CS342 Operating Systems - Fall 2019
Project #3
Dynamic Memory Management

Assigned: Nov 29, 2019, Friday.
Due date: Dec 15, 2019, Sunday, 23:55.

- *Submit through Moodle. Make sure you start submitting one day before the deadline. You can overwrite your submission as many times as you wish. Late submissions will not be accepted (no excuse; no email will be accepted).*
- *You will develop your project in Linux/C. The project can be done in groups of 2.*

1 Assignment

1.1 Part A - Memory Management

In this project you will implement a thread-safe memory management library. It will be used by a multithreaded application to allocate memory dynamically. Your library will be a static library (ending with .a).

Your memory management library (**libmemalloc.a**) will manage a given chunk of free memory. That means the library will allow a multi-threaded application to allocate and free memory from the given chunk. The chunk of free memory (a contiguous chunk of free space) will be created by the application, and a pointer to the chunk and the size of the chunk will be passed to the library. Then, the library will manage the chunk space. The application can make requests to the library to allocate and deallocate memory.

In your library implementation (**memalloc.c**), you will use *hole-list* approach (dynamic storage allocation). You will implement 3 methods for searching: first-fit (0), best-fit (1), and worst-fit (2).

You library will implement a set of functions that can be called by an application linked with your library. The prototypes of these functions will be put into a header file (memalloc.h). These prototypes are called the interface of your library. You can change the bodies of these functions, but not the prototypes. The library header file will be included (#include) by an application that would like to use your library to allocate memory dynamically. The application needs to be linked with your library as well (how this can be done is shown in the Makefile below). Your library will implement the following functions that can called by an application. In your library, can implement some other functions that you may need internally in your library, but those other function are not be called the application.

- **int mem_init (void *chunkptr, int size, int method).** This initializes the library to manage the provided chunk space. The chunkptr points to the chunk. The chunk is created outside of the library (i.e., by the application). Application can use **sbrk** function (see the example in the app.c below) or **malloc** to create the chunk. If the application uses malloc to create the chunk, it will be the only place the application uses the malloc function. After that the application should use your library functions for dynamic storage allocation. The size parameter is the size of the chunk to manage (to allocate memory from). The method parameter specifies which hole search method will

be used. The function will return -1, if the is an error encountered, otherwise 0 (success) will be returned.

- **void *mem_allocate (int size).** Allocate memory for an object (i.e., for a request). The memory should be allocated from the chunk specified with mem_init. The size of allocation is given as a parameter (request size). The allocated memory amount should at least the requested size (the allocated amount inside the library can be more depending on the your allocation method). The function will returns a pointer to the beginning of the allocated usable space, so that the application can use the pointer to put data (object) the allocated space.

- **void mem_free (void ptr).** Free the allocated space. The space to be freed is pointed by ptr.

- **void mem_print ()**. Print the current state of the memory chunk: which portions are allocated, which portions are free (in sorted order with respect to address). The format for printing out is up to you. This can be used for debugging and testing purposes.

An application app.c will be compiled and linked with your library as follow:

```
gcc –Wall –o app  app.c -L.  -lmemalloc
```

We can run the application as follows, for example. The chunk size is 1024 KB (kilobytes) in this example.

```
./app 1024
```

We will develop test applications to test and stress your library implementation. You should also develop test applications to test your library.

1.2 Part B - Experimentation and Report

Do some timing and space (fragmentation) experiments and write a report about the results. Experiment with various hole search methods (first, best, worst). Put tables and/or figures into your report showing the results. Try to interpret the results.

2 Sample Code

Below we provide some sample content for your Makefile, library (memalloc.c), header file (memalloc.h) and an application (app.c).

2.1 Makefile

```
all: libmemalloc.a app

libmemalloc.a:  memalloc.c
        gcc -Wall -c memalloc.c
        ar -cvq memalloc.a memalloc.o
        ranlib libmemalloc.a

app:    app.c
```

```
	gcc -Wall -o app app.c  -L. -lmemalloc

clean:
	rm -fr *.o *.a *~ a.out app
```

2.2 memalloc.h

```
#ifndef MEMALLOC_H
#define MEMALLOC_H

int mem_init(void *, int);
void *mem_allocate(int);
void mem_free(void *);
void mem_print (void);

#endif
```

2.3 memalloc.c

```
#include <stdlib.h>
#include <stdio.h>

// printfs are for debugging; remove them when you use/submit your library

int mem_init (void *chunkpointer, int chunksize)
{

	printf("init called\n");

	return (0);			// if success
}

void *mem_allocate (int objectsize)
{

	printf("alloc  called\n");

	return (NULL);		// if not success
}

void mem_free(void *objectptr)
{
	printf("free called\n");

	return;
}

void mem_print (void)
{
	printf("print called\n");
```

```
        return;
}
```

2.4 app.c

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include "memalloc.h"

int main(int argc, char *argv[])
{
        void *chunkptr;
        void *endptr;
        char *charptr;
        int ret;
        int i;
        int size;
        void *x1, *x2, *x3;         // object pointers

        if (argc != 2) {
                printf("usage: app <size in KB>\n");
                exit(1);
        }

        size = atoi(argv[1]);   // unit is in KB

        // allocate a chunk
        chunkptr = sbrk(0); // end of data segment
        sbrk(size * 1024); // extend data segment by indicated amount (bytes)
        endptr = sbrk(0);           // new end of data segment

        printf("chunkstart=%lx, chunkend=%lx, chunksize=%lu bytes\n",
             (unsigned long)chunkptr,
             (unsigned long)endptr, (unsigned long)(endptr - chunkptr));

        //test the chunk
        printf("---starting testing chunk\n");
        charptr = (char *)chunkptr;
        for (i = 0; i < size; ++i)
                charptr[i] = 0;
        printf("---chunk test ended - success\n");

        ret = mem_init(chunkptr, size);
        if (ret == -1) {
                printf("could not initialize \n");
                exit(1);
        }

        // below we allocate and deallocate memory dynamically
        x1 = mem_allocate(600);
```

```
        x2 = mem_allocate(4500);
        x3 = mem_allocate(1300);

        mem_free(x1);
        mem_free(x2);
        mem_free(x3);

        return 0;
}
```

3 Submission

Put all your files into a directory named with your Student Id. In a README.txt file, write your name, ID, etc. Include a Makefile. Then tar and gzip the directory. For example a student with ID 21404312 will create a directory named "21404312" and will put the files there. Then he will tar the directory (package the directory) as follows:

```
tar cvf 21404312.tar 21404312
```

Then he will gzip the tar file as follows:

```
gzip 21404312.tar}
```

In this way he will obtain a file called 21404312.tar.gz. Then he will upload this file in Moodle.

Late submission will not be accepted (no exception). A late submission will get 0 automatically (you will not be able to argue it). Make sure you make a submission one day before the deadline. You can then overwrite it.

4 Tips and Clarifications

- Your library will be a static library (.a). See Makefile provided about how a static library can be created.
- Do not use malloc or similar existing allocation routines in the implementation of your library. You will manage the chunk space with your own functions and implementation.
- You will keep the management structures also in the chunk space itself. You can use modest amount of global variables (a static array, etc.) in your library implementation. Most book-keeping information should be part of the chunk itself, and you should not use malloc to create any dynamic structure.
- Minimum chunk size is 32 KB and maximum chunk size is 32 MB.
- Minimum request size is **128 bytes** and maximum request size is **2048 KB**.
- Develop your program in a 64-bit machine (most machines are 64-bit) and 64-bit OS. In that case memory adresses (pointers) are 64 bits long (8 bytes). Hence we use *long int* to refer to a memory address as an integer.
- Do not forget synchronization.