Assignment 3 Design Document: Memory Allocation

Andy Nguyen (anhnguye@ucsc.edu)
Matthew Luu (mluu2@ucsc.edu)
Matthew Musselman (mmusselm@ucsc.edu)
CMPS111

May 28, 2012

1 Goal / Purpose

The purpose of this project is to implement and test a memory allocation user library that uses several different allocation mechanisms. This library will be used to test several memory allocation workloads.

2 Available Resources

The available resources we have for this project includes the GNU C compiler (gcc), the available system calls in Minix, and the dynamic memory allocation functions in C (malloc, realloc, calloc, free). Some of these resources that are integral to the completion of the project are discussed.

2.1 void *malloc(size_t size)

This function allocates uninitialized storage with size specified by size. If the allocation succeeds, then a pointer to the first byte in the allocated space is returned. However, the pointer return is null if malloc() was unsuccessful.

3 Design

The implementation of this project is divided into three major components, each reflecting one of the necessary memory allocators: the buddy, slab, and free-list allocators. After implementing those, the memory allocator functions are implemented to serve as a wrapper and call the appropriate allocator function. The object files that are produced during compilation of the functions are packaged together in a static library. The resulting library is a ".a" file. There are global variables to initialize for the entire memory allocator: a global array

to keep track of all the initialized allocators, and a global integer to indicate the next free allocator ID.

3.1 int meminit(long n_bytes, unsigned int flags, int parm1, int *parm2)

This function is responsible for initializing a new requested allocator. The variable n_bytes signifies the size of the requested memory, in bytes. Flags will signal the type of allocator requested, based on predetermined mappings. Parm1 and parm2 will vary depending on the allocator type requested. For the buddy allocator, parm1 represents the minimum page size to determine the extent of the bitmap region. The slab allocator uses the value passed into parm1 in order to indicate the number of pages from which a batch of objects is to be allocated, and parm2 points to an array of object sizes that the allocator should track. For the free-list allocator, neither of these parm values are used.

Implementation of this function is fairly simple, as is the other two memory wrapper functions.

1. Create a switch that will branch depending on the value passed in flag.
2. If the flag is 0x1, a buddy allocator is requested. Call the buddy initialization function.
3. On a flag value of 0x2, call the slab allocator initialization function.
4. If the value of the flag is 0x4 ORed with any of the suboptions (0x0, 0x08, 0x10, 0x18), then call the free-list allocator initialization function.
5. After calling the appropriate initialization function, increase the value of the global integer by 1. Return the current value of the global integer added with 99. This offset is added to ensure that the allocator ID will not conflict with any reserved numbers (such as 0 for NULL).

3.2 void *memalloc(int handle, long n_bytes)

Whenever a user program requests space to be allocated for an object, this function will return a pointer that references a section of free memory, based on the allocator type of the memory. Implementation is as follows:

1. First, check the value of the handle. If it is invalid, that is, if it is below 100 or over 511, print an error and return NULL.
2. Convert the handle into an index recognized by the allocator array by subtracting 100 from the handle.

3. If the memory specified by the index in the array is uninitialized, then print an error and return NULL.

4. Extract the value of the flag from the memory. It should be the first integer (4 bytes) of the memory.

5. Create a switch that will branch depending on the value passed in flag. Call the appropriate memory allocation function.

6. If the flag in the memory is invalid, then print an error. Upon exiting the switch, return a NULL pointer.

3.3 void memfree(void *region)

A pointer referring to an object is passed into memfree(), which will then call the appropriate memory freeing function. The function procedure is as follows:

1. Iterate through the allocator array.
   Find the smallest positive difference between the address of the beginning of the array and the address the pointer refers to. This will ensure that the correct allocator is picked. A negative value will indicate that the object is stored BEFORE the header of the current allocator. If the difference between the beginning of one allocator and the address of the pointer is larger than another, it will mean that there is enough memory to store information from at least two allocators, and the current one cannot be the one keeping track of the object.

2. Extract the flag from the chosen memory in the array.

3. Create a switch that will branch depending on the value passed in flag. Call the appropriate memory allocation function.

4. If the flag in the memory is invalid, then print an error and return from the
   function.

3.4 Implementing the Buddy Allocator

This type of allocator is one of three core components of the memory allocator. Generally, a minimum page size is specified. Then, depending on the size of the requested piece of memory, biparititions of the memory is made until the smallest size possible chunk of memory can be allocated for the object. Upon freeing the piece of memory, the other half of the bipartition, or buddy, is checked. If that piece is also free, then merge the pieces together and mark the

bit representing the parent of the two pieces as free. Implementation of this allocator is split up into three essential functions: initialization, allocation, and deallocation.

3.4.1 Buddy Initialization

Implementation for a buddy initializer is as follows:

1. The number of pages that will fit in the memory is determined by taking the total number of bytes requested and dividing by the size of one page.

2. Check to see that the total number of pages is a power of 2. This is done using a fast algorithm. If it is not, then print an error and return a NULL pointer.

3. One bit will be required to keep track of each page at each level. The number of total bits will be roughly double the number of pages. Dividing this number by 8 will give the total number of bytes required to store the bitmap in memory.

4. Allocate the required memory from the system using malloc(). The total size of the memory is the size of three integers plus the size of a char multiplied with the sum of n_bytes and the bitmap size.

5. If the pointer fo the memory is NULL, then malloc() has failed. Print an error message and return a NULL pointer. Otherwise, return the pointer to the newly initialized piece of memory.

3.4.2 Buddy Allocation

The implementation of the buddy allocator is a little unique from the traditional methods. It determines the appropriate level of the bitmap to begin looking for free blocks to ensure minimization of internal fragmentation. Then, each bit in that level is iterated through to search for free space. If all the bits are 1, then all the memory is allocated for other objects. When a free space is found, the parent bits that refer to that particular space is marked as allocated. The implementation is as follows:

1. Extract the number of pages and the size of each page in the memory. These values will be used to determine where and how the address of the returned pointer will be chosen.

2. Determine the number of levels in the bitmap by finding the log2 of the number of pages of minimum size in the memory.

3. Determine the level at which to begin searching for available space. To do this, start with the minimum page size. As long as the page size is not large enough to satisfy the requested memory, double the page size and subtract one from the maximum number of levels. After a sufficient number of iterations, if the level to begin search is negative, this means that the requested size is larger than the total amount of bytes in the memory. In that case, return a NULL pointer.

4. Begin iterating through the bits in the level. If the bit is zero, then space is available. Iterate through the bits that are children of the current bit and mark them all as used. This is to prevent incorrectly reporting free space for smaller objects when the space is being used for the current object.

5. Starting with the current bit representing the free page, traverse back up the bitmap tree, marking each respective parent as currently in use.

6. Determine the address of the beginning of the free page. Very simply, this is equal to the beginning of the slab of memory plus the size of the bitmap plus the offset of the page in memory. The value of the offset is found by taking the product of the page size at that level (the minimum page size multiplied by the number of those pages that fit inside one page at the current level), and the index of that page at the current level (a number between 0 and 2^level - 1, inclusive).

7. return the pointer that references the address of the free page.

3.4.3 Freeing Objects in Buddy Allocators

How dafuq do you do this? ARGGGHHHHH

3.4.4 Helper Functions for the Buddy Allocator

A few helper functions are implemented for the buddy allocator to perform certain repeated routines. A short description of them are provided:

1. lg2 - This function assumes that the number passed in is a power of 2. Because of this, finding the log2 of the number is O(log n), as the number of times the number must be divided by 2 to reach 1 is equal to the log2 of that number.

2. bit_is_free - This function checks if a particular bit in the bitmap tree is a 0 or 1. On a 0, the function returns true (a positive nonzero number).

3. setbit - Sets a particular bit in the bitmap to a state specified by the value of state.

4. set_lower_levels - Sets the children of the specified bit in the bitmap tree to the same value as the state variable.

3.5 Implementing the Slab Allocator

3.5.1 Slab Initialization

3.5.2 Slab Allocation

3.5.3 Freeing Slab Objects

3.6 Implementing the Free-List Allocator

3.6.1 Free-List Implementation

3.6.2 Free-List Allocation

3.6.3 Free-List Deallocation

3.7 Creating the Object Library

Creating the object library is a fairly simple process:

1. When compiling the memory allocator, configure the compiler to only produce the object files (.o).

2. Create an archive file with the object files by calling ar with the following command:

```
ar -cvr libmem.a [object files]
```

3. When compiling user programs that use the object library, configure the compiler to specify the directory of the object library and the name of the library name:

```
gcc -o [fname] [sources] -L [dir of lib] -l [name of lib]
```

4 Testing