BARD CMSC 328: Embeddeqd Operating Systems

Evaluating Memory Allocators

Building upon the last lab where we implemented a dynamic memory allocator, this lab asks us to think critically about the design and implementation of memory allocators. The primary task in the last lab was to build a system for managing contiguous blocks of memory through an interface with only two functions: `bmalloc` and `bfree`. The challenge stemmed from the unknown nature of the stream of requests and releases for differently sized memory chunks. However, this kind of flexibility is expected by most modern programming languages where lists, and more sophisticated variable length data structures, are created and destroyed at runtime.

Beyond the programming challenge the last lab posed for you, a good dynamic memory allocator must overcome the following challenges:

- return the requested space, if available;
- prevent the memory blocks from overlapping;
- keep track of the new memory block;
- free the requested space;
- service the memory requests (both `malloc` and `free`) in a timely manner;
- avoid wasting space;
- avoid fragmentation;
- split free blocks;
- merge or coalesce freed blocks.

Empirical Evaluation

1. Write a function `report_space_usage` that will report in bytes and as a percentage: the total heap size, how much free memory is available in your heap, how much memory is currently being used, and how much of the heap is being used for record keeping.
2. Write a benchmark program to exercise your allocator. It should be demanding in terms of requests for and releases of memory. You might consider varying the size of the request blocks and interleaving requests and releases. How efficiently is your heap being used? How does this benchmark compare with a real application?
3. Write a test program to benchmark the cpu-time behavior of `bmalloc` and `bfree` and `malloc` and `free` using `gettimeofday` , `times`, and `clock`. How fast are the routines? Compare the relative performance of the two systems. What is difference between the three different types of time: wall, user and system time?
4. How could you use the `/proc/<pid>/status` to monitor memory use by a process? Write a program that takes a PID as an argument and returns the processes memory usage.

Design Evaluation

The design of systems software can often be thought about at three different levels:

Strategy	the high level goal (e.g. learn about embedded operating systems)
Policy	the intended behavior to achieve the strategy (e.g. complete hands-on labs)
Mechanism	the concrete implementation to enact the policy (e.g. lab #2)

1. Describe the strategies, policies and mechanisms of your dynamic memory allocator.
2. Like all algorithms, we can reflect on the time and space resources used by a memory allocator. How are these resources used in your system? Can you reason about your allocator asymptotically? How could time and space resources be used more efficiently?
3. The block structure we started with in the last lab might not be ideal; think about how it might be improved. Are there fields that need to be added or eliminated?
```
typedef struct block {
    size_t size;
    struct block* next;
    struct block* prev;
    int free;
} block_t;
```
4. How are your memory blocks organized? How is the list traversed when a new memory request is issued? And how is the list traversed when memory is released?
5. How would the pattern and sizes of requests (and releases) for memory impact the design of your allocator? Could your allocator exploit such a pattern?
6. The library Electric Fence provides an alternate `malloc` and `free`. Are those versions more or less efficient than your implementations? What is the purpose of electric fence?

Deliverables

lastname_lab3/

labreport.pdf	Answers to the questions and results of your benchmark experiments
bmem.c	Your malloc implementation along with a function to report space usage
bmembench.c	A benchmark for the allocator
memreporter.c	A report program that prints performance information about a process ID #