Machine Problem 1: A Simple Memory Allocator

Warning:

This handout is to give a general overview of the machine problem and to specify requirements for submissions. It is not intended to be comprehensive! More details will be provided in the labs. Do not attempt to submit a solution based solely on this handout. You will not succeed.

Introduction

In this machine problem, you are to develop a simple **memory allocator** that implements the functions `malloc()` and `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

- Become deeply familiar with pointer and array management in the C/C++ languages.
- Become familiar with overloading of `new` and `delete` operators in C++.
- Become familiar with placement allocation in C++.
- 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: Memory Management. There are many levels in a system where memory needs to be managed.

- Inside the OS kernel, physical memory may be requested to be used as buffer for devices.
- The OS kernel may need request memory in order to manage objects such as files or thread control segments, or tables of various kinds.
- User-level programming environments (such as C++ or Java runtimes) provide support for the user to dynamically request and free memory. (Examples are `new` and `delete` in C++ or `malloc()` and `free()` in the C Language.)

In the case of user programs, we say that memory is dynamically requested and returned from and to the **heap**. In order to keep track of the free segments of memory on the heap, we need an algorithm and supporting data structures, which we call an **memory allocator**. Many different allocator algorithms exist, which each have their advantages and disadvantages, and their performance depends on the pattern of memory allocation requests that they have to handle. For example, some allocators work great when they have to serve frequent "small" requests that all have about the same size, while other allocators may work better with a mix of "small" and "large" requests.

In this machine problem, you will implement a so-called **buddy-system** allocator for memory.

Your Mission

You are to implement a C++ class (`.hpp` and `.cpp` files) that realizes a memory allocator as defined by the following excerpt from file `my_allocator.hpp`:

```
class MyAllocator {
private:
 /* -- YOU MAY NEED TO ADD YOUR OWN DATA STRUCTURES HERE. */
```

```
public:

  MyAllocator(size_t _basic_block_size, size_t _size);
  /* This function initializes the memory allocator and makes a portion of]
     '_size' bytes available. The allocator uses a '_basic_block_size' as]
     its minimal unit of allocation. */

  ~MyAllocator();
  /* This function returns any allocated memory to the operating system. */

   void* 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. */

   bool Free(void* _a);
   /* Frees the section of physical memory previously allocated]
      using 'Malloc'. Returns true if everything ok. */
}
```

The memory allocator you are supposed to implement in class `MyAllocator` will be 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.)

How will my Allocator be used?

You will use and test your allocator by replacing the standard C++ allocator with your new `MyAllocator`. You will do this by **overloading** the `new` and `delete` operators of C++.

We will provide you with a small test program, called `memtest`, which dynamically allocates and frees a bunch of integer arrays as follows:

```
    /*Here we ALLOCATE some memory using our own allocator.*/
    int * test_array = new int[to_alloc];

    /*Fill the allocated memory with a random value.*/
    /*This same value will be used later for testing.*/
    for(int i = 0; i < to_alloc; i++) {
        test_array[i] = random_val;
    }

    /*Here we recursively allocate more memory...*/

    /* Here we check if the memory allocated above */
    /* still has the same value. */
    for (int i = 0; i < to_alloc; i++) {
        if (test_array[i] != random_val) {
            std::cerr << "Memory checking error!" << endl;
            assert(false);
        }
    }

    /* Now we FREE the memory allocated above.*/
```

```
    delete[] test_array;
```

Ok, so we allocate and free arrays of integers using `new` and `delete`, but how do we get the code to exercise our allocator, and not the standard C++ allocator? This is done by **overloading** the `new` and the `delete` operators. The following is a simplified excerpt from program `memtest`; for clarity, all error checking and exception handling have been removed:

```
// When the test program calls 'new',
// it will use the following code to allocate the memory.
void* operator new(std::size_t sz) {
    cout << "operator 'new' called, size = " << sz << endl;
    void *ptr = our_allocator->Malloc(sz); // This is our allocator!!
    return ptr;
}
// Here we overload the 'delete' operator.
void operator delete(void* ptr) {
    cout << "operator 'delete' called" << endl;
    our_allocator->Free(ptr); // This is our allocator!!
};
```

That's pretty cool, isn't it?

Fibonacci Buddy-System Memory Allocation:

> **Note:**
>
> Feel free to initially skip this section; you can come back to it later, after Milestone 1.

Buddy-system allocators allocate memory in predefined segment 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 segment sizes makes the management of free memory segments very easy. The allocator keeps an array of *free lists*, one for each allowable segment size. Every request is rounded up to the next allowable segment 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 Segments: If the free list is empty (i.e. there are no free memory segments of this size,) a larger segment is selected (using the free list of some larger segment size) and split. Whenever a free memory segment is split in two, one segment 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 segment of size 3 available (i.e. the free list for size 3 is empty). The same holds for size-5 segments. The next-size available segment is of size 13. The allocator therefore selects a segment, say B, of size 13 (after deleting it from the free list). It then splits B into two segments, B_L of size 5 and B_R of size 8. Segment B_R is added to the size-8 free list. Segment B_L is further split into B_{LL} of size 2 and B_{LR} of size 3. Segment B_{LR} is returned by the request, while B_{LL} is added to the size-2 free list.

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

[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.

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

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

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

Finding and Coalescing Buddies: The buddy system performs two operations on (free) memory segments, *splitting* and *coalescing*. Whenever we split a free memory segment 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 segment, say B_X with its buddy, it is first necessary to know (a) what size the free segment is, and (b) whether the free segment is a right or a left segment. 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 segments can be coalesced into a single, larger segment. Note that we need to be able to infer whether the larger segment is a right or a left segment 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 segment (or better, its Fibonacci number index) is stored with the segment. In order to maintain the Left/Right information of segments, two bits are stored with each segment. We call them the *Left/Right* bit and the *Inheritance* bit.

Whenever we split a segment, 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 segments, the *Left/Right* bit of the new segment is equal to the *Inheritance* bit of the left child, and the *Inheritance* bit of the new segment 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 segments themselves to store the free-list data. For example, the first bytes of each free memory segment would contain the pointer to the previous and to the next free memory segment of the same size. The pointers to the first and last segment in each free list can easily be stored in an array of pointers, two for each allowable segment size.

Note on Segment Size: If you decide to put management information into allocated segments (e.g. the size, as described above), you have to be careful about how this may affect the size of the

allocated segment. For example, when you allocate a segment of size 5, and add an 8-word header to the segment, you are actually allocating a 5 segments +8 Word, which requires a segment 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 C Standard Library `malloc()` command. Don't forget to free this memory when you release the allocator (i.e. in the destructor).

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

Initializing the Free List and the Free Segments: You are given the size of the available memory as argument to the constructor of class `MyAllocator`. The given memory size is likely not a Fibonacci number. You are to partition the memory into a sequence of Fibonacci-number sized segments and initialize the segments and the free list accordingly.

The Assignment

You are to implement (in three steps) a buddy-system memory manager that allocates memory in segments 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++ class `MyAllocator`, which consists of a header file `my_allocator.hpp` and an implementation file `my_allocator.cpp`. (A copy of the header file and a rudimentary preliminary version of the `.cpp` 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 segments 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 modify the program `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. Most of the code for `memtest` is already present in file `memtest.cpp`.
- 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 basic 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 `MyAllocator::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.)

- Compiling the code is very simple, just type `make`. Type `make clean` to delete all the object files (`.o` files).

- Currently, the `makefile` is set up for the CLANG C++ compiler on the Mac. If you plan to use GNU C++ instead, comment the line `C++ = clang++` and uncomment the line `C++ = g++` in `makefile`. Our Linux machines use the GNU C++ compiler.

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 constructor of class `MyAllocator` reserves a given portion of memory for the allocator using the C Standard Library `malloc()`. Every call to `MyAllocator::Malloc()` cuts off the requested amount of memory from the remainder of the reserved memory. The call to `MyAllocator::Free()` does not free any memory. When all the memory has been used, and no more memory is left, subsequent calls to `MyAllocator::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**. This allocator manages segments of memory that are multiples of the `basic_block_size` (a parameter to the constructor). Initially, the free-list contains one large segment that covers the entire memory available to the allocator. As memory is requested using the `MyAllocator::Malloc()` call, this segment is then broken up into used and free segments, where the free segments are maintained in the free-list. When a used segment is freed, it is returned to the free-list.

For this you need a data structure to manipulate used and free segments, and to add and remove free segments to and from the free-list. Each segment (free or used) needs a so-called *Segment Header* that contains information about the segment. Such information can be whether the segment is used or free, the size of the segment, and maybe pointers to

previous and next segments in the free-list if needed. We define segment headers as Class `SegmentHeader`, and we represent the free-list as Class `FreeList`. Both segment headers and free-lists are defined in file `free_list.hpp` and `free_list.cpp`.

```
class FreeList {
private:
  /* --  YOU MAY WANT TO ADD YOUR OWN DATA STRUCTURES HERE. */
public:
  bool Remove(SegmentHeader * _segment);
  /* Remove the given segment from given free list.]
     The segment is represented by its segment header. */
  bool Add(SegmentHeader * _segment);
  /* Add a segment to the free list.]
     The segment is represented by its segment header. */
}
```

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

You are to store the segment header at the beginning of the beginning of the its segment. For free segments this is straightforward. For used segments this means that the user's memory starts after the segment header at the beginning of the segment.

Note that, given the segment headers and the basic-block size, the amount of allocated memory is different than the memory requested by the user. Say the user wants to allocate `x` bytes, then the segment must contain the header as well, thus leading to the allocator needing to allocate `x + sizeof(SegmentHeader)` bytes. Because the allocator manages memory in multiples of the basic-block size, it needs to round up to the next multiple of the basic-block size.

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 segments into smaller segments, and (3) coalescing of small free segments into larger free segments. As a result, you will have a full-fledged Fibonacci Buddy System Allocator.

What to Hand In, for each Milestone

- **Milestone 1:** You are to hand in three files, with names `my_allocator.hpp` and `my_allocator.cpp`, which define and implement your memory allocator, and `memtest.cpp`, which implements the main program. (Ignore the files `free_list.hpp` and `free_list.cpp` in the package. They will be used for future milestones.)

- **Milestone 2:** You are to hand in an updated version of the three files submitted for Milestone 1. **In addition**, you are to hand in the code for the free-list management, which shall be realized in files `free_list.hpp` and `free_list.cpp`.

- **Milestone 3:** In addition to updated versions of files from previous milestones, 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 very tedious. 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: 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.