Project 2 - Kernel Memory Allocation

EECS 343 - Fall 2014

Important Dates

Out: Monday October 13, 2014

Due: Tuesday October 28, 2014 (11:59:59 PM CST)

Project Overview

The kernel generates and destroys small tables and buffers frequently during the course of execution, each of which requires dynamic memory allocation. Since most of the needed blocks are smaller than the typical machine page size, the page-level allocator is inappropriate for the task and a separate mechanism is used instead: the *kernel memory allocator*.

In this project you will implement, evaluate and compare a set of common allocation methods for use in a kernel memory allocator.

Educational Objectives

By the end of the project you should have a better grasp of some of the issues involved in memory management, in particular in the context of the kernel and the kernel memory allocator.

Background

The operating system must manage all the physical memory and allocate it to both other kernel subsystems and to user-level processes. At boot time the kernel reserves some memory for its own text and static data structures. The rest of memory is managed dynamically – the kernel allocates portions of memory to its various clients, which release it when they no longer need it.

In a page-based virtual memory system, the memory management subsystem maintains the mapping between virtual and physical pages. In SVR4, for example, the page-level allocator is implemented by the `get_page()` and `freepage()` routines. The page-level allocator has two principal clients: the paging system which is part of the virtual memory system and allocates pages to user processes, and the kernel memory allocator.

The *kernel memory allocator (KMA)* provides odd-sized buffers of memory to various kernel subsystems. Some example users of the KMA includes the pathname translation routine, which needs to allocate a buffer to copy the pathname from user space, and routines for creating/deleting PCB entries. Since most of these requests are much smaller than a page, the page-level allocator is inappropriate for the task and a separate mechanism is needed.

Some of the best KMA implementations allow for dynamic changes in the amount of memory allocated to the KMA. Thus, if the KMA runs out of memory it can just request an additional page from the page-level allocator. For this project, all your implementations will interact with the page-level allocator.

Specification

For this project you will implement, evaluate and compare the Resource Map and Buddy System algorithms. The correctness of your implementation of these two algorithms is worth a total of 90 points, accounting for all of the basic points. The evaluation report, described later in this section, is worth 10 points. Correct implementations of the extra credit algorithms will give you the specified number of extra credit points added to your score.

- *Resource Map:* You can choose between first-fit, best-fit or worst-fit (*required: 45 points*).
- *Buddy System:* For both Buddy systems (basic and lazy), use the same range of sizes (*required: 45 points*).
- *SVR4 Lazy Buddy:* Use the *slack* values discussed in the handout and only one memory pool (instead of two as employed in the SVR4 implementation) (*extra credit: 15 points*).
- *Power-of-two Free List:* The number and sizes of free lists is your choice (*extra credit: 15 points*).
- *McKusick-Karels:* For your implementation replace those tasks commonly implemented by macros by their C versions (*extra credit: 20 points*).

Each of your KMA implementations will consist of the following two functions (notice that some algorithms will ignore the size argument passed on to free):

```
void* kma_malloc (kma_size_t size)
void kma_free (void* ptr, kma_size_t size)
```

Your KMA implementations will use the provided page-level allocator to request new pages, free pages not needed, and report statistics on page usage including number of pages requested/freed/in-use. The provided page-level allocator (`kma_page.[hc]`) implements the following interface:

```
kma_page_t* get_page()
void free_page(kma_page_t* page)
kma_page_stat_t* page_stats()
```

To get you started, we are providing you with an example algorithm, Dummy. The example algorithm, appropriately named, dummy KMA (`kma_dummy.[hc]`) allocates a full page for any request (smaller than a page in size), independently of the size requested. To see Dummy in action you can simply `make kma_dummy` and run it (`./kma_dummy testsuite/1.trace`). If you run it using (`time (1)`) with the option for the portable output format (‘`time -p ./kma_dummy testsuite/1.trace`‘) your output will be similar to this (see `time (1)`'s man page for an explanation of the `real`, `user` and `sys` values):

```
[fabianb@localhost kma]\$ time -p ./kma_dummy testsuite/1.trace
Page Requested/Freed/In Use:  1497/ 1497/    0
real 0.02
user 0.01
sys 0.00
```

In addition to the different KMA implementations, *your submission should include a final report comparing the implemented algorithms* based on their utilization factor (ratio of total memory requested to that needed to satisfy the request) and their (worst/average) free/alloc latencies (you will have to instrument your own code to obtain these timing values). Please include a justification for your claimed worst/average performances with respect to each algorithm's design. This report should go in the DOC file included with the skeleton distribution.

Testing your code

To test your code, we have provided a driver program (kma.[hc]) that accepts as an argument a trace file containing a sequence of allocation and free commands.

We will use the included 5 trace files (of increasing difficulty and duration) in the skeleton's testsuite directory to test the correctness of your implementations when you submit your project. You must correctly complete all of the given 5 traces to receive the points for that algorithm.

We have also included a script to create additional trace files: `testsuite/generate_trace`. Run it with no arguments to see the options. When you generate a trace, it will plot (with gnuplot, if available[1]) the number of allocated bytes over the course of the trace.

When you are running in non-competition mode, the driver will output a data file that records, at every step in the trace, the current allocated and the number of bytes (pages $*$ page size) allocated by your algorithm. Run `make analyze` to generate the `kma_output.png` (bytes allocated plot) and `kma_waste.png` (inefficiency) plots.

The testsuite will also run the `testsuite/5.trace` file against your selected competition algorithm, so you can know how your algorithm is doing with respect to run time and efficiency.

Answers to common questions

- You don't need to serve requests for buffers larger than about a page size (just return NULL).
- You are allowed to have **one** static variable to store a pointer to a `kma_page_t` struct to serve as an entry point into your data structures. All other state must be stored in stack variables and in pages returned by *get_page*.
- You are **not** allowed to use *malloc* to obtain any memory. You must request memory for your algorithm's internal data structures using the provided *get_page* and *free_page* interface.
- You must dynamically manage your control data structures. That is, you cannot preallocate several pages to hold, for example, your buffersize arrays. You will need a method to grow these arrays with the number of requests. We test this by limiting the number of the pages to a smaller size so that you will run out of pages when you preallocated everything in advance. (Or, you can preallocate, but need a method to free them when you run out of space. In any event, you need to dynamically manage the size.)

For Extra Points

There are two ways to get extra points in this project.

1. You can implement more algorithms from the given list. (Obviously, your implementations must be correct.) (*You should clearly indicate those algorithms submitted for extra points in your evaluation report*).

[1] You can install gnuplot on the VM with '`sudo apt-get install gnuplot-x11`'. This is optional.

2. You can enter the performance contest with one of your implementations. You propose a contender (specify the executable name in caps – e.g. KMA_DUMMY – in the COMPETITION variable in the Makefile) that will be compared with other teams' submissions.

   Your algorithm will be evaluated on a combined factor of overall runtime and memory efficiency:

$$\texttt{Performance} = \texttt{Runtime} * (1 + \texttt{Inefficiency Penalty}) \tag{1}$$

$$\texttt{Inefficiency Penalty} = \frac{\sum_{i=0}^{n-1} \frac{p_i * s - b_i}{b_i}}{n-1} \tag{2}$$

   Where n is the number of calls to `kma_malloc` or `kma_free` in the trace file and s is the page size. p_i is the number of pages allocated through `get_page` and b_i is the number of bytes currently allocated through `kma_malloc`, after the ith event in the trace.

   This penalty evaluates your algorithm's average ratio between wasted and used memory.

   To enter the competition, your algorithm must correctly execute all provided traces. The 5th trace file will be used as the competition workload. When you get an algorithm working, submit an intermediate version of the project! We will post a scoreboard of the top scoring submissions on the website. We encourage you to submit versions of your projects as you work so you can see how you are doing against other teams. Each team will only be listed once at a time on the scoreboard, and subsequent project submissions will overwrite the previous performance score.

   The top scoring team will get 50 extra points, the next two teams (2nd and 3rd place) will receive 25 points each, and the next two teams (4th and 5th place) will get 15 extra points each.

Grading

To evaluate the correctness of your implementations, we will run the 5 traces that are included with the skeleton for each algorithm. We will also inspect your code and evaluate your final report. Correct implementations of the algorithms are each worth their given number of points (see list above). We will deduct points for the following reasons:

- 100 points!: It doesn't compile when we type 'make'
- up to 10 points: no documentation (DOC file)
- up to 5 points per algorithm: poorly self-documented/commented code
- 2 points per compiler warning

If you hand in your project late, we will deduct 10% from your final score per day or portion thereof. We will not accept submissions that are more than three days late.

Deliverables and hand-in instructions

Please **beware**, your hand-in code should run on any of the *distributed virtual machine image without any modifications*.

The deliverables for this project are:

1. Source code.
2. Evaluation report (DOC file) including your comparative analysis, as well as a description of any important design decisions you made while implementing the different allocation methods compared. Please make sure that these files are included in the handin.
3. If you attempted to pass the extra credit test cases, please make a note of it in your DOC file.

To hand-in your projects:

- Set your team name in Makefile: replace ‘whoami‘ with "yourNetid1+yourNetid2" for the TEAM variable.
- Invoke `make test-reg`, which builds the deliverable tar.gz and runs the test cases.
- **If and only if the test cases terminate, you may submit the handin.**
- Submission will be done through the dedicated webpage (which you can reach from the course site).
- You can re-submit the project as many times as you want before the deadline; simply follow the same procedure.
- A few minutes after submitting your handin on the submission site, you will receive an email confirming your submission, with the output of the testsuite running on our submission server. If you haven't received a confirmation within an hour, contact the TA to verify that your submission was received.

Good luck!