Carsten Hood
UIN: 922009787
CSCE 313-200
July 15, 2016

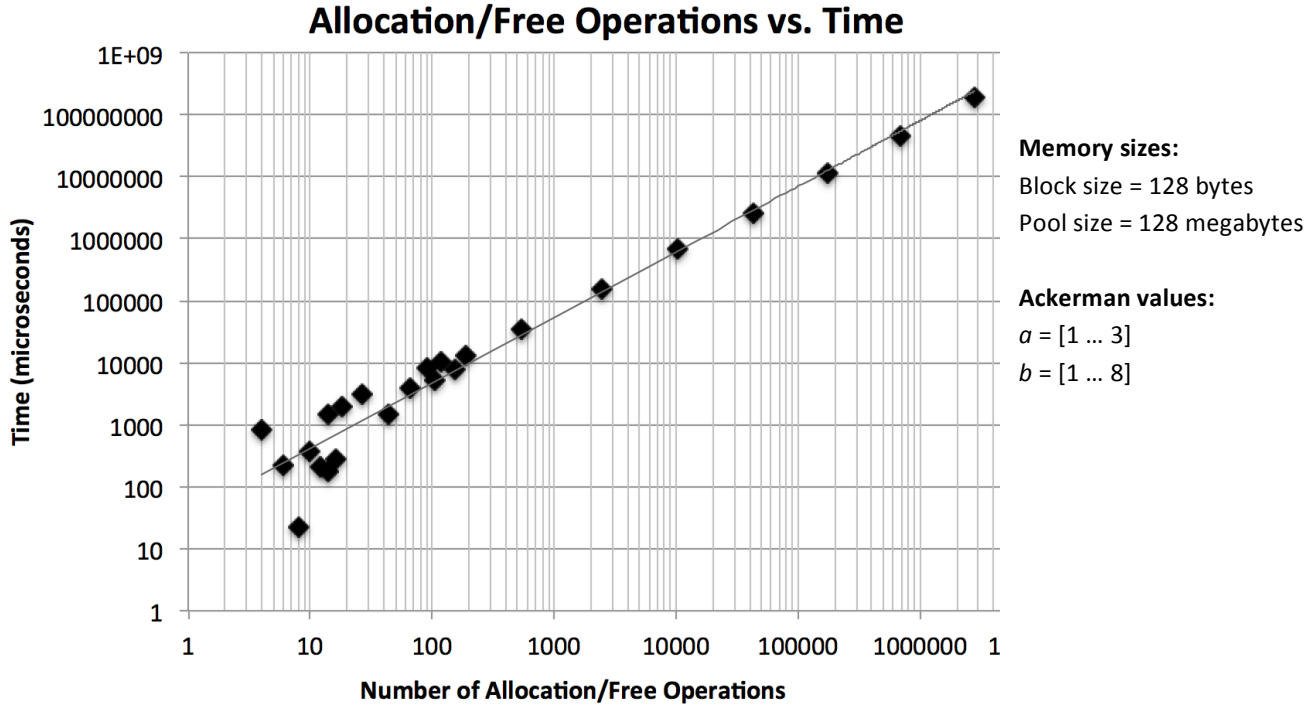
**Analysis for Machine Problem 1: A Simple Memory Allocator**

**Performance Overview**

The basic Fibonacci buddy-system allocator is a simple but effective memory-management tool. The merging and splitting sub-processes of the allocator's *free* and *allocate* functions – as well as the sub-process that locates new blocks' needed Fibonnaci indices – scale at $O(f)$, where $f$ is the number of free lists (or Fibonacci numbers). All other allocator processes take constant time. This efficiency is achieved partly by avoiding unnecessary list traversals. For example, memory blocks are added and removed from the front of free lists, and an *is-free* parameter associated with each block efficiently conveys its free/occupied state and its status as a valid block.

**Performance Plot**

The below graph illustrates the relationship between allocator runtime and performed *free*/*allocate* operations. Its data are produced by running the Ackerman test function for each combination of ($a$, $b$) input pairs where $a \in \{1, 2, 3\}$ and $b \in \{1, 2, 3, 4, 5, 6, 7, 8\}$. Because default block and pool size values result in occasional failed allocation attempts due to full or overly fragmented memory, the test uses an augmented pool size (128 megabytes) along with the default block size (128 bytes).




As shown in the plot, the Ackerman function used to test the allocator scales steeply with increased inputs (calling for both axes to be logarithmically scaled); however, the allocator itself appears to scale nearly proportionate to the number of allocation cycles.

**Bottlenecks & Inefficiencies**

One minor inefficiency of my implementation is that it marches incrementally through Fibonacci numbers to assign a correct free list index value to a newly requested block. I considered that a binary search of the Fibonacci array is usable here instead, but kept the current implementation because it is straightforward and its cost is relatively trivial. Other than this, no unexpected inefficiencies are readily apparent. Bottlenecks in the buddy-system allocator naturally occur when an *allocate* or *free* operation triggers a cascade of successive merging or splitting sub-processes that traverse much of the sequence of free lists. For example, demanding a relatively small block in a single-block memory pool necessitates a long series of recursive splitting operations. Subsequently deallocating such a block similarly results in a long chain of merging until the original pool is reinstated.

**Possible Improvement**

One possible remedy to this inefficiency may be to selectively avoid performing merging operations. For example, when a small memory block is released, the allocator might elect not to merge it with its buddy, perhaps depending on its size and some counter. This would improve performance by bypassing long merging processes and potentially expediting subsequent small-block allocation. However, this efficiency improvement would prolong memory fragmentation and possibly preclude large-block allocation, since merging frees larger memory segments. For such a simple implementation as this, the current reliable method is likely preferable.