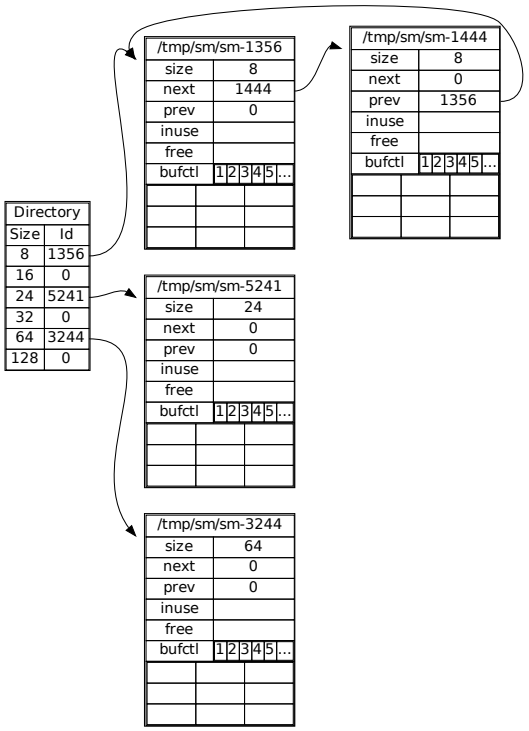
CEng 536 Advanced Unix
Fall 2011
HW2
Due: 14/11/2011

In this homework you will write a shared memory slab allocator library called `libsmslab`. One problem in shared memory multiprocess applications is to use dynamic data structures. Since standart language functions like `malloc()` and `free()` allocates the heap variables from of main memory, it is not trivial to keep dynamic structures like trees, linked lists etc. on shared memory. One has to use a non-standart library or write his/her own allocator.

The library you are going to implement will have the structure given in the figure. The Directory page is a shared memory segment of size 4K containing a directory for slab pages. Directory — not exclusively — contains object sizes and id of the corressponding slab page. Library will be capable of allocating objects of sizes given in this directory.



Each slab page has a fixed size (1024*1024 bytes). When a page is full, a new page is allocated and pages are linked with next and prev links. Each slab page is a shared memory segment. You can use memory mapped files under a directory like /tmp/sm/id-.. to realize shared memory or use SYSV shared memory ids directly. A page id is a `short`, unique integer.

In order to make sure a heap object in this library unique, library represents a pointer as a tuple consisting of slab page id and the objects position in the slab.

```
typedef struct sm_pointer {
  short pageid;
  short objind;
} sm_pointer_t ;
```

While accessing a pointer, programmer should convert this pointer into an actual memory reference by calling `smref()` and similarly call `smderef()` to convert a memory reference to `sm_pointer_t`. `smref()` automatically attaches/maps the shared memory if it is not attached yet. Library also should provide locking facilities for allocated objects. It can be realized as locking corressponding file regions of mapped file or maintaining a semaphore array per slab. No need to mention, all library meta-data should be protected against race conditions. Allocated objects are persistent, they survive even after process terminates. The library user is responsible for concurrency and integrity issues like dangling references. You can implement a garbage collector for getting bonus points (bonus B). For garbage collector, you need to keep a reference count per object in the slab, and each process increments this counter when `smref()` is called. When counter gets 0, the object is deallocated. Counter decrement is explicit through `smrelease()` call.

You need to implement the following functions:

`void sminit()`
: Initializes the library, attaches/maps the Directory page etc. If directory does not exist, creates it.

`void smclose()`
: Destroys all shared memory attachments/mappings of slabs, deallocates all library internal data structures for the process.

`void *smref(sm_pointer_t)`
: Tests if the slab page is attached/mapped and finds the object from slab and returns the actual pointer to it.

`sm_pointer_t smderef(void *)`
: Finds the slab page containing the actual pointer and returns the `sm_pointer_t` typed pointer to it.

`sm_pointer_t smalloc(int)`
: Allocates a new object from the suitable slab (the smallest fixed size that requested size fits). If all slab pages are full, creates and attaches a new slab page.

`void smfree(sm_pointer_t)`
: Deletes the object from slabs. If deleted object is the last object of slab, the slab is destroyed, the file is deleted or the shared memory is deleted.

`void smlock(sm_pointer_t)`
: Locks the object for concurrent access. Blocks until lock is established.

`void smunlock(sm_pointer_t)`
: Unlocks the object.

`int smcreateslab(int)` (bonus A)
: Creates a new slab entry for the size given with integer parameter. It will add a new directory entry.

`int smdestroyslab(int)` (bonus A)
: Destroy the directory entry for given size. All linked slabs will be destroyed as well.

`void smrelease(sm_pointer_t)` (bonus B)
: mark the pointer as 'not to be used by this process anymore'. Garbage collector decrements the reference count of the object and deallocates it when this counter gets 0. Applications will call this function instead of smfree().

Please note that you need to keep a bidirectional mapping among slab id and memory address range in each process in order to implement `smref()` and `smderef()`. This mapping should be fast. You can use C++ maps or `tsearch()` in libc as fast data structures.
Bonus A is 10 extra points and Bonus B is 15 extra points worth. A sample program using the library will be provided later.
Please provide your implementation on 4 files in a tar.gz archive (no RAR, no ZIP please):

- A header file smslab.h that can be used as include file for library users as well.
- Library implementation on a C or C++ file. It should not have main() implementation. All functions and global variables that are for library internal use should be defined as `static`.
- A Makefile compiling your homework as libsmslab.so.
- A readme.txt explaining which bonuses you completely implemented.

Please ask all questions to:
`news://news.ceng.metu.edu.tr:2050/metu.ceng.course.536/`
`https://cow.ceng.metu.edu.tr/News/thread.php?group=metu.ceng.course.536`