Machine Problem X: A Simple Memory Allocator (Due: 05/05/17)

Introduction:

The kernel manages the physical memory both for itself and for all of the tasks and user processes that run on the system. The memory occupied by the kernel code is reserved and is never used for any other purpose. Physical memory separate from this reserved space, can be used for any other task imaginable in the system. Most of this memory must be allocated and de-allocated dynamically, and an algorithm must be in place to keep track of which physical memory is in use, and by whom.

For several practical reasons, a large number of operating systems (Linux/UNIX included) partition the memory into zones, and treat each zone separately for allocation purposes. Memory allocation requests typically come with a list of zones that can be used to satisfy the request. For example, a particular request may be preferably satisfied from the "normal" zone. If that fails, then the allocation may possibly be reattempted from the high-memory zone that needs special access mechanisms. If the previous two requests fail, allocating from more zones can be attempted as long as the process has permissions to do so.

Within each zone, many systems (for example Linux) use a buddy-system allocator to allocate and free physical memory. This is what you will be providing in this machine problem (i.e. an allocator for a single memory zone, and not for an entire physical system).

In this machine problem, you are to develop a simple memory allocator that implements, using the **buddy system memory allocation scheme**, the functions my_malloc() and my_free(), similar to the standard C library calls: malloc() and free(). The general objectives of this machine problem are:

- Package a simple module with some static data as a separately compiled unit.
- Become deeply familiar with pointer management and array management in the C/C++ languages.
- Become familiar with standard methods for handling command-line arguments in a C/UNIX environment.
- Become familiar with simple UNIX development tools such as: the compiler, make, the GNU debugger, object file inspector, etc..

Assignment:

You are to implement a buddy-system memory manager using C or C++ by your choice that allocates memory in blocks with sizes that are power-of-two multiples of a basic block size. The basic block size is given as an argument when the allocator is initialized.

- The memory allocator shall be implemented as a C/C++ module my_allocator, which consists of a header file my_allocator.h and my_allocator.c. (Starter versions of the header file and .c file are provided to you)
- Evaluate the correctness (Up to a certain point) and the performance of your allocator. For this, you will be given the source code of a function which implements the highly-recursive Ackerman function. In this implementation of the Ackerman function, random blocks of memory are allocated and then de-allocated sometime later, generating a large combination of different allocation and de-allocation patterns. The Ackerman function is provided in the form of two files (i.e. the header file ackerman.h with the interface definition of the Ackerman function, and the implementation in file ackerman.c).
- You will write a program called memtest which reads the basic block size and the memory size (in bytes) from the command line, initializes the memory, and then calls the Ackerman function. It must measure the time it takes to perform the number of memory operations. Make sure that the program exits cleanly if aborted (Using an `atexit()` handler is a good way to do this).
- Use the getopt() C library function to parse the command line for arguments. The usage of the memtest function should be of the form:

  memtest [-b <blocksize>] [-s <memsize>]
  -b <blocksize> defines the block size, in bytes. The default should be 128 bytes if the -b argument is not provided.
  -s <memsize> defines the size of the memory to be allocated, in bytes. The default should be 512 kB.

- Repeatedly invoke the Ackerman function with increasingly larger number of values for n and m (be careful to keep $n \leq 3$ as the processing time increases steeply for large values of n). Identify at least one point that you may modify in the simple buddy system described above to improve the performance. Also be sure to provide an argument as to why/how your proposed change would improve performance.
- Make sure that the allocator gets de-allocated (and has its memory freed) when the program either exits or aborts. Use the atexit library function for this.

Delivarables:

You are to hand in three code files: **my_allocator.h and my_allocator.c which define and implement your memory allocator, as well as memtest.c, which implements the main program. The header file my_allocator.h will likely not change from the provided file. If you need to change it, give a compelling reason in the modified source code.**

Additionally, hand in a file (called analysis.pdf, in PDF format) with the analysis of the effects of the performance of your allocator for increasing numbers of allocate/free operations. Vary this number by varying the parameters n and m in the Ackerman function. Determine where the bottlenecks are in the system, or where poor implementation is affecting performance. Identify at least one point that you would modify in this simple implementation to improve performance. Argue why it would improve performance. The complete analysis can be made in 500 words or less with at least one or two graphs. Make sure that the analysis file contains your name to associate it with you.

Background Concepts:

Buddy-System Memory Allocation:

In our buddy-system memory allocator, memory block sizes must be a power of two with a minimum size defined by the variable `basic_block_size`. For example, if 9kB of memory is requested, the allocator must return the nearest power of two, 16kB. Because of this 7kB is wasted in a process known as fragmentation. In spite of the potential for wastage, the restriction of block sizes to powers of two makes the management of free memory blocks easy. The allocator must simply keep an array of free lists, one for each allowable block size. Every request is rounded up to the next allowable size, and the corresponding free list is checked. If there is an entry in the free list, this entry is returned and then removed from the free list.

In the event that the free list for that particular block size is empty (i.e. there are no free memory blocks of that size), a larger block is selected using the free list of some larger block size and then recursively split until a block of the desired size is obtained. Whenever a free memory block is split in two, one block gets either used or further split, and the other unused block, its buddy, is added to the corresponding free list.

For example: Assume that the system needs a 16kB block, but the free list is empty. Also assume that the free list for 32kB blocks is empty, but the 64kB list has a free element, B. The allocator should therefore select the 64kB block and split it into two blocks, B_L and B_R, each 32kB and add then to the 32kB free list. B_L should then be split again into B_{LL} and B_{LR}, each 16kB. Finally, B_{LL} should be used and B_{LR} should be added to the 16kB free list.

In the previous example, the blocks B_L and B_R are buddies, as are B_{LL} and B_{LR}. Whenever a memory block is freed, it may be coalesced with its buddy. Coalescing can only occur if the buddy is free as well. If it is, then the two buddies can be combined to form a single memory block that is twice the size of each buddy individually.

For example: Assume that B_{LL} and B_{LR} are free, and that we have just freed B_{LR}. In this case, B_{LL} and B_{LR} can be coalesced into the single block B_L. We therefore, delete B_{LL} from its free list and proceed to insert the newly formed B_L into into the next higher free list. Before we do that, we check to see if its buddy, B_R is free. If so, then B_L and B_R can be recombined to form B, recreating the 64kB block that was split in the first example of this section.

Finding Buddies:

The buddy system performs two operations on memory blocks: splitting and coalescing. Whenever we split a free memory block of size 2^n with start address A, we generate two buddies: one with start address A, and the other with the start address of A that has the $(n-1)^{th}$ bit flipped.

Finding the buddy of a block being freed is just as simple when the size of the block is known: The address of the buddy block is determined by flipping the appropriate bit of the block's start address, just as is the case when we split a block. The problem is: How will we know what the block size is when the program needs to split or coalesce a block?

The easiest way is to explicitly store it at the beginning of the allocated block as a part of a header. This wastes memory as the space where the header is being stored cannot be used by the requesting program to store any data. Alternatively, the size can be implicitly inferred from other data, typically stored in the free list. For example, Linux uses a buddy bitmap for each free list. In this bitmap, each bit represents two adjacent blocks of the same size. The bit is 0 if both the blocks are either full or free, and is 1 if exactly one block is free and the other is allocated. By comparing these bits for increasing block sizes, we can infer the current block size. This is also not tenable, as the size of the bitmaps depends on the amount of memory available.

Managing the Free List:

You want to minimize the amount of space needed to manage the free list. For example, you do not want to implement the lists using traditional means (i.e. with dynamically created elements that are connected with pointers). An easy solution is to use the free memory blocks themselves to store the free-list data. For example, the first bytes of each free memory block would contain the pointer to the previous and to the next free memory block of the same size. The pointers in the first and last block in each free list can easily be stored in an array of pointers, two for each allowable block size.

Note on Block Size:

If you decide to put management information into allocated blocks such as the size of the block, you have to be careful about how this may affect the size of the allocated block. For example, when you allocate a block of size 5, and add an 8-byte header to the block, you actually need room for the 8-byte header along with the 5-bytes for the data space requested. Consequently, you would need a 13-byte block instead of a 5-byte one (in case you couldn't tell, this is extremely wasteful).

Where does the allocator get memory from?

Inside the initializer, your program must acquire the required amount of memory from the system. It must interface with the existing system library, `malloc()`, in order to do so. Remember, your allocator must also return the memory to the system when it is done using the `free()` function.

What does my_malloc() return?

In the case above, putting the management information block in front of the allocated block is an effective way to access it easily. In this case, make sure that your `my_malloc()` routine returns the address of the allocated block, not the address of the management information block. There is no guarantee that the program calling my_malloc won't completely overwrite the header information.

Initializing the Free List and the Free Blocks:

You are given the size of the available memory as argument to the `init()` method. The number entered by a potential user is likely not a power of two multiple of the basic block size. However, in order to function correctly, your program must partition the memory into a sequence of power-of-two sized blocks and initialize the blocks and the free list accordingly. This means that your program could use more memory than requested by the user.

Grading Rubric:

A total of 200 points is assigned to this machine problem, which consists of 140 points for the code and 60 points for the report. Below is a breakdown of the code part:

Function	Points
init_allocator	15
release_allocator	5
my_malloc	30
my_free	30
print_list	10
buddy_address	10
split_block	20
combine_block	20

Here is the guideline for some of the functions.

For init_allocator, make sure it is not already initialized. Don't run the initialization routine again if the mem array has already been set up. After that, check if the parameters make sense: make sure basic_block_size < header size < mem_size. Third, round block_size and mem_size up to a power of two. Forth, set block and mem size, and setup memory array. Last, allocate and initialize pointers.

For release_allocator, make sure you don't release something isn't initialized, otherwise coalesce allocated memory regions back into one that can be freed, then free head_pointer and memory_array, and set everything to null and set initialized = false at the end.

For my_malloc, similar to malloc() in C, allocates the requested memory and returns a pointer of the allocated block, not the address of the management information block.

For my_free, similar to free() in C, is to deallocate the memory previously allocated.

print_list is a function simply prints the information of what's left in the buddy system, if the list is empty, print an error message. print_list is used for debugging and verification, therefore there is no specific format required.

buddy_address is a function that takes a block address and return its buddy address if exists. If there is no buddy address, return error message.

For split_block, make sure the given block is valid and not in use. Don't do anything if

the block size is already the basic block size. Remove the block from its current list, split the block into two smaller blocks and then set data values and insert into new list.

For combine_block, make sure the two blocks are the same size and they are not the same block. Check to see if the blocks are buddies, and make sure that mem_1 address comes before mem_2 address. If the blocks aren't buddies, they can't be combined. Remove the two blocks from the list they belong to and insert the combined block.