Brandon Buckley

4/7/15

**Implement and Evaluate a User-Space Memory Allocator**

1. **Data Structure Design**

The free-list nodes consist of doubly linked nodes for ease-of-use. It also contains a variable, *memFree*, to hold the value of the number of free bytes contained in the node. The size of *\_\_nodeFree* is is 24 bytes.

// Free List Node

typedef struct \_\_nodeFree {

struct \_\_nodeFree \*next; // pointer to next node

struct \_\_nodeFree \*prev; // pointer to prev node

int memFree; // size of free memory in node

} \_\_nodeFree;

The malloc header consists of a variable, *size*, which represents the number of bytes allocated below it. The variable, *magic*, is used as a sanity check to verify that once it is freed, the correct location in memory is being referenced.

// Malloc Header

typedef struct \_malloc\_h {

int size; // size of allocated memory

int magic; // sanity check number

} \_malloc\_h;

1. **Function Implementation**

*psumeminit( int algo, int sizeOfRegion )*

This function is called one time by the process that wants to use my memory allocator. It will initialize the swap space of size *sizeOfRegion.*  If *sizeOfRegion* modulus *getpagesize()* is not equal to zero, I round up to the nearest page size. I then call *mmap* to initialize my swap space and save the returning pointer to my head node, *head*, of my free-list. After initializing my head node, I return.

*psumalloc( int size )*

This function first finds a node by the best-fit or worst-fit algorithm implementation specified in psumeminit. Once a free node is selected, *size* number of bytes is allocated and the free node is divided and copied down. If not enough contiguous space is found in the free-list, the function returns NULL, otherwise, it returns a pointer to the newly allocated starting address in memory.

*psufree( void\* ptr )*

This function frees the memory object that *ptr* points to. If the pointer is NULL then it returns -1. A sanity check is then performed to verify that the magic number matches the expected value. If this is true, then we have confirmed that a valid address has been received. Next, the free-list is traversed until a location is found that keeps the list numerically sorted. A free node then replaces the malloc header and is placed in the free-list at the appropriate position. Lastly, the newly added free node is checked with nodes above or below to be coalesced when necessary.

*psumemdump( void )*

This function prints the free-list to the standard output (terminal).

1. **Three Data Structure Options**
2. Singly Linked-List – This data structure could be used for simplicity. It would also equal the same size of the malloc header so malloc and free would causes less problems.
3. Doubly Linked-List – This data structure was used because of the functionality of the previous pointer. This was utilized in coalescing the linked list making it easier to traverse the free-list nodes. This can cause a problem when mallocing and freeing because the malloc header and the free node sizes do not match up.
4. Red-Black Augmented Binary Search Tree – This data structure will keep the nodes sorted as a linked list can. The advantage of the Red-Black augmentation is that it will keep the tree balanced and searching will be faster.
5. **Design Issues**

The only issue with the design and data structure I choose is that my free node is larger than my malloced node. This causes an issue when mallocing or freeing because the free node can overwrite part of the malloced header. I either have to change my list to a singly linked list, or I change the minimum size of allocation in the test files to 16 bytes (16 + 8 bytes = 24 bytes === sizeof(freeNode)).