Final Project, Part 1 - A Basic Memory Management System

** You may work in groups of TWO people for this assignment **

The aim of Part 1 is to implement a simple, first-fit dynamic memory management policy on an allocatable region of memory, **M1**. The idea is to write two routines called *Malloc()* and *Free()* that operate on **M1**, which is established by the operating system. This is the basis for how heap-based memory management works for applications running on a real system, or a runtime environment such as a Java virtual machine.

Approach

You need to create the following functions, with *exactly* the type declaration as shown:

```
void Init (size_t size) {

  addrs_t baseptr;

  /* Use the system malloc() routine (or new in C++) only to allocate size bytes for
     the initial memory area, M1. baseptr is the starting address of M1. */
  baseptr = (addrs_t) malloc (size);

  Perform any other initialization here.
}

addrs_t Malloc (size_t size) {

  Implement your own memory allocation routine here.
  This should allocate the first contiguous size bytes available in M1.
   Since some machine architectures are 64-bit, it should be safe to allocate space starting
   at the first address divisible by 8. Hence align all addresses on 8-byte boundaries!

   If enough space exists, allocate space and return the base address of the memory.
   If insufficient space exists, return NULL.
}

void Free (addrs_t addr) {

   This frees the previously allocated size bytes starting from address addr in the
   memory area, M1. You can assume the size argument is stored in a data structure after
   the Malloc() routine has been called, just as with the UNIX free() command.
}

addrs_t Put (any_t data, size_t size) {

  Allocate size bytes from M1 using Malloc().
  Copy size bytes of data into Malloc'd memory.
  You can assume data is a storage area outside M1.
  Return starting address of data in Malloc'd memory.
}

void Get (any_t return_data, addrs_t addr, size_t size) {

   Copy size bytes from addr in the memory area, M1, to data address.
   As with Put(), you can assume data is a storage area outside M1.
   De-allocate size bytes of memory starting from addr using Free().
}
```

Additionally, you will need to keep a *free-list* of allocated and available slots in the memory area created by *Init()*. Moreover, when a block of memory is freed, it must be merged together with other contiguous free memory (either side of the block being freed). As memory is allocated, the memory area, **M1**, is fragmented and the free-list should keep track of these fragments and their sizes.

WARNING: make sure all Malloc'd addresses are 8 bytes aligned, to avoid possible address errors which the compiler cannot detect.

Testing

You should test your basic memory management system by:

- allocating space for a number of variable-length messages,
- placing these messages into the allocatable memory area, beginning at the address returned from your *Malloc()* routine,
- removing each message from the memory area, copying it to local address space (outside the memory area), freeing up the space occupied by the message in the memory area (using your own *Free()* routine), and finally displaying the message.

The following type of test code should appear in your main() function, although you will realistically want to stress test your heap management system with a greater range of requests:

```
typedef char *addrs_t;
typedef void *any_t;

void main (int argc, char **argv) {

  int i, n;
  char s[80];
  addrs_t addr1, addr2;
  char data[80];
  int mem_size = DEFAULT_MEM_SIZE; // Set DEFAULT_MEM_SIZE to 1<<20 bytes for a heap region

  if  (argc > 2) {
    fprintf (stderr, "Usage: %s [memory area size in bytes]\n", argv[0]);
    exit (1);
  }
  else if (argc == 2)
    mem_size = atoi (argv[1]);

  Init (mem_size);

  for (i = 0;; i++) {
    n = sprintf (s, "String 1, the current count is %d\n", i);
    addr1 = Put (s, n+1);
    addr2 = Put (s, n+1);
    if (addr1)
      printf ("Data at %x is: %s", addr1, addr1);
    if (addr2)
      printf ("Data at %x is: %s", addr2, addr2);
    if (addr2)
      Get ((any_t)data, addr2, n+1);
    if (addr1)
      Get ((any_t)data, addr1, n+1);
  }
}
```

Part 2 - A Virtualized Heap Allocation Scheme

For this part of the assignment, you are now tasked with the development of two new functions, *VMalloc()* and *VFree()* that operate on a new heap memory area, **M2**. Significantly, these two functions have the following prototypes:

```
addrs_t *VMalloc (size_t size);
void VFree (addrs_t *addr);
```

These functions behave almost identically to Malloc() and Free(), except they work with POINTERS to addrs_t. Specifically, these pointers go to a *redirection table* that is an array of **R** pointers to allocated blocks of memory within the heap. Each

pointer returned by VMalloc(), and later passed to VFree(), points to a specific entry in the redirection table. The redirection table has one pointer entry for each allocated block of memory requested by a prior VMalloc(), but which has not yet been freed.

For every call to VMalloc() or VFree(), the memory allocator must compact all current allocations to consume the least amount of space in the heap region. This means moving those allocations (and their contents) to different heap regions that are aligned on 8-byte boundaries. Once an allocation is moved, the address referring to it in the redirection table needs to be updated to keep track of where it is located. This means that, aside for unusable chunks of memory used to pad data on aligned address boundaries, there is ONE large chunk of contiguous memory that is currently unallocated. Having your memory allocator keep track of the size of this free heap region will quickly identify whether or not your heap can satisfy a subsequent VMalloc() request for a given size.

In essence, VMalloc() and VFree() are similar to Malloc() and Free() except they perform compaction and move allocated data around the heap to make way for one large chunk of unallocated memory. This should mean they are slower than Malloc() and Free() but can potentially lead to more dynamic memory requests being satisfied.

Example Redirection Table (for use with VMalloc() and VFree())

Redirection Table

Final Testing

For final testing, we will provide a sequence of allocation and deallocation requests on two regions of memory (**M1** and **M2**) of some specific size. You will then compare Malloc()/Free() **[Case 1]** against VMalloc()/VFree() **[Case 2]** for each of these requests.

You should use the following code to measure the clock cycles for your requests. We will give you a test template at a later date with this timing code around a series of requests. You should attempt to come up with the most efficient ways of implementing Parts 1 and 2.

```
#define rdtsc(x)       __asm__ __volatile__("rdtsc \n\t" : "=A" (*(x)))
unsigned long long start, finish;

rdtsc(&start);

// Insert your code here that you wish to time

rdtsc(&finish);
```

Note also for Part 2, you will need to create two functions VPut() and VGet() to test your VMalloc() and VFree().

The function prototypes for VPut() and VGet() are:

addrs_t *VPut (any_t data, size_t size) {

Allocate *size* bytes from **M2** using *VMalloc()*.
Copy *size* bytes of *data* into Malloc'd memory.
You can assume *data* is a storage area outside **M2**.
Return *pointer to redirection table* for Malloc'd memory.
}

void VGet (any_t return_data, addrs_t *addr, size_t size) {

Copy *size* bytes from the memory area, **M2**, to *data* address. The
addr argument specifies a pointer to a redirection table entry.
As with *VPut()*, you can assume *data* is a storage area outside **M2**.
Finally, de-allocate size bytes of memory using V*Free()* with *addr*
pointing to a redirection table entry.
}

Data Structures and Heap Checker

You are free to choose whatever data structures you need to manage the size of each allocated chunk of data from your heap. You will somehow need to track the size of an allocation so that a subsequent free of that memory will de-allocate the right number of bytes. ****You must allocate space within the heap for any data structures you use to track the allocation of space and which parts of the heap are free. However, the redirection table in the above diagram is assumed to be OUTSIDE the heap area.****

You should also pay attention to how you combine free blocks into single larger blocks, when two or more contiguous blocks are formed. Issues such as internal fragmentation (due to padding and alignment) and external fragmentation (due to small-sized unallocated blocks) should be considered. Also, the data structure you use to track free and allocated blocks is up to you.

You should develop a heap checker program that reports for each Malloc/Free/VMalloc/VFree call the state of the heap. The state of the heap should identify the following in *exactly* the format requested, where XXXX is replaced by a 64-bit long integer:

<<Part 1 for Region M1>>
Number of allocated blocks : XXXX
Number of free blocks : XXXX (discounting padding bytes)
Raw total number of bytes allocated : XXXX (which is the actual total bytes requested)
Padded total number of bytes allocated : XXXX (which is the total bytes requested plus internally fragmented blocks wasted due to padding/alignment)
Raw total number of bytes free : XXXX
*Aligned total number of bytes free : XXXX (which is sizeof(**M1**) minus the padded total number of bytes allocated. You should account for meta-datastructures inside **M1** also)*
Total number of Malloc requests : XXXX
Total number of Free requests: XXXX
Total number of request failures: XXXX (which were unable to satisfy the allocation or de-allocation requests)
Average clock cycles for a Malloc request: XXXX

Average clock cycles for a Free request: XXXX
Total clock cycles for all requests: XXXX

<<Part 2 for Region M2>>
Number of allocated blocks : XXXX
Number of free blocks : XXXX
Raw total number of bytes allocated : XXXX
Padded total number of bytes allocated : XXXX
Raw total number of bytes free : XXXX
*Aligned total number of bytes free : XXXX (which is sizeof(**M2**) minus the padded total number of bytes allocated. You should account for meta-datastructures inside **M2** also)*
Total number of VMalloc requests : XXXX
Total number of VFree requests: XXXX
Total number of request failures: XXXX
Average clock cycles for a VMalloc request: XXXX
Average clock cycles for a VFree request: XXXX
Total clock cycles for all requests: XXXX

NOTE: You do *not* need to include in your heap checker output anything shown above in parentheses. Additionally, the invariant with Part 2 (using the redirection table) is that for every memory request there should be *one* free block, discounting small unusable padding blocks. Depending on the tested requests, some of the above values may be 0.

Grading

Grading will check for:

- Proper implementation of Malloc()/Free() and VMalloc()/VFree()
- No memory leaks, first-fit policy, no address errors
- No segmentation faults, correct address alignment and range checking
- Correctly implemented free-list and/or other data structures as appropriate
- Program style and comments
- Performance/Efficiency
- Correct heap checker output

Happy Programming!