Machine Problem 1: A Simple Memory Allocator

Introduction

In this machine problem, you are to develop a simple **memory allocator** that implements the functions `my_malloc()` and `my_free()`, very similarly to the UNIX calls `malloc()` and `free()`. (Let's assume that – for whatever reason – you are unhappy with the memory allocator provided by the system.) The 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 language (or C++ for that matter).
- Become familiar with standard methods for handling command-line arguments in a C/UNIX environment.
- Become familiar with simple UNIX development tools (compiler, make, debugger, object file inspector, etc.)

Background: Kernel Memory Management. The kernel manages the physical memory both for itself and for the system and user processes. The memory occupied by the kernel code and its data is reserved and is never used for any other purpose. Other physical memory may be used as frames for virtual memory, for buffer caches, and so on. Most of this memory must be allocated and de-allocated dynamically, and an infrastructure must be in place to keep track of which physical memory is in use, and by whom.

Ideally, physical memory should look like a single, contiguous segment from which an allocator can take memory portions and return them. This is not the case in most systems. Rather, different segments of physical memory have different properties. For example, DMA may not be able to address physical memory above 16MB. Similarly, the system may contain more physical memory than what can be directly addressed, and the segments above need to be handled using appropriate memory space extension mechanisms. For all these reasons, many operating systems (for example Linux) partition the memory into so-called **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, from the high-memory zone that needs special access mechanisms. Only if that fails too, the allocation may attempt to allocation from the DMA zone.

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 (for a single zone, and of course not at physical memory level).

Your Assignment. You are to implement a C module (`.h` and `.c` files) that realizes a memory allocator as defined by the following file `my_allocator.h`:

```
#ifndef _MY_ALLOCATOR_H_
#define _MY_ALLOCATOR_H_

/* File: my_allocator.h */

typedef void * Addr;
```

```
unsigned int init_allocator(unsigned int _basic_block_size, unsigned int _length);
/* This function initializes the memory allocator and makes a portion of
   '_length' bytes available. The allocator uses a '_basic_block_size' as
   its minimal unit of allocation. The function returns the amount of
   memory made available to the allocator. If an error occurred, it returns 0. */

int release_allocator();
/* This function returns any allocated memory to the operating system.
   After this function is called, any allocation fails. */

 Addr my_malloc(unsigned int _length) {};
 /* Allocate _length number of bytes of free memory and returns the
    address of the allocated portion. Returns 0 when out of memory. */

 int my_free(Addr _a) {};
 /* Frees the section of physical memory previously allocated
    using 'my_malloc'. Returns 0 if everything ok. */

#endif
```

You are to provide the implementation of the memory allocator in form of the file `my_allocator.c`. The memory allocator you are supposed to implement is based on the so-called "Buddy-System" scheme. (A description of this scheme is given in Section 9.8.1 of the Silbershatz *et al.* textbook. A more detailed description is given in Section 2.5 of D. Knuth, "The Art of Computer Programming. Volume 1 / Fundamental Algorithms". We give a short overview below.)

Buddy-System Memory Allocation:

In our buddy-system memory allocator, memory block sizes are a power of two, starting at the *basic block size*. For example, if 9kB of memory are requested, the allocator returns 16kB, and 7kB goes wasted – this is called *fragmentation*. This restriction on allowable block sizes makes the management of free memory blocks very easy: The allocator keeps 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 simply used and deleted from the free list.

If the free list is empty (i.e. there are no free memory blocks of this size,) a larger block is selected (using the free list of some larger block size) and split. Whenever a free memory block is split in two, one block gets either used or further split, and the other – its *buddy* – is added to its corresponding free list.

Example: In the example above, there is no 16kB block available (i.e. the free list for 16kB is empty). The same holds for 32kB blocks. The next-size available block is of size 64kB. The allocator therefore selects a block, say B, of size 64kB (after deleting it from the free list). It then splits B into two blocks, B_L and B_R of 32kB each. Block B_R is added to the 32kB free list. Block B_L is further split into B_{LL} and B_{LR} of size 16 kB each. Block B_{LL} is returned by the request, while B_{LR} is added to the 16kB free list.

In this 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: If the buddy is free as well, the two buddies can be combined to form a single memory block of twice the size of each

buddy.

Example: Assume that B_{LL} and B_R are free, and that we are just freeing 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 its free list. Before we do that, we check with its buddy B_R. In this example, B_R is free, which allows for B_L and B_R to be coalesced in turn, to form the 64kB block B. In this process, Block B_R is removed from its free list and the newly-formed block B is added to the 64kB free list.

Finding Buddies: The buddy system performs two operations on (free) memory blocks, *splitting* and *coalescing*. Whenever we split a free memory block of size 2^s with start address A, we generate two buddies: one with start address A, and the other with start address A with the $(s-1)^{th}$ 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 to get hold of the block size? The easy way is to explicitly store it at the beginning of the allocated block, as part of a *header*. This wastes memory. Alternatively, the size can be implicitly infered 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 pretty, as the sizes 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 to 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 (e.g. the size, as described above), 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-word header to the block, you are actually allocating a 5 blocks +8 Word, which requires a block of size 8! (This is extremely wasteful.)

Where does my allocator get the memory from? Inside the initializer you will be allocating the required amount of memory from the run time system, using the `malloc()` command. Don't forget to free this memory when you release the allocator.

What does `my_malloc()` return? In the case above, putting the management information block in front of the the allocated memory block is as good a place as any. In this case make sure that your `my_malloc()` routine returns the address of the allocated block, *not* the address of the management info block.

Initializing the Free List and the Free Blocks: You are given the size of the available memory as argument to the `init()` method. The given memory size is likely not a power-of-two multiple

of basic blocks. You are to partition the memory into a sequence of power-of-two sized blocks and initialize the blocks and the free list accordingly.

The Assignment

You are to implement a buddy-system memory manager 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 module `my_allocator`, which consists of a header file `my_allocator.h` and `my_allocator.c`. (A copy of the header file and a rudimentary preliminary version of the `.c` file are provided.)
- Evaluate the correctness (up to some point) and the *performance* of your allocator. For this you will be given the source code of a function with a strange implementation of a highly-recursive function (called *Ackermann* function). In this implementation of the Ackermann function, random blocks of memory are allocated and de-allocated sometime later, generating a large combination of different allocation patterns. The Ackerman function is provided in 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 Ackermann function. It measures the time it takes to perform the number of memory operations. Make sure that the program exits cleanly if aborted (using `atexit()` to install the exit handler.)
- Use the `getopt()` C library function to parse the command line for arguments. The synopsis of the `memtest` program is of the form

```
memtest [-b <blocksize>] [-s <memsize>]

-b <blocksize> defines the block size, in bytes. Default is 128
               bytes.
-s <memsize>   defines the size of the memory to be allocated, in
               bytes. Default is 512kB.
```

- Repeatedly invoke the Ackerman function with increasingly larger numbers of values for n and m (be careful to keep $n \leq 3$; the processing time increases very steeply for larger numbers of n). Identify *at least one point* that you may modify in the simple buddy system described above to improve the performance, and argue why it would improve performance.
- Make sure that the allocator gets de-allocated (and its memory freed) when the program either exits or aborts (for example, when the user presses Ctrl-C). Use the `atexit` library function for this.

What to Hand In

- You are to hand in three files, with names `my_allocator.h` and `my_allocator.c`, which define and implement your memory allocator, and `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.
- Hand in a file (called `analysis.pdf`, in PDF format) with the analysis of the effects on the performance of the system 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, and one or two graphs. Make sure that the analysis file contains your name.
- Grading of these MPs is a very tedious chore. These handin instructions are meant to mitigate the difficulty of grading, and to ensure that the grader does not overlook any of your efforts.
- **Failure to follow the handing instructions will result in lost points.**