Writeup 4

Alex Hoffer
Nehemiah Edwards

CS 444
Spring 2017

Abstract

This fourth write-up details Alex and Nehemiah's work on Project 4. This project consists of adapting the *slob.c* file (which is a "Simple List of Blocks") to use a Best-Fit algorithm rather than a First-Fit one. Finally, after implementing the Best-Fit algorithm, this project consisted of building the Yocto kernel using both versions of the SLOB file and running a test script on them to evaluate and compare their performance.

Contents

1 Design

1.1 For Best-Fit Algorithm

We must adapt *slob.c*, which uses a First-Fit algorithm, to instead use a Best-Fit algorithm. In terms of algorithm design, our task is relatively simple. The difference between the First-Fit and Best-Fit algorithms in the context of the Linux SLOB is not massive. In fact, the only true place there are differences between these two algorithms is within the *slob_alloc* method, where new pages are allocated. If we look within this method, we can observe the way it is currently implemented to reflect a First-Fit approach: the method iterates through a list of pages and immediately allocates and places a page in the first place with enough of free space (e.g. "memory units"). So, our plan is to modify the behavior within this loop in the following way: if we are looking for the Best-Fit, this means we must loop through each page of the list (e.g. we'll have to get rid of the break statements that are in there currently) and check each and every page to see if it is as small as possible, meaning that it holds just enough, and not more, memory than we need. So we'll go through the whole list and set a page pointer variable to the pages we pass by that are less memory than before, so that when our loop terminates, we'll be left with a variable that points to the smallest page that fits our need.

1.2 For Analyzing How Many Units of Memory Are Unoccupied vs Occupied

Our task in figuring out how many units of memory are occupied is that we must keep a running count of how many pages have been allocated for use. Our plan for this is to keep a global variable that we add 1 to when a new page is allocated. From what I can tell, a new page is allocated in the *slob_alloc* method, specifically in the section of this method marked by the comment "Not enough space: must allocate a new page". We also have to subtract 1 from this global variable when a page is freed. Freeing a page occurs in the method *slob_free*, so we plan on decrementing this variable there. In terms of figuring out how many units of memory are unoccupied, our plan is a bit more complex, though the implementation is still just a few lines of code. In *slob_alloc*, note that there are references to three separate categories of memory units there are available: small, medium, and large, where the size of pages vary based on the category they belong to. When this method is called, we can use the method *list_for_each_entry* to iterate through each of these three lists, collecting how many available units of memory they have by adding each page's units to a global variable. Luckily for us, it looks like we can access how many units are free for the taking within a page by accessing the page structure's field *units*.

2 Version control log

History of versions

Version	Date	Author(s)	Changes
1	5.24.2017	Alex	modify original slob.c to keep running count of occupied pages
2	5.25.2017	Alex	modify slob to keep number of unused memory units
3	5.26.2017	Alex	modify slob to invoke system calls
4	5.27.2017	Alex	modify original slob.c to use best-fit
5	5.29.2017	Alex	Add old write up, need to adapt this to make final writeup

6	5.30.2017	Alex	Add final write up to a tarball directory with an assignment 4 patch file in preparation for tar.bz2'ing and submitting

3 Work log

3.1 Week 1

Check out the slob file in the mm directory. Figure out what it's doing- where are things being added, where are things being removed, and where exactly does the First-Fit algorithm come into play. Once we discovered that the real meat of the algorithm can be found in the *slob_alloc* method, we moved on to figuring out how to implement a system call. This part of the assignment took us the longest. The implementation of the system call wasn't hard, but what took a long time was since our virtual machine runs 32-bit we had to find all of the places within the kernel we'd need to modify in order to run our system call. The hidden one for us was in the syscall_32.tbl file found in arch/x86/syscalls/syscall_32.tbl. Then once we thought everything was working, we got a little pissed because for some reason using make -j4 all wasn't producing an object file! We ended up discovering that we needed to use our old friend make menuconfig to compile what we wanted.

3.2 Week 2

Week 2 started with us writing a test script. We already knew how to scp it over into the running Yocto kernel, so writing the script was fun. We just wrote a C file that looped 10 times, allocated a ton of memory, and then filled that memory with garbage. If we had more time, we would've written a script that used a bunch of processes running concurrently to see how that would've impacted our results. Within this test script, we called the two system calls we implemented that return how much memory is free and how much memory is currently in use, and then computed the fragmentation percentage based on this information. Week 2 ended with us changing *slob_alloc* within the slob.c file to use a Best-Fit algorithm. We were pleased to see that our design plan worked out exactly how we thought it would. Finally, we talked with Kevin to see what kind of fragmentation percentages we could expect from both algorithms and how we could improve our test script.

4 Questions

4.1 Main point?

We think the main point of this project was to get our hands dirty in the memory management folder of the virtual machine. Specifically, we've talked a lot about in class about how different operating systems handle memory resources and the differences between how certain algorithms choose pages of memory to provide to processes, but it's hard to understand what this abstract stuff means until you see it in its code manifestation. We also think Kevin wanted us to see firsthand how slow the Best-Fit algorithm was in practice. Finally, he probably wanted us to see how system calls were written and invoked, which was pretty cool.

4.2 Approach?

This is really the first and only assignment of ours that are design plans actually worked how we figured they would. Out of all of our projects, this one seemed to have the least amount of coding. Our approach in algorithm implementation was simply to iterate through all categories of free pages to search for the smallest possible page that was unoccupied.

4.3 Ensuring solution correctness?

We wrote a test script for execution from within the virtual machine that computed how many memory units were occupied and how many were unoccupied, computed the highest possible memory address for both of these categories of pages, and then used this information to compute and print out the fragmentation percentage.

4.4 What did we learn?

We learned that there is a good reason why the Linux SLOB uses First-Fit. Best-Fit was miserably slow and ended up far more fragmented than First-Fit. This is funny because Best-Fit is literally designed with the intention of decreasing fragmentation opportunities. If it wasn't for asking Kevin about why our Best-Fit was improbably more fragmented than our First-Fit, we'd probably assume that our implementation was flawed.