ITCS 6144/8144 Assignment - Memory Allocator

Prepared by: Abdullah Al Raqibul Islam (UNCC ID # 801151189)

Test platform:

- Processor: Intel(R) Xeon(R) CPU E5-2620 2.00GHz (12 Core)
- Linux version: 5.0.0-27-generic
- GCC version: 7.4.0 (Ubuntu 7.4.0-1ubuntu1~18.04.1)

Project Objective:

In this project, I have developed an emulated memory allocator by implementing buddy and slab algorithm. As this project is a user level emulator (to gain better understanding of kernel level memory management), I have followed several custom rules that may not be matched with the standard one. Here I am going to give a brief description of assumptions and logics applicable for this project.

Assumptions, Logics, and Features:

1. At the beginning, initialize the memory regions from where I will allocate the requested memory. To get the initial memory region, I have used **mmap** function to get the memory:

```
mmap(NULL, MEM_SIZE, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
```

2. Once receiving the initial memory pages, the program will manage them using the buddy and slab algorithms. In this project, I have taken 4MB of memory.
3. In the initialization process, I have initialized all the necessary data structures as well. As I have mentioned before, I have implemented two algorithms (i.e. buddy and slab), I have used two data-structures, one for storing buddy and another for slabs. I have used external memory to store buddy information. Slabs information is kept in the memory regions which I have taken early by using mmap. I have initialized 10 slab caches with different size of objects (i.e. ranging from 2^5 to 2^14).
4. **kmalloc_8144()** function will allocate a given size memory and return a void pointer (i.e. void *). This memory will come from the slab cache only. When more pages are needed for the slab cache, it will come from buddy allocator to allocate more page frames for the slab caches.
5. **kfree_8144()** function will be used to free the allocated memory. The freed memory should be returned back to slab system. The slab system should return the free pages to buddy allocator when **purge_8144()** is called.
6. **internal_frag()** and **external_frag()** can be called to calculate the internal fragmentations and external fragmentations of the memory allocator.
7. As I have used the in-memory slab implementation, I keep slab information (slab header information) in the memory received from buddy. In slab, while preparing a object chain, I also keep object information (i.e. is this object memory is been assigned) as a object header. So, while initializing a slab cache, there are some spaces left over (depending on the object size stored in the slab cache) at the end of the slab area. I considered this unused area as internal fragmentation. So, we can precalculate the internal fragmentation while initialized the slab cache.

8. On the other hand, the external fragmentation is the memory in between the two used buddy area. To get better view about internal and external fragmentation, please check the figure 1.
9. At the end of the program, finalize the memory regions as well as the necessary data structures.
10. The program data structures are well protected by the mutex lock to handle concurrent request.

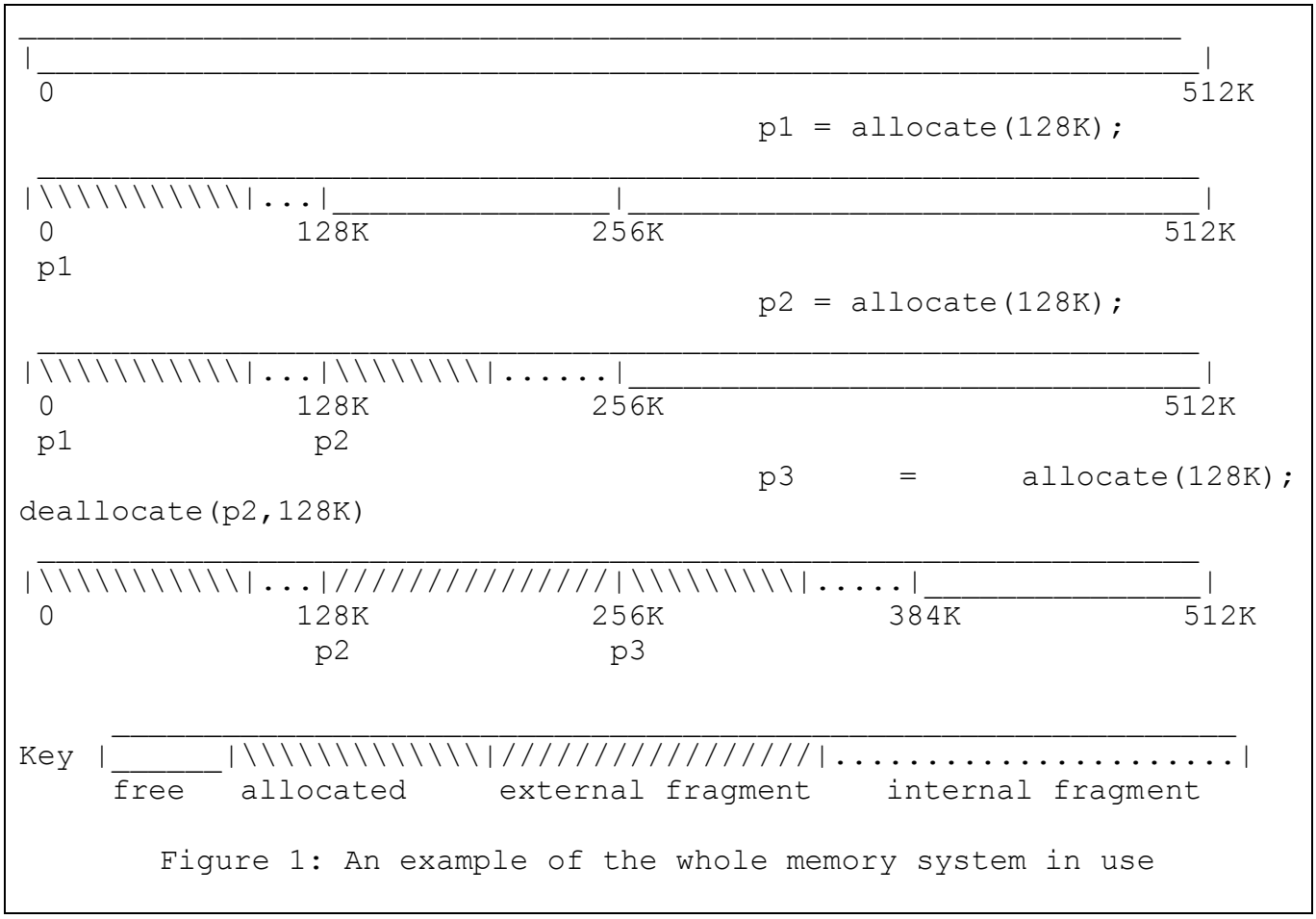

Figure 1: An example of the whole memory system in use

Buddy and Slab Data Structures:

Let's discuss on the data structures used to maintain buddy and slab information. As I have mentioned earlier, memory will always assigned from slab caches and if the specific slab cache don't have any space left, it will call buddy to get memory for it. After getting memory from buddy, slab will initialize it's cache and return the object pointer to the requesting thread. If the slab cache is full and buddy don't have any memory left, a null pointer is returned as a response.

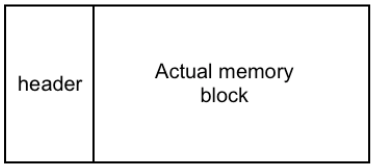

Figure 2.1: Single slab cache

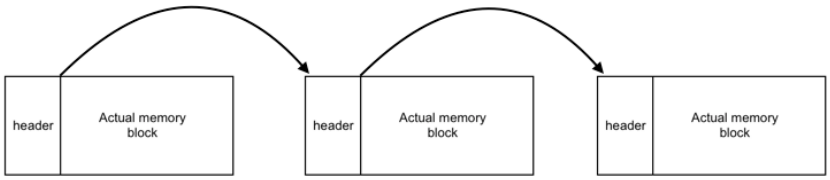

Figure 2.2: Chain of slab caches

Figure 2: Slab Architecture

Now, let's do some basic mathematics that can be helpful to understand the memory allocator in depth. The memory region from where the memory pointer will be allocated has a size of 4 MB or 2^25 Bytes. This whole region will be manipulated by buddy system. On the other hand, memory allocation requests will be handled by the slab system. For different size of objects, slab caches are been initialized at the beginning of the program. We have considered objects with size ranging from 2^5 Bytes to 2^14 Bytes. While initializing a slab cache, we request buddy with fixed size (i.e. 128KB in this case) to assign the memory. When buddy allocate a pointer for that size, we initialize the whole slab cache for further object allocation. As I have mentioned earlier, each slab header and object header is implemented as in-memory basis. So while initializing slab caches with different object sizes, here is the list of possible objects allocation and internal fragmentations (in Byte),

```
2^5 size object: # of objects: 2339, internal fragmentation: 48
2^6 size object: # of objects: 1489, internal fragmentation: 0
2^7 size object: # of objects: 862, internal fragmentation: 8
2^8 size object: # of objects: 467, internal fragmentation: 272
2^9 size object: # of objects: 244, internal fragmentation: 248
2^10 size object: # of objects: 125, internal fragmentation: 32
2^11 size object: # of objects: 63, internal fragmentation: 496
2^12 size object: # of objects: 31, internal fragmentation: 3312
2^13 size object: # of objects: 15, internal fragmentation: 7792
2^14 size object: # of objects: 7, internal fragmentation: 16176
```

In this project, the highest order of buddy will be 10. Which means, order 10 will hold one 2^15 region. Here is the detailed list,

```
Order 10: size 2^25, number of partitions 1
Order 9: size 2^24, number of partitions 2
Order 8: size 2^23, number of partitions 4
Order 7: size 2^22, number of partitions 8
Order 6: size 2^21, number of partitions 16
Order 5: size 2^20, number of partitions 32
Order 4: size 2^19, number of partitions 64
Order 3: size 2^18, number of partitions 128
Order 2: size 2^17, number of partitions 256
Order 1: size 2^16, number of partitions 512
Order 0: size 2^15, number of partitions 1024
```

As we will call buddy to allocate memory with fixed size 128 KB (i.e. 2^17 Bytes), buddy will never be bellow of order 2 in this case.

Also, given the address of an allocated block, we can easily find its buddy by XOR its address. Suppose we have block B1 of order x, we can compute the buddy as follow:

B2 = B1 XOR (1 << x)

Correctness of the Implementation:

I have prepared two test scenarios to check the correctness of my buddy and slab implementations. Here is the brief idea about my tests,

1. Test the correctness of the slab memory allocator implementation: In this test what I have done,
    a. called kmalloc_8144() in a way so that we will have three connected slab caches for a specific size
    b. called kfree_8144() for the middle slab cache
    c. called purge_8144() and observe the integrity of slab caches
    d. slab implementation is correct!
2. Test correctness of the buddy implementation: This test case is designed for the following specification,
    a. total memory: 4 MB (i.e. 2^25 Bytes), each slab cache will hold memory: 128 KB (i.e. 2^17 Bytes)
    b. so, we will have at best 256 (i.e. 2^8) slab caches
    c. initially, 10 slab caches are initialized
    d. if the object size is 2^14, we will able to allocate 7 objects in each slab cache
    e. for 2^14 size objects, we can occupy at best (256-10+1) slab caches (i.e. 9 caches is occupied for objects of other sizes)
    f. for this test case, we call a special purge method purge_8144_v1()
    g. in the purge_8144() method, we always purge by keeping a single slab cache for each size of objects
    h. but in this purge method, we purge a slab cache if it is empty and return it back to buddy
    i. so, in this test what we have done,
        i. initialize the slabs
        ii. occupy maximum possible slab caches
        iii. free all the occupied memory
        iv. purge all the free slabs
        v. and we get the single unit of buddy back
    j. buddy implementation is correct!

Limitations & Future Improvements:

- For keeping buddy information, I have used external memory which can be used from the initially occupied memory region (like slab).
- Slab architecture contains a single chain of objects. We can use different chain (i.e. EMPTY, PARTIAL, FULL, etc.) to make the purge faster.
- For marking object pointers (i.e. IS_FREE) we can use bitset to get better memory usage. Bitset will also help to gain better performance in finding empty slots in the slab cache chain.
- We can calculate the performance w.r.t. the number of successful allocation requests.

References:

1. Memory Allocators 101 - Write a simple memory allocator: https://arjunsreedharan.org/post/148675821737/memory-allocators-101-write-a-simple-memory
2. The SLAB Memory Allocator: http://3zanders.co.uk/2018/02/24/the-slab-allocator/
3. Dynamic Storage Allocation: http://homepage.cs.uiowa.edu/~jones/opsys/notes/27.shtml
4. Memory management Buddy allocator, Slab allocator: http://students.mimuw.edu.pl/ZSO/Wyklady/05_pamiec/5_pamiec_en.html

Compile & Run Instruction:

```
-- go to project directory
$ cd path-to-directory/assignment_2-memory-allocator/

-- create build directory
$ mkdir build

-- go to build directory
$ cd build

-- create make file from CMakeLists, make file will be generated in
the same "build" directory
$ cmake ..

-- make the project, executable files will be generated in the same
"build" directory
$ make clean && make

-- run the program
$ ./mem_test
```