# Operating Systems 2
## SPRING SEMESTER 2018

INSTRUCTOR: DR. SPARSH MITTAL
Supreet Singh cs16btech11038
Mayank Hooda cs16btech11022

---

## Task : Implement Linux Slab Allocator

## Problem 1 :

We implement a Linux Slab Allocator (in user space). We make the library called *libmymem.so* which exports/declares two APIs viz.,

*void* mymalloc(unsigned size);*
*void myfree(void *ptr);* in *libmymem.hpp*

### Data Structures Used :

- Bitset ( For storing the bitmaps associated with availability of a memory space)
- Hash Table (For storing the linked list corresponding to various sizes)

### Code Heuristics:

- For accessing and updating any range of available memory locations of a given slab , we use a bitset which acts as a bitmap array , with each bit alterable. A *bool* array uses much more memory than using a bitset because bitset lets us save a binary number in every bit while bool takes 1 byte. For the same storage requirement ,

$$Space\ Complexity\ (Bitset)\ =\ Space\ Complexity(Bool\ array)/8$$

- For giving the best possible fit , we give the slab which allocates the next biggest memory segment( in powers of 2). Hence on an average $size/2$ amount of internal fragmentation occurs.

- For allocating memory to a slab , we use *mmap* function which allocates 64 KB space to every slab. We use an ANONYMOUS map for mapping into the process virtual space .This space is used to allocate memory to objects but it also includes the metadata associated with each slab entry which includes :
    - Total Objects
    - Free Objects
    - Bitset map
- Therefore total number of objects allocatable on a single slab is lesser than absolute $(Size\ of\ Slab)$ / $(Size\ of\ each\ object)$ example being 7 objects instead of 8 in 8192B slab .
- The hash table has a slab-linked list associated with every entry wherein we try to fill the current slab till it's full otherwise we make a new slab for further elements.
- For freeing purpose , we create a pointer reference in the object struct so as to backtrack our way to the slab.
- We just update the bitmap whenever the free function is called but do not delete the slab until every element is not freed. A slab is not deleted till every possible allocation is freed. If it is , then <*sys/mman.h*> library function *munmap* is used.
- If any reference to a memory occurs which tries to free it twice , we check the bitmap corresponding to the entry , if it results in 0 then we throw off an error.
- For assuring the privateness of our structs , we proceed with *static* variables.

## Problem 2 :

We have to make Linux Slab allocator that is re-entrant & thread-safe.

### Data Structures Used :

Additional to the previous data structures used , we use :

- 12 Mutex locks(For each bucket) (For locking the critical section)

### Code Heuristics:

- We do net let two threads get memory allocated from the same bucket(corresponding buckets) at the same time because that might lead in inconsistency issues as they involve several writes to a same slab itself.

- We do not let any myfree function execute parallely with a mymalloc or a mymalloc. Thus we lock all the 12 locks on executing the myfree function and unlock all of them at the end of the myfree function. Any call to two different indexes in the hash table might also lead to error since the index is calculated after entering into the function , which is not thread safe.
- The code is reentrant as we do not have any global variables whose state has to be consistent upon calling of any ISRs.