CSCI 4061: Introduction to Operating Systems (Spring 2014)

Released: March 6, 2014 Due: March 28, 2014

1 Pre-requisites

It is assumed that you have good knowledge of C. Defining good: you can write, compile, run and debug C code.

2 Introduction

This assignment will shed a light on how memory space is dynamically allocated and freed in Unix. You will also learn how to check some common memory problems like buffer overflow, segmentation fault and memory leaks.

3 Problem

You will develop your own **memory management library** in this assignment.

C library is a set of related C functions that perform certain functionalities. These functions are called within another C program. Usually, C library is called by including the .h header file that contains the functions headers. This header file is also associated with one or more .c files that contain the actual implementation of the library functions.

Memory management can allocate memory space to programs, free/deallocate memory space, assign values to allocated memory space and handle errors.

3.1 Tasks

(a) **Initial memory space** by implementing the following function:

```
int mm_init(unsigned long size)
```

This function applies for a chunk of memory pool from the system by using `malloc` function once. If this memory management library is used by other programs, this function should be called first in their programs. And afterwards, all the memory space they require should be allocated from this memory pool and return to this memory pool once freed.

This function will initialize the data structures for your library functions. Also it takes an input parameter size which represents the total amount of memory (in bytes) available to the program. You can use `malloc` once here to create the memory pool of the size given by size.

Return Value: It should return 0 on success, else it would return -1.

(b) **Allocate memory space** to programs by implementing the following fuction

```
char *mm_alloc(unsigned long nbytes)
```

This function is similar to `malloc()`. It takes the number of characters (char) you want to allocate as the argument and returns the pointer to the beginning of the allocated memory if successful. This memory is allocated from the initial "memory pool" created by using the `mm_init()` call with the **first fit algorithm**(explained in §3.3). If sufficient memory is not available to satisfy the request, a NULL pointer will be returned and an error message should be printed. Pay attention, `malloc` is NOT allowed, except for internal data structures use in this function.

Return Value: A pointer to allocated memory on success, else NULL is returned. If no sufficient memory is available to satisfy the request, it should print an error message: “Request declined: no enough memory available!” to the log file **out.log**. (This log file locates in the same directory as the mm.c file)

(c) **Free allocated memory** to memory pool

```
int mm_free(char *ptr)
```

This function is similar to `free()`. The argument is the pointer you want to de-allocate. If the pointer that is going to be freed is previously returned by `mm_alloc()`, it’s a valid pointer. Otherwise it’s invalid. The function must ensure that **only the memory space referred by a valid pointer is being freed** back to the memory pool. The size of the freed memory should be the same as the memory allocated to this pointer. If the pointer is invalid, don't free anything. (In real life, trying to free an invalid pointer may cause core dumped!)

Defragmentation Freeing also causes a search of all the free blocks, to find the proper place to insert the block being freed. If the block being freed is adjacent to a free block on either side, it is coalesced with it into a single bigger block, so storage does not become too fragmented. Method of determining the adjacency is decided by your data structure describing those free blocks.

Return Value: The function should return 0 if a valid pointer is freed, and -1 otherwise, in which case it should print an error message: “Free error: not the right pointer!” to the log file **out.log**.

(d) **Clean up**

```
void mm_end(unsigned long *free_num)
```

This function will first count the total number of free blocks at the state when this function is called. The total number of free blocks is returned by `free_num`. This function will also clean up any data structures associated with your library and free the memory space that are allocated for your internal data structure by using `malloc` (if there are any). It will also free the memory pool that is allocated by using `malloc` in the `mm_init` function (Your library itself can't cause memory leaks!). This should be called just before the end of the program (after all the other functions defined in your memory management library).

3.2 Data Structure

Since the memory space managed by our function may not be contiguous, whatever the allocated space or the free space, you should design and maintain appropriate data structure to support the tasks listed above. Our grading will have no preference on the choice of data structure, as long as it can solve the problem.

3.3 First Fit Memory Allocation Algorithm

In the first fit memory allocation, a memory request is met by finding a free memory block starting from the beginning of the memory pool which can meet the request. In other words, the selected memory area for allocation is the first memory area with sufficient size. Consider following example, say there is total of 35 bytes of memory is available, and first fit memory allocation algorithms is used. Initially, we have 2 allocated memory blocks A, C shown as shaded region in table 1. And the blank region represents two free blocks B, D. The sizes of block A, B, C are all 10 bytes. And the size of block D is 5 bytes. Then we try to apply for 5 bytes of memory space for a variable `test` by using `mm_alloc` as shown below:

```
test=mm_alloc(5);            //test=110
```

The memory state after the allocations is show in table 2. We can see that the algorithm will search for the first block that is big enough for 5 bytes, which is B. Since the size of block B is larger than the required size, it is split and the proper amount (Block E) is returned to the user while the residue (Block F) remains free.

Table 1. Initial Memory Map

Memory Block	A	B	C	D
Starting Address	100	110	120	130

Table 2. Memory Map after mm_alloc for test

Memory Block	A	E	F	C	D
Starting Address	100	110	115	120	130

3.4 Extra Credit

To get extra credit, your memory management should also provide functionalities to check common memory problems: buffer overflow and memory leaks.

(a) **Assign values** to allocated memory space and **check buffer overflow**

```
int mm_assign(char *ptr, char val)
```

This function is actually a wrapper for the statement `*ptr = val`. This will ensure that only a valid pointer (part of a currently allocated block) is being modified in order to catch buffer overflows and bad memory writes. Here, a "valid pointer" refers to a pointer in an unfreed part of memory which was previously returned by `mm_alloc()`.

Return Value: If the pointer is valid, the function should return 0. It should return -1 and print an error the log file **out.log** in case that `ptr` is not in a currently allocated region: “Buffer Overflow: Try to access illegal memory space.”

(b) Implement a function to **check memory leaks** errors in the program.

```
unsigned long mm_check()
```

When you call this function, it should check whether there are any memory leaks in the program. If so, how many blocks are leaked?

Return Value: It should return 0 on success. Otherwise it would return the number of leaked memory blocks.

3.5 Assumptions

1. You can assume that `mm_alloc()` only allocates char pointers, and `mm_assign()` only assigns char values.
2. Since your functions are meant to be used as library functions, do not make any assumptions about how many times they can be called.
3. Programs calling this library would not make any calls to system’s `malloc` and `free`, and will do all memory allocations through our memory manager.

4 Compile and Link

Your code will be a library and will be linked with another program so you do not need to write a main or try to create an executable except for testing your work. During grading, we are going to provide the main function. We've already given the mm.c and mm.h file which will include some base code. For successful linking you should include mm.h header file in all the C files that use these memory management functions. To do this, add the following line to your Makefile. This line will produce a mm.o object file, which will be used by any program that use our memory management library.

```
mm.o: mm.c mm.h
	gcc -c mm.c -o mm.o
```

In order to make your code compile successfully with main function, please add the following line in your Makefile.

```
test: mm.o test.c
	gcc -o mm test.c mm.o
```

These two lines means there are functions in test.c using our memory management library. So we must link the object file mm.o to generate our executable "test". Assume test.c is the c file which will contain the main function provided by TAs for the purpose of testing and grading your assignment. So including this line will help our grading. Thank you!

Of course, you can write a test file by yourself to test the correctness of your library. But please don't include it in your submission version. Thank you!

5 Example

Here's an example of how your functions implemented in memory management library will be used.

```
#include “mm.h”

int main()
{
   char *A, *B, *C, *D;
   unsigned long free_num, leaks;

   mm_init(10000);        //create the memory pool and initialize data structure

   A= mm_alloc(10);
   B= mm_alloc(1000);
```

```
    C= mm_alloc(200);

    mm_free(B);
    D=mm_alloc(20);
    mm_free(C);      //After this block is freed, it will be coalesced with
                       free blocks of both sides

     for (i=0;i<=20;i++){
        mm_assign(D+i,(char)i);  //When i=21, should print “Buffer Overflow: Try to
                                   access illegal memory space.” to out.log
    }

    mm_free(D+10); //Trying to free the memory space pointed by an invalid pointer.
                     Print “Free error: not the right pointer!” to out.log

    leaks=mm_check();
    printf("Total memory leaks:%lu", leaks);
    // leaks=2, A,D are not freed

    mm_end(&free_num);
    printf("free blocks number:%lu", free_num);
    //free_num=1

    return 0;
}
```

6 Few Do’s

Technical:

1. Program must be written in C.
2. Make sure your program compiles using gcc compiler on any of the CSELab ubuntu machines (you can write a test file by yourself to test the correctness of your library). We will use the same development Environment. No points will be given if program does not compile with our test c file.
3. You must also provide a Makefile along with the source code. Lines specified in §4 Compile and Link should be included in your Makefile.
4. You should put the log file out.log in the same directory as mm.c file.

Non-Technical:

1. Assignment can be done in group of at most two head counts. Please include your names at the top in README file.

2. It is expected that you do not take any unfair advantage of any ambiguity in the problem statement. In that case, please confirm with Instructor or Teaching Assistants before you proceed with any approach. It might not make sense coming up with a solution that does not help hone your skills in the expected area.

7 Submission Guidelines

The code and Makefile should be packaged and submitted as a tar.gz file. Please do not use any other forms of compression. Name of the TAR file should be csci4061pa3. This TAR should contain a directory named csci4061pa3. This directory should contain the source files and the Makefile as described in section 6 above. Feel free to include any .c or .h files that you feel are necessary.

Pay Attention, only one submission per group is needed. Thank you!

8 Grading

Maximum you can get for this assignment is 110%. Point distribution is as follows:

1. Following Submission Instructions and a sensible README: 5 points
2. Documentation and style: 15 points
3. Functionality and Error Handling: 80 points
4. Extra Credit: 10 points