CS342 Operating Systems – Spring 2022
Project #3 – Implementing a Dynamic Memory Allocator

Assigned: Mar 31, 2022.
Due date: Apr 17, 2022, 23:59. Document version: 1.0

*This project can be done in groups of two students. You can do it individually as well. You will use the C programming language. You will develop your programs in Linux. As always, the project is **for learning** and **acquiring skills**.*

Objectives: To learn and practice with dynamic memory allocation, memory mapping, bitmaps, heap, internal and external fragmentation, pages and virtual addresses, process address space, virtual memory layout, virtual memory regions, backing file or store, resident set size, physical memory usage, virtual memory usage, protection bits, bit-wise operations, manipulating bits, doing experiments, collecting and interpreting experimental data.

Part A (80 pts):

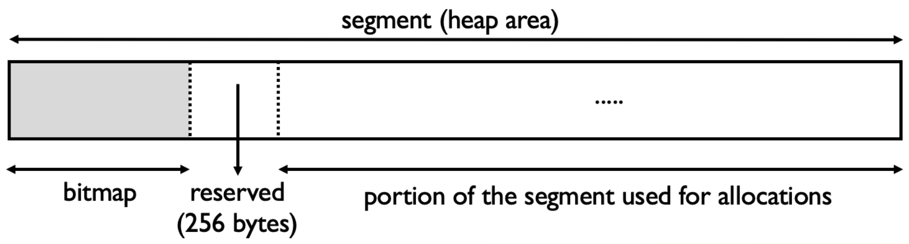
In this project you will implement a **dynamic memory allocator (dma) library**. An application will use your library, instead of `malloc()` and `free()` functions, to allocate and free memory dynamically from a virtual memory region. The library should be **thread-safe**. That means, multiple threads of an application may try to allocate memory at the same time, and this should not cause a problem. Your library will be called as **`libdma.a`** (*dynamic storage allocator* library).

Your library will first allocate and map a virtual memory region using the `mmap()` system call in the library initialization function. This virtual memory region will be a contiguous **segment** (an initially empty chunk of memory) in the process virtual address space, which will be managed by your library. Dynamic memory allocations to an application that is using your library will be done from this segment. The term segment here has nothing to do with the memory management technique called segmentation. This segment will be ***your* heap area** (different from the *ordinary* heap area that `malloc`() uses).

The size of the segment will be a power of 2, and therefore, a multiple of pagesize as well. The segment must also start at a **page** boundary (mmap may guarantee that, but to be sure you need to read the `mmap` manual page). We will assume the pagesize to be 4KB. We will consider the segment as a sequence of bytes (each byte has a virtual address from the virtual address space of the process). But our memory management unit will be a **word**. In this project, **word-size** will be **8 bytes**. The minimum size of an allocated block will be 2 words. The **allocation unit** will be two-words (16 bytes) as well. That means an allocated block can be *n* words long, where *n* is an even number (i.e., a multiple of 16 bytes).

You will keep track of the free and allocated blocks in the segment using a **bitmap** that will sit at the start of the segment (at offset 0 of the segment). A bitmap is a vector of bits. We will have **1 bit** for *each* **word** of the **segment** in the bitmap. A **free block** of size *n* words will be represented with ***n* consecutive 1's** in the bitmap. A **used (allocated) block** of size *n* words, however, will be represented with a **01** followed by **(*n*-2) 0's** in the bitmap (*n* >= 2). For example: a used block of size 2 words will be represented as 01. A used block of size 6 words will be represented as 010000. In this way, we will be able to detect the *boundaries* of the allocated blocks. A free block of size 2 words will be represented as 11 in the bitmap, and a free block of size 4 words will be represented as 1111.

The **size of the bitmap** will depend on the size of the segment. If the segment size is 2^{20} bytes, for example, then the number of bits needed in the bitmap will be $2^{20} / 2^3 = 2^{17}$ bits. It makes $2^{17} / 2^3 = 2^{14}$ bytes. The bitmap will include the bits even for the words storing the bitmap (for simplicity). After the bitmap, we will have 256 bytes **reserved** space (will not be used to store any information, but will not be free either). In your bitmap, you can encode the bitmap area, and the reserved space as used blocks. Right after the reserved space, the rest of the segment will be used for allocations (see figure below). Hence, the rest of the segment will contain allocated and free blocks. Allocated blocks will contain application data (structures, data values of different types, objects - not in the sense of object-oriented programming -, buffers, etc.). Besides the bitmap, there is no other meta-structure that will be kept on the segment.



Allocated block size will always be a multiple of double-word-size (16 bytes), even the application requests memory that is not a multiple of word-size. For example, if the application requests memory for 3 bytes, we need to allocate a block of size 16 bytes (2 words). If the application requests memory for 20 bytes, we need to allocate a block of size 32 bytes (4 words). If the application requests memory for 1020 bytes, we need to allocate a block of size 1024 bytes.

We may have *internal* **fragmentation**. For example, if we allocated a block of size 16 bytes for a request of 7 bytes, the internal fragmentation amount for this allocation will 9 bytes. We may also have *external* fragmentation.

The following is an example representation (**encoding**) in the bitmap for a portion of the segment.

$\underbrace{0100}\underbrace{111111}\underbrace{01}\underbrace{01}\underbrace{01000000}\underbrace{11}\underbrace{01}\underbrace{1111}\underbrace{010000}\underbrace{11}\underbrace{01}\underbrace{11}$

It indicates the existence of the following blocks in the segment: a used block of size 4 (words), a free block of size 6, a used block of size 2, a used block of size 2, a used block of size 8, a free block of size 2, a used block of size 2, a free block of size 4, a used block of size 6, a free block of size 2, a used block of size 2, a free block of size 2. The following bit stream is showing the block boundaries with spaces, and also indicates if the bit pattern *represents* a free block or an allocated (used) block, and the size of the block. Of course, in our real bitmap we will not have spaces like this between bits.

```
0100 111111 01 01 01000000 11 01 1111 010000 11 01 11
A    F      A  A  A        F  A  F    A      F  A  F
4    6      2  2  8        2  2  4    6      2  2  2
```

To **allocate** memory, you need to **search the bitmap** from left to right (towards *increasing* addresses) to find a large enough free block. A free block is represented with a sequence of 1's (but be careful about the fact that 01 indicates the start of a used block). You will stop the search at the first large enough free block, and allocate it. Next time, you will again start searching from the *beginning* of the bitmap (always). That means we are using *first-fit* strategy. If memory is requested for *k* bytes, you need to allocate memory of size the smallest multiple of 16 bytes larger or equal to *k*. For example, if request is for 70 bytes, you need to allocate memory for a block of 80 bytes. Therefore, you need to search for a free block of size 80. The search *algorithm*, i.e., scanning and checking of the bits in the bitmap for finding a free block, is up to you. You may try to make that search as efficient (fast) as possible.

Your **library interface** will be as follows (will be written in a header file called **dma.h**).

```
#ifndef DMA_H
#define DMA_H
int   dma_init (int m);
void *dma_alloc (int size);
void  dma_free (void *p);
void  dma_print_page(int pno);
void  dma_print_bitmap();
void  dma_print_blocks();
int   dma_give_intfrag();
#endif
```

You will implement these functions in a file called **`dma.c`**. This file will contain the **implementation of your library**. You can implement some additional functions as well if you wish for internal use in the library (not as part of the interface). You should not change the interface. Below we are explaining these functions.

- `int` **`dma_init`** `(int m)`: This function will initialize the library. As part of the initialization, the function will allocate a contiguous segment of free memory from the virtual memory of the process using mmap() system call. This segment will be your heap. Your library will manage it. The size of the segment will be 2^m bytes. For example, if `m` is equal to 16, then the segment size will be 64 KB (2^{16}). The value of `m` can be between 14 and 22 (inclusive). Hence, the smallest segment can be 16 KB long and the largest segment can be 4 MB long. The function will do other *initializations*, like initializing the bitmap. You can assume that `dma_init()` will be called by the main thread in a multi-threaded program, before any other thread is created. If initialization is successful, the function will return 0. Otherwise, it will return -1.

- `void *`**`dma_alloc`**` (int size):` This function will allocate memory for the calling application from the segment. The `size` parameter indicates the size (in bytes) of the requested memory to be allocated. The value of size can be any positive integer value (internally, however, we will allocate space that is a multiple of 16 bytes). A pointer to the beginning of the allocated space will be returned, if memory is successfully allocated. Otherwise, if the requested memory can not be allocated, for example due to lack of a large enough free block, NULL will be returned. If indicated size is too big, the library will not be able to satisfy the request and will return NULL in this case as well. By checking the return value, we can understand if a request could be satisfied or not.

- `void `**`dma_free`**` (void *p):` This function will free the allocated memory block pointed by `p`. From the pointer value (which is an `unsigned long int`) and the start virtual address of the segment, you can drive the *start bit position* of the block representation in the bitmap. From the bitmap encoding for the block, you can also drive the *size* of the block. The corresponding bits in the bitmap need to be set to 1. We assume a pointer (such as `void *`) is 8 bytes long. Similarly, a `long int` is also 8 bytes long. Therefore, we can consider pointer values as unsigned long integers.

- `void `**`dma_print_page`**` (int pno)`. This function will print a page of the segment to the screen using hexadecimal digits. The first page of the segment (starting at the address returned from `mmap()` call) will be considered as page 0. Hence, `pno` parameter can have a value in the interval [0, *SS*/*PS*), where *SS* is segment size and *PS* is the pagesize. Each output line will contain 64 hexadecimal characters. An example output can be as follows (use this format).

```
ffff8000ffffffff800000000000ffffffff55d0ffff…
…
…
```

- `void `**`dma_print_bitmap()`**. This function will print the bitmap to the screen as a sequence of 1's and 0's. Each line will have 64 bits. We will put a *space* character after each 8 bit sequence in the line, like below (use this format).

```
01000000 … 00000000 00000000 00000000 00000000
00000000 … 00000000 00000000 00000000 00000000
…
11111111 … 01000000 11111111 01000000 00000000
…
```

- `void` **`dma_print_blocks()`**. This function will print information about the allocated and free blocks. For each block it will print the start address using hexadecimal digits, and the size (in bytes) using both hexadecimal digits and decimal digits (in parentheses), in a separate line. It will also print at the beginning of the line if the block is allocated (A) or is free (F). The printing will be in the order of start addresses. You will also print information about the used blocks containing the bitmap and reserved space. An example output can be as follows (use this format):

  ```
  A, 0x00007ffff75e4000, 0x400 (1024)
  A, 0x00007ffff75e4400, 0x100 (256)
  A, 0x00007ffff75e4500, 0x200 (512)
  A, 0x00007ffff75e4700, 0x100 (256)
  F, 0x00007ffff75e4800, 0x300 (768)
  A, 0x00007ffff75e4b00, 0x20 (32)
  F, 0x00007ffff75e4b20, 0x50 (80)
  A, 0x00007ffff75e4b70, 0x230 (560)
  …
  ```

  Note that the printed addresses must be 16 hexadecimal digits long (64 bits, even though only 48 bits are used). They are real virtual addresses, not offsets into the segment. We are assuming that the machine architecture is x86-64. If the architecture is different, still the output should use 16 hexadecimal digits for the addresses.

- `int` **`dma_give_intfrag()`**. This function will return the internal fragmentation amount. Hence, in your library you need to maintain the total internal fragmentation amount while you are allocating memory to requests.

In the implementation of your library, you can use **global variables**. But keep in mind that multiple threads may call the library functions simultaneously, therefore accesses to global variables and bitmap, etc., need to be protected by use of Pthreads mutex **locks**. You can use one or more locks (it depends on your design). We should not have race conditions happening. Additionally, use of **`malloc()`** is **forbidden** in the implementation of the library.

In `dma_init`() function, the mmap() call will return a *pointer* pointing to the beginning of the mapped segment (i.e., it returns the *start* virtual address of the segment). Using that pointer, you can access any position in the segment in your library implementation. If `mmap()` fails in creating a mapping, it returns `(void *)-1`. You should check the return value. To get more information about `mmap`, type "`man mmap`" at command line.

To give an example for the usage of `mmap()`, we have a small program called **`mem.c`** shown below. It is mapping a 4MB memory region (our segment) and printing out the start address of the mapped region to the screen. In your library implementation, you can call `mmap()` as shown in this example.

```
#include <sys/mman.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main (int argc, char ** argv)
{
    void *p;
    int size;

    size = 4096 * 1024; // 4 MB
    p = mmap (NULL, (size_t) size,
     PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS,
     -1, 0);
    printf  ("%lx\n", (long) p); // print start address
    while (1)
        ;  // loop so that we can run pmap on the process
    return (0);
}
```

When we run the program, it prints the start address of the mapped region as below:

```
$ ./mem
7ffff75e4000
```

The segment is mapped to the virtual memory of the process `mem` starting at virtual address **`00007ffff75e4000`**. We can compare this address with the output of the **`pmap`** command. The `pmap` command lists information about the virtual memory areas used by a process. The pid of the process mem is `19238`. We can get the `pmap` output as follows (output in **x86-64** architecture) (abbreviated little bit). All addresses are in hexadecimal.

```
$ pmap -x 19238
19238:   ./mem
Address           Kbytes     RSS   Dirty Mode  Mapping
0000555555554000       4       4       0 r-x-- mem
0000555555754000       4       4       4 r---- mem
0000555555755000       4       4       4 rw--- mem
0000555555756000     132       4       4 rw---   [ anon ]
00007ffff75e4000    4096       0       0 rw---   [ anon ]
00007ffff79e4000    1948    1124       0 r-x-- libc-2.27.so
00007ffff7bcb000    2048       0       0 ----- libc-2.27.so
```

```
00007ffff7dcb000      16      16      16 r---- libc-2.27.so
00007ffff7dcf000       8       8       8 rw--- libc-2.27.so
00007ffff7dd1000      16      12      12 rw---   [ anon ]
00007ffff7dd5000     156     156       0 r-x-- ld-2.27.so
00007ffff7fdf000       8       8       8 rw---   [ anon ]
00007ffff7ff7000      12       0       0 r----   [ anon ]
00007ffff7ffa000       8       4       0 r-x--   [ anon ]
00007ffff7ffc000       4       4       4 r---- ld-2.27.so
00007ffff7ffd000       4       4       4 rw--- ld-2.27.so
00007ffff7ffe000       4       4       4 rw---   [ anon ]
00007ffffffde000     132       8       8 rw---   [ stack ]
ffffffffff600000       4       0       0 r-x--   [ anon ]
---------------- ------- ------- -------
total kB            8608    1364      76
```

In the output above we see that the segment is mapped to the part of the virtual memory indicated with **bold**.

```
00007ffff75e4000   4096K rw---   [ anon ]
```

The size of the segment is 4 MB (4096 KB). Since it is *not backed* by a file, it is called an **anonymous** memory region (anon). As we can see, the segment starts at a page boundary (last 3 hex digits are 0), not in the middle of a page.

There are other virtual memory regions used by the process. For example, `libc` is also mapped to some region of the virtual memory (starting at address `00007ffff79e4000`). The stack is sitting at virtual address `00007ffffffde000`. The *ordinary* heap (from where `malloc()` allocates memory) is living at address `0000555555756000`. The code of the program (text section) starts at virtual address `0000555555554000`. The output is also showing the name of the *backing* file for some of the memory regions. For example, for the first 3 virtual memory regions, the backing file is the executable file `mem`. The memory regions used by the `libc` shared library are backed by the file `libc-2.27.so`.

The output is also showing the sizes of the virtual memory regions (`Kbytes` column) and also the current physical memory usage (in Kbytes) of the regions (`RSS` -resident set size - column). Since our segment is just mapped, but is not written yet, it has no pages created and allocated space in physical memory yet (RSS is 0 Kbytes). As you can see, the majority of the physical memory usage is coming from `libc`. In total, the program `mem` is using around 8 MB of virtual memory (including the mapped shared libraries), but just around 1.3 MB of physical memory (majority of that comes from `libc` and the dynamic linker `ld`). When we will write values into our segment, at that time pages of the segment (corresponding to where writes are happening) will be created and allocated physical frames. The output is also showing the protection bits for each virtual memory region. For example, the code part is read-only and executable (`r-x--`). Our segment is readable and writable (**`rw---`**).

An **application program**, for example **`app.c`**, that will use your library can be as follows.

```
#include <sys/mman.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "dma.h"     // include the library interface

int
main (int argc, char ** argv)
{
    void *p1;
    void *p2;
    void *p3;
    void *p4;
    int ret;

    ret = dma_init (20); // create a segment of 1 MB
    if (ret != 0) {
        printf ("something was wrong\n");
        exit(1);
    }

    p1 = dma_alloc (100); // allocate space for 100 bytes
    p2 = dma_alloc (1024);
    p3 = dma_alloc (64); //always check the return value
    p4 = dma_alloc (220);
    dma_free (p3);
    p3 = dma_alloc (2048);

    dma_print_blocks();

    dma_free (p1);
    dma_free (p2);
    dma_free (p3);
    dma_free (p4);

}
```

Below we see a **`Makefile`** that can be used to compile and build the library and an application that uses the library.

```
all: libdma.a  app

libdma.a: dma.c
     gcc -Wall -g -c dma.c
     ar rcs libdma.a dma.o

app:    app.c
     gcc -Wall -g -o app app.c -L. -ldma
```

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

Part B – Experiments (20 pts):

Please do a lot of timing (and space) experiments. Measure the time required to allocate memory of various sizes. You can use again the `gettimeofday()` function to get the time (in microseconds granularity) at an instant. Measure the time required to release (free) memory. You can also measure the fragmentation amount for various cases. You can do random allocations (random sizes) as well. If you do a lot of random (size) allocations, you can also see the effect of external fragmentation (you will see the library will not be able to satisfy some of the requests). Do a lot of experiments. Plot the results or put them into tables. Try to interpret them and try to draw conclusions.

Submission:

Put all your files into a directory named with your Student ID (if done as a group, the ID of one of the students will be used). In a README.txt file, write your name, ID, etc. (if done as a group, all names and IDs will be included). Do not include executable file in your tar package. Include a Makefile so that we can just type make and compile your program. 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 into Moodle.

Tips and Clarifications:

- Start early, work incrementally.
- Normally bitmaps are not used to keep track of heap usage and free memory. Since search is slow. But this is an educational project. The purpose here to exercise, learn and develop skills.
- Normally, heap area is grown *dynamically*. That means a large memory region (like 4 MB) will not be allocated immediately to be the heap area. We will not do that in this project for simplicity. The ordinary heap area, for example, is grown dynamically. It can start with some certain initial size. Then when more and more `malloc()`'s are done, it will be grown. There will be usually space to grow, since the virtual memory of a process in a 64-bit machine is very big and sparse.

- Your library will be linked *statically* with an application. The Makefile example shows how the library can be built and linked with an application.
- Since the minimum size of the segment is 16 KB, the size of the bitmap is a multiple of 16 bytes.
- You need to write test applications to test your library. We will also write test applications to test and evaluate your library.