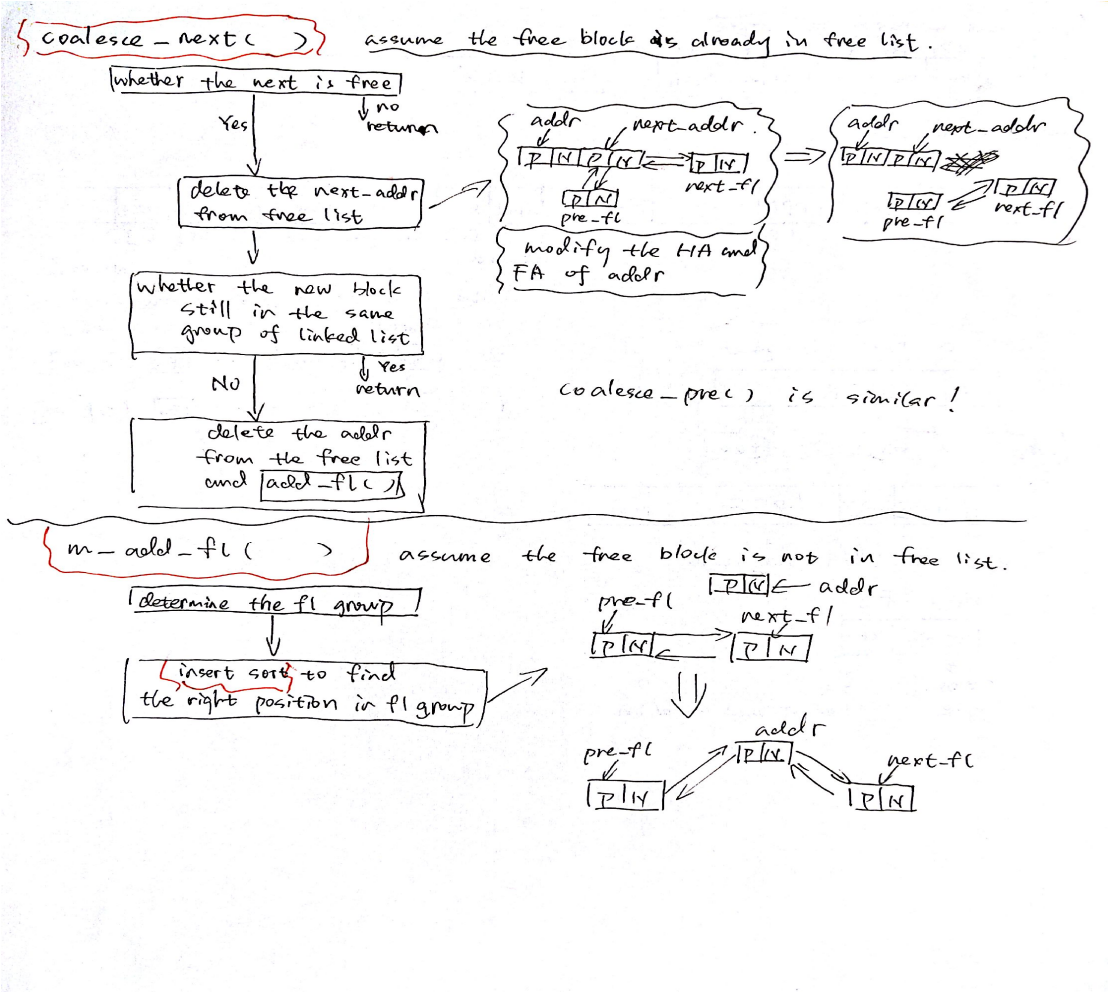
When running large test with worst fit, the program may be killed. But rerun the program could make it work.

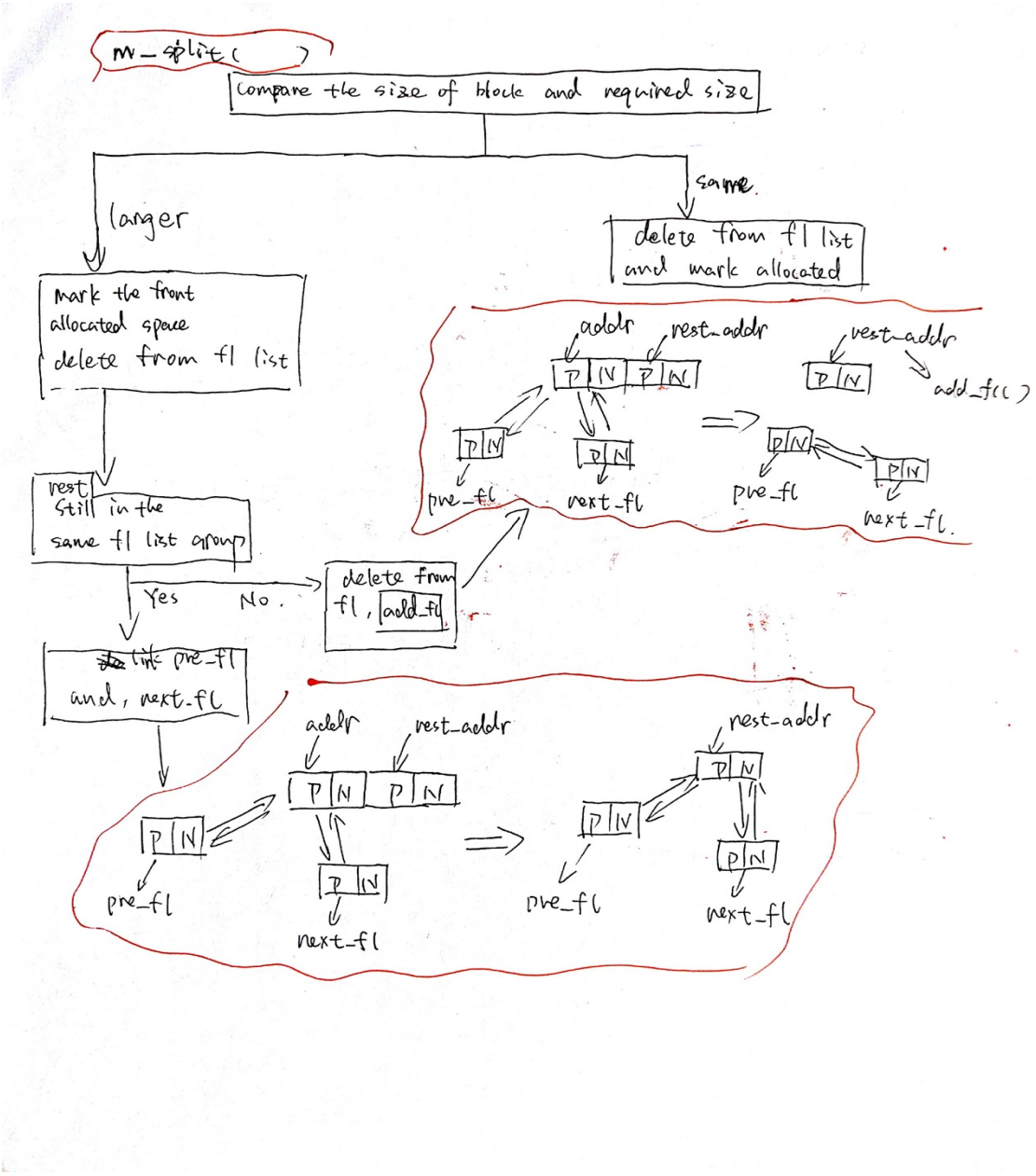
3.2. The general heap model



As the image showed above, the whole heap has same pre-defined setting to help manage the whole memory space. The heap_start is the start address of dynamic allocated space we use sbrk() create. The heap_end is the end address of the memory space we manager. The list_start is the multiple free list start address. The list_start+i*4 is the start pointer of the i th free list. As the same, the list_end+(i+1)*4 is the end pointer of the i th free list. The "+1" is compatible for the expression of locating previous pointer and next pointer in the regular block. Each free list has one start and one end, would not need use NULL to check the boundary. **The standard pointer to locate one block is the end of HA part. All the address to locate one block in the interface of each function is the standard pointer.** The standard used block structure is showed as above. In the used block, only the HA part is kept for tracking. In the standard free block structure, we need keep the HA part, PRE part (previous free list pointer), NEXT part (next free list pointer) and FA part. That is why we need 16 B alignment. Because the worst case we still need 16 B to track free block. We have 7 free lists for store the block which size belongs to [0, 64], (64, 128), (128, 512), (512, 1024), (1024, 4096), (4096, 16384), (16384, infinite). The block in each free list was ascendingly sorted. If one free list is empty, the LS(i) will directly link to LE(i). **Because the free list is ascendingly sorted, the best fit is actually first fit.**

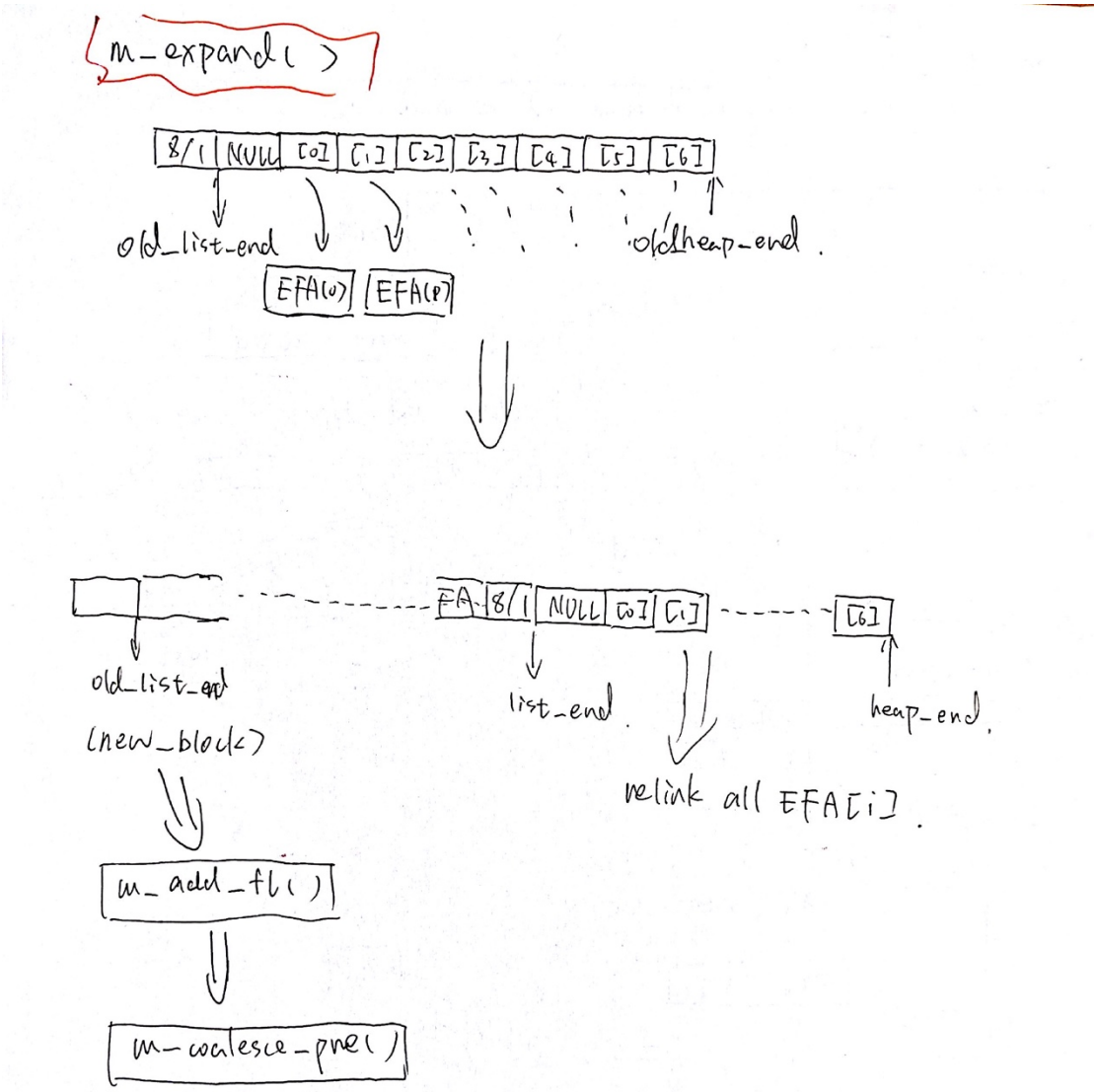
3.3. Diagrams of several important functions





The picture above shows the details of **m_coalesce_next()** and **m_add_fl()**. The **m_coalesce_pre()** is similar to **m_coalesce_next()**. There is a important difference is that because the HA part will not be erase when the block was erased, we can check the next block through the flag bit in the HA part. That is why when we check whether we could coalesce the previous block, we need to check 1. The flag bit is free at FA part. 2. When locate standard pointer, this pointer should belong to [heap_start, heap_end]. 3. The HA part located by standard pointer is same as its FA part. The **m_add_fl()** function insert a unlinked free block via insert sort method. This is best solution when we just add one element into a sorted list (O(n)).

The **m_split()** function split one large free block into one used block and one free block. We need reinsert the rest free block into free list.



The **m_expand()** function expand the whole heap when there is no suitable free block in heap. The linked list end at the end of the heap need to be relocate and relink to the multiple linked list.

4. Final program test result and analysis

4.1. The FF version result

```
parallels@ubuntu:~/ece650/Malloc/Finaltest/test$ make
gcc -O3 -fPIC -m32  -I../ -L../ -DFF -Wl,-rpath=../ -o equal_size_allocs equal_size_allocs.c -lmymalloc -lrt
gcc -O3 -fPIC -m32  -I../ -L../ -DFF -Wl,-rpath=../ -o small_range_rand_allocs small_range_rand_allocs.c -lmymalloc -l
gcc -O3 -fPIC -m32  -I../ -L../ -DFF -Wl,-rpath=../ -o large_range_rand_allocs large_range_rand_allocs.c -lmymalloc -l
parallels@ubuntu:~/ece650/Malloc/Finaltest/test$ ./equal_size_allocs
Execution Time = 3.736918 seconds
Fragmentation  = 0.449989
parallels@ubuntu:~/ece650/Malloc/Finaltest/test$ ./small_range_rand_allocs
data_segment_size = 3492580, data_segment_free_space = 152800
Execution Time = 2.429294 seconds
Fragmentation  = 0.043750
parallels@ubuntu:~/ece650/Malloc/Finaltest/test$ ./large_range_rand_allocs
Execution Time = 8.439659 seconds
Fragmentation  = 0.065744
parallels@ubuntu:~/ece650/Malloc/Finaltest/test$
```

This result is test with equal(10000*10000), small(100*10000), large(50*10000).

4.2. The BF version result

```
parallels@ubuntu:~/ece650/Malloc/Finaltest/test$ make
gcc -O3 -fPIC -m32  -I../ -L../ -DBF -Wl,-rpath=../ -o equal_size_allocs equal_size_allocs.c -lmymalloc -lrt
gcc -O3 -fPIC -m32  -I../ -L../ -DBF -Wl,-rpath=../ -o small_range_rand_allocs small_range_rand_allocs.c -lmymalloc -lrt
gcc -O3 -fPIC -m32  -I../ -L../ -DBF -Wl,-rpath=../ -o large_range_rand_allocs large_range_rand_allocs.c -lmymalloc -lrt
parallels@ubuntu:~/ece650/Malloc/Finaltest/test$ ./equal_size_allocs
Execution Time = 3.942347 seconds
Fragmentation  = 0.449989
parallels@ubuntu:~/ece650/Malloc/Finaltest/test$ ./small_range_rand_allocs
data_segment_size = 3492580, data_segment_free_space = 152800
Execution Time = 2.372695 seconds
Fragmentation  = 0.043750
parallels@ubuntu:~/ece650/Malloc/Finaltest/test$ ./large_range_rand_allocs
Execution Time = 8.262055 seconds
Fragmentation  = 0.065744
parallels@ubuntu:~/ece650/Malloc/Finaltest/test$
```

This result is test with equal(10000*10000), small(100*10000), large(50*10000). Because the free list is ascendingly sorted the BF is equal to FF.

4.3. The WF version result

```
parallels@ubuntu:~/ece650/Malloc/Finaltest/test$ make
gcc -O3 -fPIC -m32  -I../ -L../ -DWF -Wl,-rpath=../ -o equal_size_allocs equal_size_allocs.c -lmymalloc -lrt
gcc -O3 -fPIC -m32  -I../ -L../ -DWF -Wl,-rpath=../ -o small_range_rand_allocs small_range_rand_allocs.c -lmymalloc -lrt
gcc -O3 -fPIC -m32  -I../ -L../ -DWF -Wl,-rpath=../ -o large_range_rand_allocs large_range_rand_allocs.c -lmymalloc -lrt
parallels@ubuntu:~/ece650/Malloc/Finaltest/test$ ./equal_size_allocs
Execution Time = 5.395057 seconds
Fragmentation  = 0.449989
parallels@ubuntu:~/ece650/Malloc/Finaltest/test$ ./small_range_rand_allocs
data_segment_size = 5132260, data_segment_free_space = 1792480
Execution Time = 0.921896 seconds
Fragmentation  = 0.349257
parallels@ubuntu:~/ece650/Malloc/Finaltest/test$ ./large_range_rand_allocs
Killed
parallels@ubuntu:~/ece650/Malloc/Finaltest/test$ ./large_range_rand_allocs
Execution Time = 9.342993 seconds
Fragmentation  = 0.915318
parallels@ubuntu:~/ece650/Malloc/Finaltest/test$
```

This is test with equal(10000*10000), small(100*10000), large(50*10000). The WF is surprisingly fast. The larget test was killed at first. Rerun could make it work. But the fragmentation is very disappointing.

4.4. Analysis on the result

The different on three version of code is just the free block search policy. The WF is just go through from the end of last free list and keep searching to the front. When we first meet the larger free block, we split it and allocate. When still could not find the free block larger than required, expand. The BF is just search from the start require size's free list. Because it is sorted, we could just stop when we first meet the larger block. In my program, the FF is same as BF.

The equal test would cause some kind of shake (when coalesce one block and then split again). This is very significant delay running time when the free list is not sorted and use LIFO policy to add new free block. But after I sorted the free list and multiple free list policy. The shake could relieve. But the best solution for that is used delay coalesce policy. In my program, I use immediate coalescing policy which is not a good option for ultimate version. Just because the block is equal, the fragmentation problem is similar in FF, BF, WF version code.

FF:

Because I used sorted free list, the first and best fit version is actually same thing.

But when I use unsorted list, the first fit with delayed coalescing policy could work very fast on equal test case. This is because each block has similar size. I could achieve only one second with delayed coalescing policy but first fit would create more fragments because it split blocks more often than best fit. Because it need totally different type of free list and unsorted free list works really bad on BF and WF. More than that, it is not good to create an independent unsorted free list for ff version code. So, I give up the unsorted free list version code. After all, for first fit, speed is a good part but fragmentation problem is worse.

BF:
The best fit could achieve a balance between the fragmentation and speed. It could reduce fragmentation because it only chooses the smallest fit block. Most of time, it would choose the same size block as long as the malloc has some kind of pattern. But if you want to run BF really fast, you need use complex structure to store free block like multiple free list.

WF:
It is obviously, WF would create more fragments than the others. Because it could always split small block and keep a lot middle size free blocks which would make more middle size fragments when we only need large blocks. That is why the fragment keep growing when face large test. But the tricky thing is it would super fast on small test. So, WF only works on small test when we want to focus on speed performance.