CSC469 ASSIGNMENT 1

Shaun Memon and Binuri Walpitagamage

March 20th 2016

1. ALLOCATOR

The design of the allocator implemented follows the Hoard allocator referenced in the assignment.

1.1 DATA STRUCTURES

Heap

Each thread is mapped to a heap according to its thread id. When mallocing, the thread can only access memory blocks owned by its heap. A heap keeps track of its total size, allocated size as well as a pointer to the array superblocks of varying sizes. In order to update information in a heap structure, a thread must gain access to the heap's lock.

Superblock

The structure holds smaller memory blocks (referred to as slots) of a specified size that is to be allocated. It keeps track of the number of free slots available, the size of those free slots, a pointer to the free slots, as well as relevant information regarding its owner thread id and heap id. Superblocks are arranged in a doubly linked list structure so that it is easier to extract superblocks when allocating and deallocating memory. In order to maintain the integrity of the blocks, a "magic num" field is used as a check when freeing slots from a superblock.

Each superblock is owned by a thread which is the only entity allowed to allocate blocks from said superblock. However, a different thread will be able to free a block belonging to a different thread. Similar to the heap structure, a thread must gain access to the superblock lock before updating any information.

Slot

A slot is a memory block that is allocated to the user depending on the size request. The slot structures are arranged in a linked list so that the superblock can easily maintain it's free slots in the LIFO order thereby improving locality.

1.2 ALGORITHMS

mm_init(void)

Uses mem_init to initialize our "physical" memory by allocating a chunk of memory from the OS which is then used as our whole "physical memory" for the rest of the program. mem_sbrk is then used to expand the heap size (our virtual heap inside the memory chunk created in mem_init). Heap structures are then initialized within this first page of memory allocated in the heap. There is one heap for each processor and the first heap structure is the global heap which maintains the superblocks that can be reused by any thread. The rest of the heap structures maintain superblocks that can only be updated by its corresponding owner processor and thread.

mm_malloc(size_t sz)

Depending on the id of the thread that is calling the function, a heap is chosen by hashing the thread id to a processor heap number. If that heap contains a superblock belonging to that thread with free slots satisfying the size requirement, then the first free slot from that superblock is allocated. However, if the chosen heap does not contain a superblock with sufficient space, then the global heap is searched to see if a satisfying superblock is available to be reused and is transferred to the chosen heap. If a superblock is not available in the global heap as well, then the mem_sbrk method is used to expand the heap memory allocated to create a new superblock for the heap from which the first slot is allocated to the calling thread.

mm_free(void *ptr)

In order to access metadata on the block of memory freed, we first have to locate the superblock to which the freed pointer belongs. In our implementation, this is done by aligning the pointer address to the nearest lower page boundary as superblocks are allocated to be the size of one page. The integrity of the calculation and the superblock is ensured by checking for the "magic num" field upon which the lock of the superblock as well as the lock of it's owner heap is acquired. The slot that was freed is added to the beginning of superblocks list of free slots and the free slot count is incremented.

If the superblock belongs to a the global heap, locks are released and the the function returns. However, if that is not the case, then we check to see if a superblock from the current heap should be moved to the global heap by checking the emptiness thresholds as described in the Hoard paper. We take the least recently used block of the size class that is being freed and transfer that block to the appropriate size class of the global heap. Another enhancement that we could have performed is to keep superblocks ordered by their fullness groups and move the least full superblock across any size group to the global heap.

1.3 ALTERNATIVES CONSIDERED

One alternative that we considered for the implementation is the use of a bitmap to keep track of the free slot usage in the superblocks instead of a linked list. Although a bitmap would have reduced the space usage, there was no method to keep track of the LIFO order of the blocks thereby reducing locality. Therefore, we decided that the use of a linked list was the better choice for the assignment.
Another enhancement that the Hoard paper calls for is storing supernodes by their fullness in fullness groups. This would ensure that the memory is allocated from the fullest qualifying superblock leaving emptier ones to be transferred to the global heap when necessary. We developed the allocator grouping superblocks by their size class instead. To combat this we thought about keeping the size specific superblock linked lists in fullness order, most full superblocks being towards the head of the linked list and hence they would be allocated from first. However, time constraints and a lack of testing at the end prevented us from making that change.

2. PERFORMANCE ANALYSIS

The main focus of the Hoard implementation was to avoid false sharing. Our implementation follows the same focus by using separate heaps to maintain memory allocated to different threads so that when allocating memory blocks, no two threads will share the same cache line. Unlike the hoard implementation, this principle is also followed when freeing memory blocks since we only transfer superblocks to the global heap when empty thereby avoiding two threads using the same superblock. Moreover, the ownership of a heap by a thread also reduces the lock contention since threads only acquire the lock of the heaps belonging to its own heap. Lock contention may occur when freeing since a different thread is allowed to free a memory block belonging to a different

Internal and external fragmentation is another factor tackled by memory management systems. For a2alloc, each heap contains superblocks with memory blocks of a specific size class. These size classes are restricted to powers of 2 thereby bounding the worst case internal fragmentation to a factor of 2[1]. On the other hand, since memory blocks of a certain size are given out from a specific superblock with fixed sized memory blocks, freed memory blocks are reused in the same manner as newly created blocks. Since these freed slots are placed in LIFO order, we improve the locality of the memory slots.

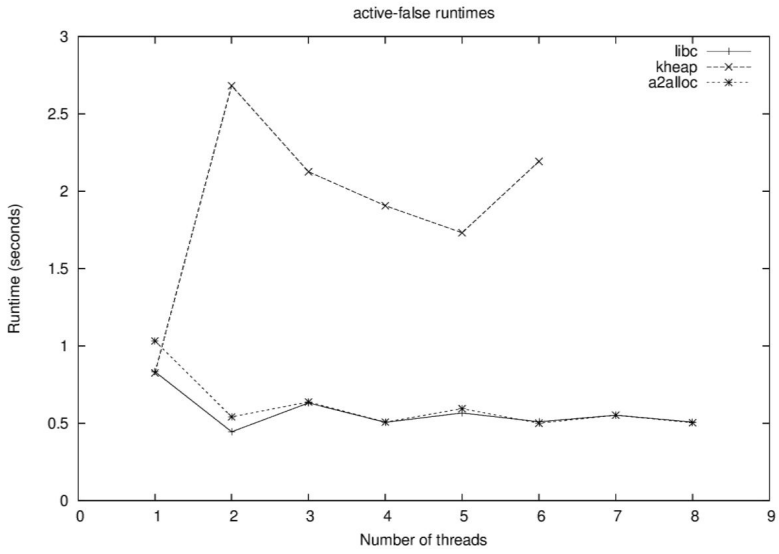
Active False Runtimes

~/a2/benchmarks/cache-thrash$ ./graphtests.pl

name = libc
   scalability score 0.428
   fragmentation score = 0.000

name = kheap
   scalability score 0.092
   fragmentation score = 6.750

name = a2alloc
   scalability score 0.493
   fragmentation score = 6.500



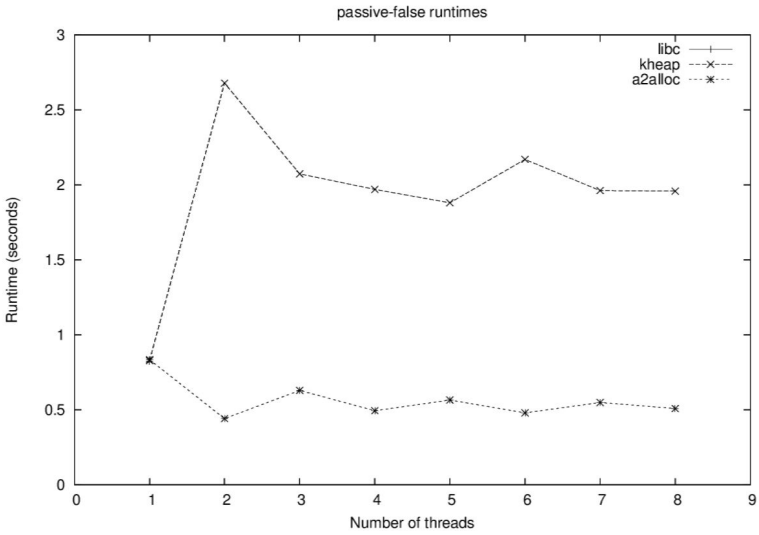
Passive False Runtimes

~/a2/benchmarks/cache-scratch$ ./graphtests.pl

name = libc
  scalability score 0.000
  fragmentation score = 0.000

name = kheap
  scalability score 0.101
  fragmentation score = 11.625

name = a2alloc
  scalability score 0.435
  fragmentation score = 7.469



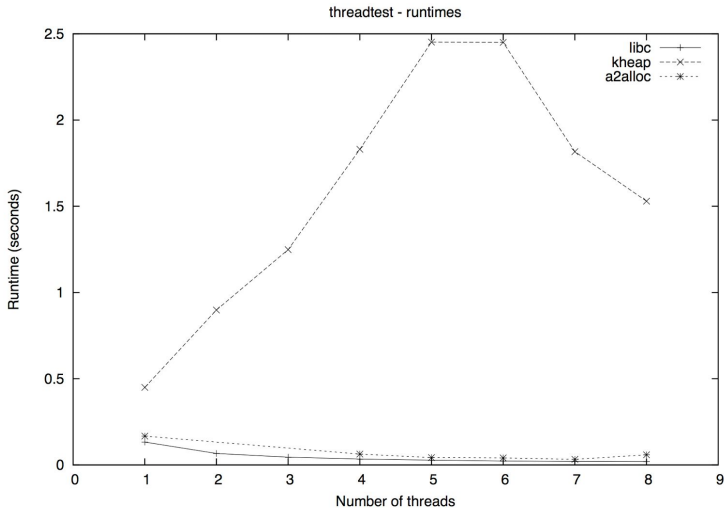
Threadtest Runtimes

---

name = libc
  scalability score 0.978
  fragmentation score = 33.000

name = kheap
  scalability score 0.089
  fragmentation score = 829.250

name = a2alloc
  scalability score 0.480
  fragmentation score = 9.125



REFERENCES

[1] Hoard: A Scalable Memory Allocator for Multithreaded Applications, E.D. Berger, K.S. McKinley, R. D. Blumofe and P. R. Wilson, in ASPLOS IX, 2000