Operating Systems–II: CS3523
Spring 2018
Programming Assignment 5: Linux Slab Allocator
Last date for submission: 22nd April 2018, midnight

- *The assignment needs to be done as a team of two students each.*

There are two parts for this assignment. Problem Statement 1 describes about implementing Linux Slab Allocator while Problem Statement 2 asks you to make it re-entrant & thread-safe.

Problem Statement 1:
Implement a simplified version of Linux Slab Allocator (in user space). You have to implement the library called **libmymem.so** which exports/declares two APIs viz.,

```
void* mymalloc(unsigned size);
void myfree(void *ptr);
```

in libmymem.hpp

In Linux, the library source files are denoted as *.hpp & *.cpp. The library binaries are stored in .so format. So your source files will be: libmymem.hpp & libmymem.cpp

The goal of this assignment is to design & implement a slab allocator in a C++ procedural library for memory management on Linux.

Details of the Simple slab allocator

- Linux kernel implements slab allocator to minimize fragmentation, and avoid compaction. Slab allocator gets memory from page allocator, and provides APIs for kernel modules for allocating small objects. (As root or sudo, see the slabs with name kmalloc-<size> in “cat /proc/slabinfo” output)
- Linux user-space memory allocator (the libc malloc/free) also implements a slab allocator. This code is quite complicated. It gets memory from OS using private anonymous mmap.
- You need to design and implement a simple slab allocator in a library (a simplified version of what Linux slab allocator).
- Once a slab allocator is implemented in user-space, with few changes it can be ported to kernel-space.
- Slab allocator gets memory from the OS using private anonymous mmap() in chunks of 64KB [Refer the sample mmap_priv_anon.c]. Each chunk is called a slab.
- The size of objects to be allocated can vary from 1 to 8192 bytes. You should aim to fit as many objects as possible in one slab (mmaped segment), to maximize the utilization of the virtual address segments.
- It needs to keep track of all the slabs in a private data-structure shown in figure below. The complete design of this data-structure will be explained in the session organized by the TA.

- The mymalloc() function in libmymem.cpp allocates object from a best-fit bucket. This is due to the nature of the data-structure.
- The myfree() function in libmymem.cpp frees the object pointed by the ptr (which was allocated using the mymalloc()). It makes the object available for re-use.
- Instructions for generating the library

```
$ g++ -Wall -Werror -fpic -c libmymem.o -I . libmymem.cpp
$ g++ -shared -o libmymem.so libmymem.o
```

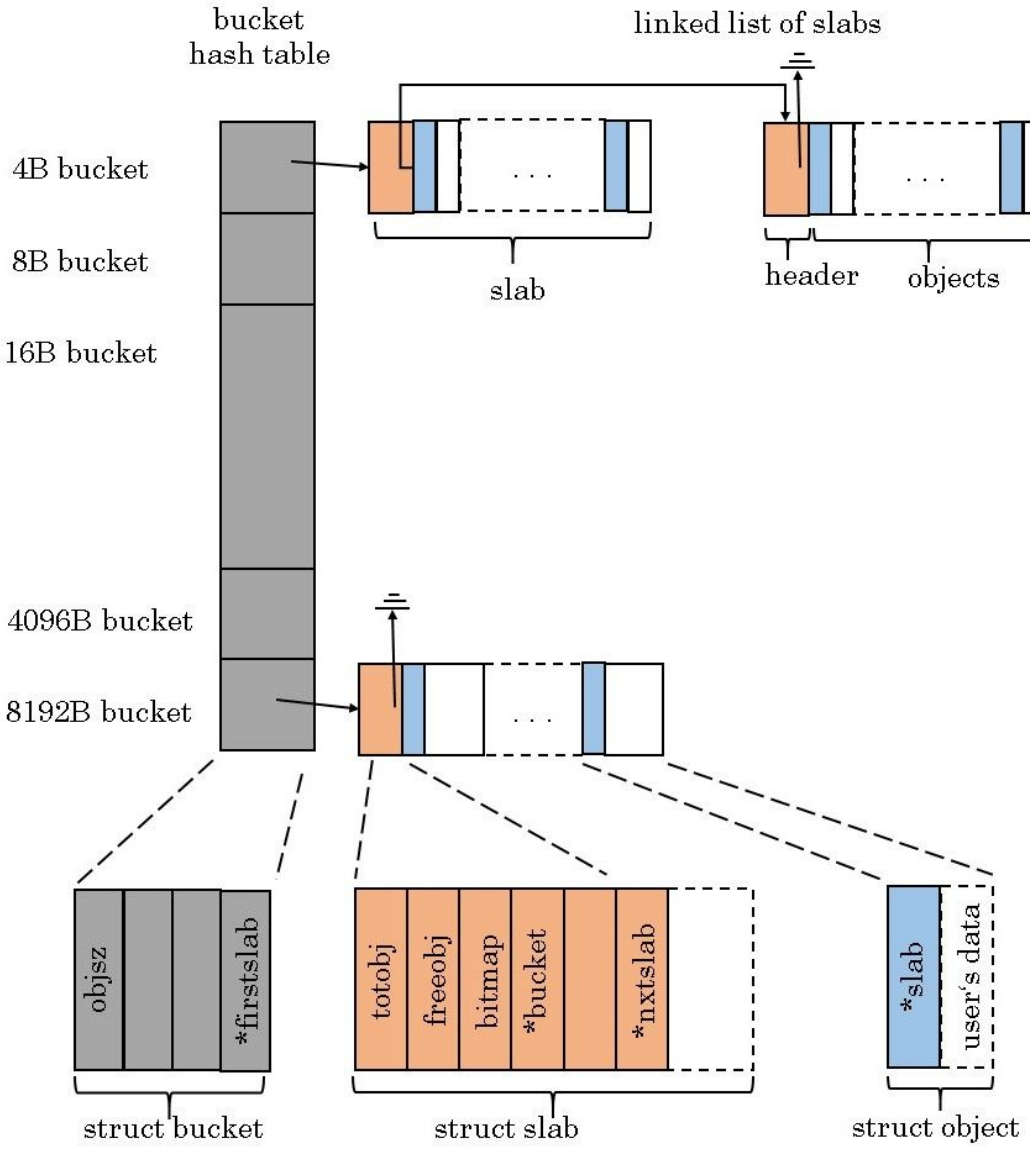

Figure not to scale

Now, to test your memory allocation library, you have to develop a test application called as memutil.cpp described as follows:

- This utility program in C/C++ calls the mymalloc() and myfree() specified number of times with some minute random sleep between allocation and free. The size of allocations can randomly vary from 1 to 8192 bytes. After allocation, write should be done to the memory so that demand paging system in Linux will do page frame allocation.

- Ensure there are no memory leaks in the memutil
- Instructions for generating the utility

```
// note: The flag is -l mymem & not -l libmymem
$ g++ -I . -L . -Wall -o memutil memutil.c -l mymem
```

- Instructions to run the utility

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
$ mmemutil -n 100
```

-n : number of iterations of alloc-write-sleep-free (from 1 to 100000).

Note:

- We will test your library with our own memutil.cpp, so make sure your library sticks to the APIs given above.
- The mymalloc() and myfree() should not use the LibC new/malloc()/calloc()/realloc() and delete/free(). It also should not use any C++ STL classes.

Problem Statement2:

As described earlier, for this part, you have to make Linux Slab allocator re-entrant & thread-safe.

- Make sure that the private data-structure is thread-safe and re-entrant.
- The memutil.cpp program need to be made multi-threaded to as shown in pseudo code below.
- Pseudo code of the memutil

```
main() {
        // Create nthreads to do allocations/free concurrently
        for (t=0; t<nthreads; t++) {
                // create a list of object sizes for this ith thread
                // Create a thread
        }
        // wait for all threads to join
}

// The thread iterates for niterations and allocates objects of various sizes.
threadmain(list<int> objsizelist, int niterations)
{
        for (i=0; i<niterations; i++) {
```

```
                randomly pickup a size from objsizelist;
                invoke mymalloc() bail out if error;
                // write to the allocated memory
                // sleep for small random time (no sleep causes more contention)
                // call myfree()
        }
    }
```

- Instructions for generating the utility

```
$ g++ -I . -L . -Wall -o memutil memutil.c -l mymem -l pthread
```

- Instructions to run the utility

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
$ mmemutil -n 100 -t 2
```

-n : number of iterations of alloc-write-sleep-free per each thread. (from 1 to 100000)
-t : specifies number of threads (can vary from 1 to 64)

Submission Instructions:

1. Place the source code (above 3 files), readme, pdf report in a folder : Assgn5-mem-<RollNo1>-<RollNo2>
2. And zip the folder.
3. Upload the zip file.

If you don't follow these guidelines, then your assignment will not be evaluated. Upload all these on the drive by the specified deadline.

Any queries in the problem statement to be sent to the TAs before the timeline given in the assignment post.

Evaluation:

- Problem1
  - Library : 30 marks
  - Header file: 5 marks
  - Utility: 10 marks
  - Code indentation and comments : 5 marks
- Problem2
  - Library thread-safety & re-entrancy: 30 marks
  - Multi threading the utility : 15 marks
  - Code indentation and comments : 5 marks.
- If the code does not compile, no marks will be given.
- If there are correctness issues, segmentation faults, marks will be deducted from the component that the bug belongs to.