Performance Analysis

Advantages and Disadvantages of my choices:

There are many advantages and disadvantages to using a one way linked list or a two way linked list. My main reason I chose a one way linked list was because I did not want to mess with the structure design and the size of my structure. I also already knew how to implement a simple function that could find the previous block to any particular block of memory. If I used a two way linked list however, using the previous option would have been a lot simpler and would have allowed me to now type as much code and logic. I believe it would be more efficient to use a two way linked list because you do not have to traverse through the list to find the previous like I did in my find_prev function. Another option was to keep a global previous struc_t pointer.

The advantage to first fit is that it takes less time to really find a block of memory that the computer could give to the user. However just any block of memory that can satisfy the parameters for the amount of bytes does not necessarily make it the best fit. That is why this is the disadvantage because it could create more fragmentation in random parts of the free list. The advantage to best fit is that it traverses the list looking for the closest block of memory to the parameters and could even find an exact match that could create no fragmentation. The disadvantage is that it has to traverse through the entire list which takes time and most of the time it finds close matches and then shaves off of it to create a very small block of memory in the free list. Thus creating a lot of fragmentation. The advantage of Worst Sort is that it creates the least fragmentation of all the sorts because it always returns the biggest memory block available in the free list. An optional implementation that I put into my code was that if the worst sort found an exact match then it would immediately return that block. Since the point of the worst sort is to create the least fragmentation as possible this option is very essential in both time and the purpose. A disadvantage would be that worst sort has to usually traverse through the whole list and takes a lot of time.

Coalescing fixes the issues with fragmentation and run time. Once a memory is freed and its next and previous blocks in memory are also available in the free list, coalescing allows all three blocks to combine into one. It can also work with just the prev block or the next block alone. Coalescing allows the memory to become a larger memory so that fragmentation is less likely and the sorts have a much larger memory block to chose from. It also allows that more memory does not have be given over and over again by the system because of the amount of fragmentations.

All tables consist of values after cleanup.

Default Equilibrium Diver Without Coalescing		Default Equilibrium Driver With Coalescing	
Total Memory Size	11145216 Bytes	Total Memory Size	5689344 Bytes
# Of Chunks In List	49556 Chunks	# Of Chunks In List	31994 Chunks
Avg Size of Chunks	221 Units	Avg Size of Chunks	177 Units
Time To Complete	24.441 Seconds	Time To Complete	2.950 Seconds

As you can see the values of the Default Equilibrium Driver With Coalescing is an overall better method to output memory. It takes less time and does not waste as much memory because of fragmentation.

Values using the special case “-r 0”, when the size of each allocation request is the same.

Default Equilibrium Diver Without Coalescing		Default Equilibrium Driver With Coalescing	
Total Memory Size	614400 Bytes	Total Memory Size	614400 Bytes
# Of Chunks In List	1198 Chunks	# Of Chunks In List	245 Chunks
Avg Size of Chunks	512 Units	Avg Size of Chunks	2507 Units
Time To Complete	.729485 Seconds	Time To Complete	.282512 Seconds

Using the equilbrium driver in this case you can see that the total number of bytes are the same however the time, the # of chunks in list and the average size of the chunks as a result of the # of chunks in the list are all much better with coalescing because of the lack of fragmentation.

Comparing values after cleanup with all sorts using either Malloc or Mem_alloc

Equilibrium Diver with Coalescing Using Malloc and First Sort		Equilibrium Driver With Coalescing Using Mem_Alloc and First Sort	
Total Memory Size	675840 Bytes	Total Memory Size	5689344 Bytes
# Of Chunks In List	512 Chunks	# Of Chunks In List	31994 Chunks
Time To Complete	.238256 Seconds	Time To Complete	3.079 Seconds

Equilibrium Diver with Coalescing Using Malloc and Best Sort		Equilibrium Driver With Coalescing Using Mem_Alloc and Best Sort	
Total Memory Size	675840 Bytes	Total Memory Size	602112 Bytes
# Of Chunks In List	512 Chunks	# Of Chunks In List	297 Chunks
Time To Complete	.226311 Seconds	Time To Complete	.494462 Seconds

Equilibrium Diver with Coalescing Using Malloc and Worst Sort		Equilibrium Driver With Coalescing Using Mem_Alloc and Worst Sort	
Total Memory Size	675840 Bytes	Total Memory Size	655360 Bytes
# Of Chunks In List	512 Chunks	# Of Chunks In List	17 Chunks
Time To Complete	.218952 Seconds	Time To Complete	.296297 Seconds

In all cases, Malloc worked faster the Mem_alloc. There was the biggest difference in the first sorting function. The total memory size was a lot bigger for the first sort and also took a lot longer than malloc or any of the the other sort options.
However if coalescing was not used in all these sorts, the run time would take a lot longer and you can expect lot more memory size and fragmentations.

Unit Driver Testing Evaluation:

```
struct chunk_t *z = Mem_alloc(sizeof(chunk_t)*255);
        printf("need 255 units or the entire block"); // This special first case sees if it the entire
block will be removed as soon as it is initialized.
        Mem_print();
        Mem_free(z);
        printf("should return back the entire block");
        Mem_print();
        struct chunk_t *q = Mem_alloc(sizeof(chunk_t)*4); // Once the block that was removed
//returns back to the free list, mem_alloc calls will begin shaving from it.
        printf("need 4 units\n");
        Mem_print();
        struct chunk_t *b = Mem_alloc(sizeof(chunk_t)*8);
        printf("need 8 units\n");
        Mem_print();
        Mem_free(b);
        printf("should return back 8 units\n");
        Mem_print();
        struct chunk_t *w = Mem_alloc(sizeof(chunk_t)*400);
        printf("need 400 units\n");
        Mem_print();
        struct chunk_t *e = Mem_alloc(sizeof(chunk_t)*600);
        printf("need 600 units\n");
        Mem_print();
        struct chunk_t *r = Mem_alloc(sizeof(chunk_t)*130);
        printf("need 130 units\n");
        Mem_print();
        struct chunk_t *a =  Mem_alloc(sizeof(chunk_t)*112);
        printf("need 112 units\n");
        Mem_print();
        struct chunk_t *t = Mem_alloc(sizeof(chunk_t)*57);
        printf("need 57 units\n");
        Mem_print();
        Mem_free(a); // When coalescing is on, these memory blocks will join with the next
and previous
        printf("should return back 112\n");
        Mem_print();
        struct chunk_t *y = Mem_alloc(sizeof(chunk_t)*1);
        printf("need 1 units\n");
        Mem_print();
```

Mem_free(q); **// There is a variety of different mem_allocs, mem_frees and mem_prints in this unit driver.**

Mem_free(w); **// This ending allows all the memory that was malloced to be returned back to the free list**

Mem_free(e); **// just as a real program and a user would do. finally mem_stats will report if all memory**

Mem_free(r); **// has been returned and some other information**

Mem_free(t);

Mem_free(y); **// in the end all the memory blocks are returned and linked or if coalescing is on then al**l

Mem_print(); **// memory blocks are not conjoined into one large memory block.**

Mem_stats();