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 to

- Package a simple module with some static data as a separately compiled unit.
- Become deeply familiar with pointer 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 be attempted 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 Mission

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

```
typedef void * Addr;

size_t 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(size_t _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. */
```

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 *Fibonacci Buddy System*, which in turn is a generalization of the so-called "Buddy-System" scheme. (The Fibonacci Buddy System scheme is described in D. S. Hirschberg: "A class of dynamic memory allocation algorithms" in *Communications of the ACM*, 16(10):615:618, October 1973. A description of the Binary Buddy-System scheme – so-to-say Buddy System in the narrow sense – is given in Section 9.8.1 of the Silbershatz *et al.* textbook. A more detailed description of the Binary Buddy System is given in Section 2.5 of D. Knuth, "The Art of Computer Programming. Volume 1 / Fundamental Algorithms". We give a very short overview of the Fibonacci Buddy System below.)

Fibonacci Buddy-System Memory Allocation:

Buddy-system allocators allocate memory in predefined block sizes, which are integer multiples of a *basic block size* (powers of two in the case of binary buddy systems, and Fibonacci numbers[1] in the case of Fibonacci buddy systems). For example, if 9kB of memory are requested, the allocator returns 12kB (3 is a Fibonacci number, times a basic block size of 4kB,) and 3kB goes wasted – this is called *fragmentation*.

We will see that the restriction to 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 block 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.

Splitting Free Blocks: 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 block of size 3 available (i.e. the free list for size 3 is empty). The same holds for size-5 blocks. The next-size available block is of size 13. The allocator therefore selects a block, say B, of size 13 (after deleting it from the free list). It then splits B into

[1]Reminder: Fibonacci numbers satisfy the recursive relation $F(k) = F(k-1) + F(k-2)$. The numbers 1, 2, 3, 5, 8, 13, 21, ... are Fibonacci numbers.

two blocks, B_L of size 5 and B_R of size 8. Block B_R is added to the size-8 free list. Block B_L is further split into B_{LL} of size 2 and B_{LR} of size 3. Block B_{LR} is returned by the request, while B_{LL} is added to the size-2 free list.

In this example, the blocks B_L and B_R are buddies, as are B_{LL} and B_{LR}.

Coalescing Free Blocks: As more blocks get allocated and freed, the process above leads to lots of small free blocks, with no large free blocks left. Large free blocks are created by *coalescing* buddies: Two buddies – which by construction are neighboring – can be coalesced into a large free block if both buddies are free.

Coalescing can happen at different points in time, but is done most conveniently whenever a memory block is freed: If the buddy is free as well, the two buddies can be combined to form a single free memory block.

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 block B of size 13. In this process, Block B_R is removed from its free list and the newly-formed block B is added to the size-13 free list.

Finding and Coalescing Buddies: The buddy system performs two operations on (free) memory blocks, *splitting* and *coalescing*. Whenever we split a free memory block of size F_n (where F_n denotes the n^{th} Fibonacci number), with start address A, we generate two buddies: a left buddy with start address A and size F_{n-2}, and a right one with start address $A + F_{n-2}$ and size F_{n-1}.

Right Buddy or Left Buddy? When attempting to coalesce a free block, say B_X with its buddy, it is first necessary to know (a) what size the free block is, and (b) whether the free block is a right or a left block. Once this is known, the size and the start point of the buddy can be determined. If the buddy is free as well, the two blocks can be coalesced into a single, larger block. Note that we need to be able to infer whether the larger block is a right or a left block if it ever needs to be coalesced with its own buddy.

All this information can be maintained with the following trick: The size of the block (or better, its Fibonacci number index) is stored with the block. In order to maintain the Left/Right information of blocks, two bits are stored with each block. We call them the *Left/Right* bit and the *Inheritance* bit.

Whenever we split a block, the left child's *Left/Right* bit is set to *Left*, and the right child's bit is set to *Right*. In addition, we set the left child's *Inheritance* bit to be the *Left/Right* bit of the parent. The right child's *Inheritance* bit is set to the *Inheritance* bit of the parent.

Whenever we coalesce two blocks, the *Left/Right* bit of the new block is equal to the *Inheritance* bit of the left child, and the *Inheritance* bit of the new block is equal to the *Inheritance* bit of the right child.

In this fashion, information about buddies can be maintained across splits and merges.

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_allocator()` method. The given memory size is likely not a Fibonacci number. You are to partition the memory into a sequence of Fibonacci-number sized blocks and initialize the blocks and the free list accordingly.

The Assignment

You are to implement (in three steps) a buddy-system memory manager that allocates memory in blocks with sizes that are Fibonacci-number 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 an implementation file `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 the file `memtest.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.) Most of the code for `memtest` is already there in file `memtest.c`.
- 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 test program with increasingly larger numbers of values for a and b (be careful to keep $a \leq 3$; the processing time increases very steeply for larger numbers of a). 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. (**Note:** It may happen that one or more calls to `my_malloc()` fail because there is not enough memory left to allocate. In such cases the timing results are not indicative because the test program skips possibly many allocation/de-allocation steps. Ignore timing results of test runs that run out of memory.)
- 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.

Submission Process

This machine problem will be submitted using **three milestones**, where for each milestone you submit an improved version of your allocator. You will start with a simple but silly allocator in Milestone 1 and end with a fully functioning Fibonacci-Buddy-System allocator in Milestone 3. **This is just an overview. More details about each of the three milestones will be given in the lab!**

Milestone 1 - Salami Allocator: In this warm-up submission you will write a totally trivial allocator: The function `init_allocator()` reserves a given portion of memory for the allocator. Every call to `my_malloc()` cuts off the requested amount of memory from the remainder of the reserved memory. The call to `my_free()` does not free any memory. When all the memory has been used, and no more memory is left, subsequent calls to `my_malloc()` return NULL. For the implementation all you need is a pointer to the beginning of the remaining memory. Whenever memory is allocated, advance the pointer, until you reach the end. This allocator is a good start for what follows.

Milestone 2 - Single-Freelist Allocator: Once you get your brain wrapped around pointers and the general structure of allocators, we can proceed to a more sophisticated allocator, which allocates *and* frees memory using a single free list. The call to `init_allocator()` reserves the memory and initializes a free list of segments of length `blocksize` (a parameter to `init_allocator()`. Initially, all blocks are free and are therefore linked up in the free list. Calls to `my_malloc()` allocate one block (i.e. at most `blocksize` bytes) and remove it from the free list. Calls to `my_free()` return the memory by adding the block back to the free list.

For this you need a set of function to manipulate free lists and blocks in free lists. These functions are to be defined in a module called `free_list`, which consists of files `free_list.h` and `free_list.c`. The header file should look similar to the following:

```
typedef struct fl_header { /* header for block in free list */
   /* put stuff here */
} FL_HEADER; /* I don't like to type struct all the time, so */
             /* I typedef it away... */

void FL_remove(FL_HEADER * free_list, FL_HEADER * block);
/* Remove the given block from given free list. The free-list
   pointer points to the first block in the free list. Depending
   on your implementation you may not need the free-list pointer.
*/
void FL_add(FL_HEADER * free_list, FL_HEADER * block);
/* Add a block to the free list. */
```

Internally, the free list should be implemented as a doubly-linked list of block headers. The implementation of the free list will be discussed more in detail in the lab.

Milestone 3 - Full-fledged Fibonacci Buddy System Allocator In this milestone you add (1) support for multiple free lists (one for each supported Fibonacci number size, (2) splitting of blocks into smaller blocks, and (3) coalescing of small free blocks into larger free blocks. As a result, you will have a full-fledged Fibonacci Buddy System Allocator.

What to Hand In, for each Milestone

- 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.)]
- **Only for Milestone 3:** 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 a and b 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 hand-in 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.**

Submission and Grading of Machine Problems with multiple Milestones

Submission of Milestones: Each milestone must be submitted separately by its given deadline. Late-submission penalties apply for each milestone separately.

Credit for Milestones: In general, you have to submit all milestones of the machine problem in order to get full credit. Each milestone will give you partial credit.

Skipping Milestones: We use these milestones to give you guidelines and to keep you on schedule. If you feel that you don't want to be babied (really bad idea!) you can elect to skip the submission of individual milestones leading to the final milestone. If so, you need to submit a PDF statement instead of the milestone submission where you declare that you elect to skip the particular milestone. If you skip a milestone and then you fail to submit the subsequent milestone, the partial credit for the skipped milestone will be **counted against you**! (For example, if Milestone 1 were to give you 20 points, skipping it an then not submitting Milestone 2 would count not zero, but -20 points, whereas you would receive the 20 points prorated by the end result if you skipped Milestone 1 and then submitted Milestone 2. Remember that you have to submit a statement to skip a milestone, otherwise we just assume that you failed to submit, and you get zero points.) If you want to skip two milestone, you need to submit a separate statement **for each milestone** that you intend to skip. **You cannot skip the final milestone**.

If this all sounds too complicated, your best bet is to submit all the milestones.