PA3: Implement and evaluate a user-space memory allocator

CMPSC 473, FALL 2017

Released on Oct. 05, 2017, due on October 26, 2017, 11:59:59pm

Ata Fatahi Baarzi, Aman Jain, Diman Tootaghaj, and Bhuvan Urgaonkar

1 Overview

In this project, you will implement your own versions of the user-space memory management functions `malloc()` and `free()`. User-space memory management has deep similarities with operating system memory management that we are covering in class. To distinguish your functions from `malloc()` and `free()`, you will call them `psumalloc()` and `psufree()`. Although your functions will be much simpler than the system's functions, it is our hope that they will still help you appreciate several complexities of user-space memory management. Somewhat less directly, perhaps, they will also help you better appreciate operating system virtual memory management (VMM) that we are covering in class. After testing your code, you will carry out simple experiments to measure specified performance metrics. You will describe your design and the outcome of your experimental evaluation in a short report.

2 Background

A memory allocator performs two main tasks:

- When needed by the process, it asks the operating system VMM to expand the process heap by calling either `sbrk()` or `mmap()`. Obviously, this occurs at the granularity of a page (make sure you understand this clearly). Similarly, it also releases back to the OS VMM pages that are not needed anymore.
- It carves out objects requested by the process from this memory. This involves managing a free list of memory "regions" and finding a contiguous chunk of memory among one of these free regions that is large enough for the user's request. By a region we mean a contiguous portion of memory (virtual or physical? Think!) among the pages assigned to the process. When the user later frees memory, it adds it back to the free list. Notice that this free list is a data structure that itself resides in the process heap.

The memory allocator is provided as part of a standard library and is *not part of the OS*. That is, the memory allocator operates entirely within the address space of a single process and knows nothing about which physical pages have

been allocated to this process or the mapping from virtual addresses to physical addresses. Make sure you understand this clearly. Recall the following definitions of `malloc()` and `free()`:

- `void *malloc(size_t size)` allocates `size` bytes and returns a pointer to the allocated memory.
- `void free(void *ptr)` frees the memory space pointed to by `ptr`, which must have been returned by a previous call to `malloc()` (or `calloc()` or `realloc()`). If `free(ptr)` has already been called before, undefined behaviour occurs. If `ptr` is `NULL`, no operation is performed.

You will begin by reading Chapter 17 of OS3 which describes how a generic user-space memory allocator is implemented. Reading till Section 17.3 should suffice for your purposes. The most important concept to understand is the role of the data structure called the free list. You will find the examples spanning Figures 17.4-17.7 particularly enlightening. Once you have understood this material well, you will be ready to begin your design and implementation.

3 Design and Implementation

You will implement your own memory allocator for the heap of a user-level process. Here are some simplifications you will make in your design (compared to what a real memory allocator does):

- When requesting memory from the OS, you will use the system call `mmap()`. Use man pages and/or online resources to learn about `mmap()`. Chapter 17 presents a small example usage of `mmap()` that you might find useful. In a later segment of the course on IO virtualization, if time permits, we will learn about some aspects of how the OS implements `mmap()`. An altenative way for requesting additional pages from the OS would be based on using `sbrk()` which we will not be using - it is arguably a bit more complex than `mmap()` and this complexity is only a distraction as far as our main interests in this project go.
- Although a real memory allocator generally requests additional memory from the OS in a "piecemeal" manner (i.e., only when it can't satisfy a request from the user), your memory allocator will call `mmap()` only one time (when it is first initialized).
- You are free to use any data structures you want to manage the free list. You must think about at least 3 different data structures (as we did in class when discussing how to implement the list of ready/runnable processes/threads for a CPU scheduler) for managing the free list, compare these for different relevant operations in terms of their runtimes, and describe this comparison in your report. Your data structure may not be an array (see more below). If you choose to implement a data structure that is not the best as per your own comparison, you must justify your choice.
- Finally, you will implement the following two policies for choosing a chunk of memory to satisfy an allocation request: (i) `best-fit` and (ii) `worst-fit`. These policies were mentioned briefly during Lecture 11. You can find descriptions of these policies in Section 17.3 of OS3.

We are now ready to specify the functions you will implement. These are:

- `int psumeminit(int algo, int sizeOfRegion)`: This function is called one time by the process that wants to use your memory allocator. `algo` represents which memory allocation policy should be used - 0 for `best-fit` and 1 for `worst-fit`. `sizeOfRegion` is the number of bytes that you request from the OS using `mmap`. `algo` & `sizeOfRegion` are defined as macros in the code we are providing you - see more below about "workloads." Note that you need to use this allocated memory for your own data structures as well; that is, your infrastructure for tracking the mapping from addresses to memory objects has to be placed in this region as well. Finally, the return value should be 0 upon successful completion of all the function's tasks and -1 otherwise.

  IMPORTANT: If you call `malloc()`, or any other related function, in any of your functions, you will get a 0 grade. Similarly, you will get a 0 grade if you use arrays for implementing your free list or other related data structures. If in doubt, ask the teaching staff immediately.

- `void *psumalloc(int size)`: This function is similar to `malloc()`. It takes as input the size in bytes of the object to be allocated and returns a pointer to the start of that object. The function returns `NULL` if there is not enough contiguous free space within `sizeOfRegion` allocated by `psumeminit()` to satisfy this request.

- `int psufree(void *ptr)`: This function frees the memory object that `ptr` points to. Just like with the standard `free()`, if `ptr` is `NULL`, then no operation is performed. The function returns 0 on success and -1 if `ptr` was not allocated by `psumalloc()`. If `ptr` is `NULL`, also return -1.

- `void psumemdump()`: This function is for your own debugging purposes. When invoked, it prints information about free regions of memory on the standard output or into a file (whichever you prefer).

You must provide these functions in a shared library named `psulibmem.so`. You will easily find good online resources on how to create such a library if you are not already familiar with this.

4 Experimental Evaluation

After testing your code, you will carry out a small experimental study of the performance of your memory allocators on 2 "workloads" (i.e., test programs) that we will provide. Below we describe these workloads.

4.1 Workloads

Each workload will be a small program that will allocate/deallocate memory using your functions. There are only two kinds of modifications you may make to a test program: (i) insert calls to `psumemdump()` for your own debugging, and (ii) code for performance measurement purposes (described in Section 4.2).

Keep in mind that we will evaluate your code using the unmodified workloads that we provide (and not workloads modified by you).
The workloads vary the size of their requests according to two different patterns:

- `small:` this patter consists of requesting all small objects (the sizes are random, distributed between 8 bytes and 256 bytes).
- `mixed:` this pattern consists of alternating between requesting a small object (64 bytes) and a large object (64 KB).

Each workload consists of 100 iterations. In each iteration, the workloads vary the order of `psumalloc` and `psufree` invocations as follows. They allocate 100 objects and then free about half of the objects (each chosen with a probability 0.5 independently of others). Upon finishing all iterations, they free all remaining objects. Combining these 2 workloads with our two allocation policies (`best-fit` and `worst-fit`), you will have a total of 4 experiments.

4.2 Measurements

You will create a mapping of `sizeOfRegion` bytes via your call to `mmap`. For each of the workloads, you will present the following measurements in the form of graphs or tables in your report:

- average, median, 25th percentile[1], and 75th percentile of `psumalloc()` and `psufree()` response times separately for `best-fit` and `worst-fit`. To measure response time, you will record the times before and after calling the function and take the difference.
- number of occasions when a `psumalloc()` could not be satisfied due to lack of enough memory.
- number of occasions when a `psumalloc()` could not be satisfied due to internal fragmentation.

5 Submission and Grading

Same as PA2, PA3 will be done in groups of 2. A group may choose either members repository as their workplace. Do remember to list both members names in all files you submit (including your report). Your project won't be graded if you don't have a name on your report. You are supposed to submit your assignment with your private GitHub repository. Therefore, accept the invitation to access your private repository. If you still need instructions for working with git or adding your teammate to your repository, refer to the included Git manual file. The TAs will grade you by inspecting your code, running some test cases for your code, and by looking at your report.

[1] A percentile (or a centile) is a measure used in statistics indicating the value below which a given percentage of observations in a group of observations fall. For example, the 20th percentile is the value (or score) below which 20% of the observations may be found.

5.1 Code and Report

You must push and commit to your private repository all your source files (*.c and *.h) and report file and a Makefile that the TAs will use to compile your code. You should also create a short README with instructions on how to compile and run your code. Note that you will not write a `main()` function for the code that you submit. You should implement one for your own testing, of course. The TAs will use automated scripts to link the library created by compiling your code against our test programs and then run these.
The quality and performance of your code will make up 60% of your grade. Roughly, the TAs will be looking at these aspects when assigning your grade: (i) the quality and clarity of your implementation (complemented by reading your description in your report), (ii) adherence to guidelines above about not using global arrays for free list implementation and not making use of system's `malloc()` or related functions, and (iii) correct treatment of the workloads. The remaining 40% of your grade will be based on a short included report file which contains your name(s). Your report should have the following two components:

- A few simple paragraphs describing the overall structure of your code and any important structures. For example, include a brief description of the data structures you use to map between addresses and memory objects. Describe your thoughts comparing 3 different data structures for implementing the free list, and why you picked one of these over the others. Describe how you handled any ambiguities in the specification. Finally, list any features that you did not implement or that you know are not working correctly.

- Graphs or tables reporting your experimental findings. If you have an explanation for some of the behavior you observe (e.g., why best-fit outperforms worst-fit for workload X), describe it.

6 Some final remarks

- **IMPORTANT: We will be comparing your codes for similarity and all parties involved in copying will receive a 0 grade on PA3 at the very least and an F grade in the course in the worst case. Outsourcing your code to an external entity is considered cheating. You will also be reported to the college of engineering disciplinary committee.**

- It is very important that you start early. You will very likely find this project to be more difficult (or at least more time consuming) than PA1 and PA2. Please seek as much as help as you need from the instructor and the TAs.

- Good luck and happy coding!