Lab Exercise 3: Memory Management

COP 4610 -- Operating Systems Principles

Demo in Lab sessions: U01 - Tue Mar 28th; U02: Fri Mar 31st)

Introduction:

The memory management layer is the part of the kernel that serves out all memory allocation requests. To handle smaller memory requests (less than a whole page, e.g. through **`malloc()`**), the kernel currently gives a choice of three different allocators: the SLAB allocator, the SLUB allocator, and the SLOB allocator. SLUB (the most recent of these) and SLAB are complex allocation frameworks for use in resource-rich systems such as desktop computers. They are designed to reduce internal fragmentation of memory, and to permit efficient reuse of freed memory.

The SLOB (Simple List of Blocks) allocator, on the other hand, is designed to be a small and efficient allocation framework for use in small systems such as embedded systems. Unfortunately, a major limitation of the SLOB allocator is the high degree to which it suffers from internal fragmentation. One likely cause for SLOB's high fragmentation rate is the fact that it uses a simple first-fit algorithm for memory allocation.

In this project, you will investigate this issue by modifying the SLOB allocator to use the **best-fit** allocation algorithm, and by writing system calls to provide a measure of the degree of internal fragmentation within the SLOB allocator at a specific point in time.

Objectives:

1. Attempt to improve the fragmentation rate of the SLOB allocator by modifying it to use a different memory allocation algorithm.
2. Understand and modify an existing component of the Linux kernel tree.

Tasks:

1. Read through Chapter 8 (Memory Management) of the Bovet book and read the file header at the top of `mm/slob.c` to familiarize yourself with the memory allocation schemes presently used in the kernel. Browse through the code in `mm/slob.c` to understand how the SLOB allocator works. Specifically, make sure that you understand that the function `slob_alloc()` implements the **first-fit** allocation algorithm.

2. To get the kernel to use the SLOB allocator, start with the kernel that you worked with in Lab 1 (the one with your new system call). Use '`make menuconfig`'. Go into *General setup* → enable *Configure standard kernel features (for small systems)*; then *Choose SLAB allocator* → SLOB (Simple Allocator). Once this is done, you should be able to compile a kernel that uses SLOB instead of SLAB.

3. Devise a method to keep track of all of the memory claimed by the SLOB allocator for small allocations (i.e., every time the `if(!b)` beginning on line 368 of `slob.c` is entered while the `free_slob_small` list is iterated through) along with the amount of memory that was not served in an allocation request (i.e. the amount of memory contained in all blocks on the free list). **Write separate system calls** (refer to lab #1) at the bottom of `mm/slob.c` to return each of these amounts in bytes. I.e., `asmlinkage long sys_get_slob_amt_claimed(void)` for the number of requested bytes and `asmlinkage long sys_get_slob_amt_free(void)` for the number of bytes in the free list (`free_slob_small`) that cannot be allocated due to fragmentation. Comparing the values returned by each of these functions will give you some idea of the degree of fragmentation suffered by SLOB. Write a simple program that uses these two system calls to make this comparison. For example, this program can ask for memory multiple times using `malloc()` and between the memory requests it invokes the two system calls to keep track of the memory fragmentation situation. Feel free to add your own design here (e.g., running another program using *execve()*) in your test program.

   Fragmentation happens when a memory request cannot be satisfied by any single free block but the sum of all free blocks can. Specifically, this situation happens when the *true* branch of `if(!b)` beginning on line 368 of `slob.c` is entered. At this point, we can at least collect (1) the size of the current memory request (memory claimed) and (2) the total size of memory in the free list (`free_slob_small`). The two new system calls can report these two numbers in the simplest case. For example, number (1) can be 255 bytes and number (2) can be 1342 bytes. From these two numbers we can have some idea about the fragmentation. However, this method only reflects the situation at a particular moment, i.e., the latest memory allocation request. But we may want to have information over sometime; then we need to get the accumulated numbers (or average numbers) for (1) and (2). Use two global arrays to store the last 100 measurements of (1) and (2), and the two system calls to report the averages of these measurements, respectively.

   The header of `slob.c` (the comments) gives a very useful description about the free list. Please read it carefully, so that you know how to count the total bytes in the free list.

   Although the Linux SLOB allocator maintains three free lists, for this lab you only need to focus on the `free_slob_small` free list.

4. Modify `mm/slob.c` so that it uses the **best-fit** algorithm. Begin with the function `slob_alloc()`, and make changes elsewhere as appropriate. The changes necessary to do this should not be too significant, so if you find yourself writing many lines of code you should rethink your implementation.

Make sure you add a section to the file header including the names of your team members, and a description of the changes you made.

5. Use the program you wrote in task **4** to compare the amount of fragmentation suffered by SLOB using the first-fit algorithm and the amount suffered by SLOB using the best-fit algorithm. Record your observations and conclusions in a README file.

6. Follow the "Kernel Patch Guide" to make a patch file representing the changes between your modified kernel and the vanilla kernel. Name your patch file `linux-2.6.36-best_fit_slob.patch`. This will make it easy for the TA to examine your changes. Make sure this patch contains only the changes you want to submit, no experimental changes.

Useful resources:

• Chapter 8 (Memory Management) of book: Daniel P. Bovet, Marco Cesati. Understanding the Linux Kernel, ISBN: 978-0596005658 http://users.cs.fiu.edu/~prabakar/cop4610/Common/Kernel.pdf
• The file header of `mm/slob.c`

What to turn in for deliverables:

Create a gzipped `tar` archive containing the items below
and submit it in Moodle (one submission per team) before **11:55pm March 31, 2017**.

- Your modified file `mm/slob.c`
- The patch file created in task **6**
- A README file describing the results of your comparisons in task **5**
- Source code of the program you wrote in task **3** including a Makefile

Tentative evaluation criteria:

Item	Possible points
gzipped `tar` archive submission	1
Submitted `slob.c`, patch file, source code of your test program in task 3 including a Makefile	4
Header block with team member names, program description, etc	2
Useful function headers (for functions changed) o Description of changes o Useful comment outline	4
README file (for task 5)	10
The responsibility of each team member is clearly stated (in README or comments)	3
The two new system calls are correctly implemented (can compile, can run, and can return correct results)	16
The **best-fit** algorithm is correctly implemented (can compile, can run, and can run correctly)	30
Your test program can compile and run	10
Clean presentation during the demo	20
Cites sources for borrowed code	0 points, but deduct up to 10 points for sources not cited