In this project, you will be implementing a memory allocator for the heap of a user-level process. Your task is to build your own malloc() and free().

Memory allocators have two distinct tasks. First, the memory allocator asks the operating system to expand the heap portion of the process's address space by calling mmap. Second, the memory allocator gives out this memory to the calling process. This involves managing a free list of memory and finding a contiguous chunk of memory that is large enough for the user's request; when the user later frees memory, it is added back to this list.

Program Specifications

You are required to make following functions:

- **Data structure:** Maintain the following linked list for keeping track of free space:

```
Typedef struct __list_t{
        int size;
        _Bool type;
        struct __list_t *next;
}*MemNodePtr;
```

- **int Mem_Init(int sizeOfRegion):** Mem_Init is called one time by a process using your routines. sizeOfRegion is the number of bytes that you should request from the OS using mmap().You need to wrap up size of region in term of page size.

  Return 0 on a success (when call to mmap is successful). Otherwise, return -1 and set m_error to E_BAD_ARGS. Mem_Init should return a failure when it is called more than once or sizeOfRegion is less than or equal to 0.

- **void *Mem_Alloc(int size, int style):** Mem_Alloc() is similar to the library function malloc(). Mem_Alloc takes as input the size in bytes of the object to be allocated and returns a pointer to the start of that object. The function returns NULL if there is not enough contiguous free space within sizeOfRegion allocated by Mem_Init to satisfy this request (and sets m_error to E_NO_SPACE).

  The style parameter determines how to look through the list for a free space. It can be set WORSTFIT (WF) for worst-fit, and FIRSTFIT (FF) for first-fit. WF looks for the largest chunk and allocates the requested space out of that; FF looks for the first chunk that fits and returns the requested space out of that.

- **int Mem_Free(void *ptr):** Mem_Free() frees the memory object that ptr points to. Just like with the standard free(), if ptr is NULL, then no operation is performed. The function returns 0 on success, and -1 otherwise.
- **Coalescing:** Mem_Free() should make sure to coalesce free space. Coalescing rejoins neighboring freed blocks into one bigger free chunk, thus ensuring that big chunks remain free for subsequent calls to Mem_Alloc().

- **void Mem_Dump():** This is just a debugging routine for your own use. Have it print the regions of free memory to the screen.

Making shared library:

You must provide these routines in a shared library named "libmem.so". Placing the routines in a shared library instead of a simple object file makes it easier for other programmers to link with your code. To create a shared library named libmem.so, use the following commands (assuming your library code is in a single file "mem.c"):

```
gcc -c -fpic mem.c -Wall -Werror
gcc -shared -o libmem.so mem.o
```

To link with this library, you simply specify the base name of the library with "-lmem" and the path so that the linker can find the library "-L.".

```
gcc -lmem -L. -o myprogram mymain.c -Wall -Werror
```

Of course, these commands should be placed in a Makefile. Before you run "myprogram", you will need to set the environment variable, LD_LIBRARY_PATH, so that the system can find your library at run-time. Assuming you always run myprogram from this same directory, you can use the command:

```
LD_LIBRARY_PATH=directory in which you are
export  LD_LIBRARY_PATH
```

Restrictions and Precautions:

1. Dont use malloc or free in your function.
2. Similarly, you should not allocate global arrays. However, you may allocate a few global variables (e.g., a pointer to the head of your free list.)
3. Take all errors into account so your code don’t hang indefinitely.
4. Carefully update linked list on each operation.