CS377: Operating Systems

Lab 3: Memory Allocator

Due November 29, 2005

Assignment

For this lab, you will be developing a scalable and fast memory allocator. Your allocator will be a shared library that can be used with any existing program that was dynamically linked to libc (standard C library). You will implement the complete malloc API (malloc, free, calloc, realloc). Not only should these be fast, they need to scale with the number of threads (in other words, you will not be protecting the internal data structures with a Big Giant Lock). You will then test your allocator by setting the LD_PRELOAD environment variable to point to your library, which will make programs use your allocator instead of the standard one. You should write simple test programs to check that it works, and then try it out with applications like Mozilla and test applications to be posted on the course web page.

Interface to the OS

Your memory allocator will request memory from the operating system for the heap using mmap() from the special file "/dev/zero". You should obtain 4K chunks of memory for most objects (see below).

Internal memory layout

You should allocate memory for the program in sizes of powers of 2, starting at 8. When asked for a memory region of a different size, you should allocate the smallest allowed region larger then requested size (best fit). Every object should have an 8 byte header, invisible to the user program. Each page should only contain objects of the same size, and should be divided up right after it's been allocated. For every object larger then 1024 bytes you should allocate a separate page. You will not allocate any objects larger then 4088 bytes (4K-8 bytes). You should maintain unused objects in lists, segregated by size. Moreover, you should create a number of separate free lists (e.g., 5) for use by different threads. Calls to malloc()/free() made from thread with id TID (obtained via pthread_self()) should use pool number TID % 5.

Object headers

Each header should be 8 bytes long. The high order 4 bytes contain the (actual usable) size of the object, while the low order 4 bytes should be 0 if the object is allocated, and otherwise contain a pointer to the next object on the free list.

malloc()

Here's pseudo code for malloc(). Note: calls to mmap and pthread functions are simplified. See the man pages for exact syntax.

```
void *malloc(int size) {
   pool = pthread_self() % T
   pthread_mutex_lock(pool);
   if  (size > threshold) {
        page = mmap(4K, "/dev/zero")
        [set first 4 bytes to contain 4k-8bytes]
        pthread_mutex_unlock(pool)
        return (page + HEADER_SIZE)
    }
    else {
        real_size = [smallest power of 2 larger then size];
        if (free_list[pool][real_size] != NULL) {
            obj = free_list[pool][size];
            free_list[pool][size] = "obj->next" //second 4 bytes of obj
            pthread_mutex_unlock(pool);
            return (obj+HEADER_SIZE);
        } else {
            new_page = mmap(4k, "/dev/zero")
            obj = new_page;
            [set first 4 bytes to contain real size]
            [split the rest of the page into separate objects, and add them to free list]
            pthread_mutex_unlock()
            return (obj+HEADER_SIZE)
        }
}
```

free()

free() should check if the object is actually allocated, and place it onto appropriate free list. If the object was taking up an entire page or more, that memory should be unmapped instead.

Internal data structures

All of your internal data structures should be static, and get initialized on the first call to malloc.

Logistics

Your code should compile using gcc on Edlab Linux machines. You should keep it in SVN as usual. To make a shared library, you need to compile your code like this:

```
gcc –shared mymalloc.c –o libmymalloc.so –ldl
```

To use it, do this:

```
export LD_PRELOAD=/your/directory/goes/here/libmymalloc.so
```

To go back to using the default library, do this:

```
export LD_PRELOAD=
```