Lock & Lockfree malloc

Jack Xu

Feb 9, 2024

1 Intro

2 Locked algorithm

2.1 Data Representation

The data in the locked version is represented in terms of "chunks."

Each chunk consists of the following, borrowing inspiration from ptmalloc:

1. Previous size + occupied bit (63 bits of size, 1 bit flag corresponding to occupation)
2. Current size + occupied bit (63 bits of size, 1 bit flag corresponding to occupation)
3. malloc'ed region that is at least max(alloc'd size, 16 bytes) big.
   (a) If current chunk is free, this contains 2 pointers
      i. previous free chunk (void *)
      ii. next free chunk (void *)
   (b) Otherwise, this contains the malloc'd region.

Globally, these following variables keep track of state.

1. Pointers to head, tail chunks
2. Pointers to beginning of allocated memory, end of allocated memory. This helps debugging in verifying that tail's end aligns with the end of allocated memory.

2.2 Algorithm

2.2.1 Malloc

Malloc uses some heuristics to boost performance and minimize fragmentation. They will be mentioned after the high level overview of what malloc does:

Finding a best fit chunk:

1. The allocation size is rounded up to a multiple of 16 (accounting for header size)
2. The algorithm looks for the head of the first free chunk. This will give it access to the linked list of free chunks (as recorded in data structure of free chunks)
3. The algo scans through the free chunks, recording the chunk which will both fit and is the smallest possible size. On ties, the one with smallest memory address is chosen.
   (a) If a chunk is not found, then one will be allocated.
      i. The allocation logic works as following:
      ii. There is some INIT_SIZE, for which to allocate.
      iii. If INIT_SIZE is not large enough, INIT_SIZE is doubled until it is bigger than the requested allocation.
      iv. The resultant size is asked by sbrk and the tail, and mem_end are updated accordingly.
      v. The resultant tail chunk is returned
4. On fitting to the chunk, it uses heuristics to decide where in the chunk it will fit (left or right side). This is to minimize fragmentation.
   (a) If the chunk is not big enough to be split into two chunks and still fit the requested allocation, the entire chunk is occupied.
   (b) If the chunk is big enough to be split into two chunks, and will still fit the requested allocation, it uses the following rules to determine whether the left or the right chunk will be used
      i. If the previous or next chunk is NULL, meaning this is the beginning or end, the allocation is made on the LEFT side.
      ii. Otherwise, if the left chunk is bigger than the right chunk, the allocation is made on the RIGHT side
      iii. Otherwise the allocation is made on the left side.

Some optimizations that were made

1. Aligning on boundaries:
   (a) Allocation of chunk sizes are made such that all chunks sizes (including overhead from embedded data) are a multiple of 16 bytes.
2. Allocation on left or right side of free chunks
   (a) This would minimize fragmentation in the scenarios where a free chunk borders two chunks of different sizes. If say, an allocation of 48 is to be made in a chunk of 64 bytes, which neighbors a chunk of 1024 bytes on the left and 32 bytes on the right, such an allocation would be made on the RIGHT. This is so when the 1024 is freed, it would have the maximal amount of contiguous memory.
3. Maintaining a free list
   (a) This speeds up the process of looking for a free chunk, and is good for frequent, small sized allocations as seen in the tests with random freeing in between

Some areas of improvement include, but which were prohibited by time constraints, include

1. Maintaining the head of the free list. This would be a simple optimization that would rid the need to look for the beginning of the free list
2. Adding thread independent areas to minimize contention
3. Adding specific regions for allocating fixed size increments. This can take on a separate header format with less overhead (currently it is 16 bytes of overhead).

2.2.2 Free

Upon freeing a block, the following occur, in the following order:

1. Chunk is set to unoccupied
2. The left and right adjacent chunks are checked to see if they are occupied
   (a) If left is free, the left is merged with the current chunk.
   (b) If the right is free, the right is meregd with the current chunk.
3. Then the appropriate next, prev free chunks pointers are updated, including updating the next pointer of the previous free chunk, and the previous pointer of the next free chunk.

3 Lockfree algorithm

Unfortunately due to time constraints, a working version was not achieved. However, the considerations which went into how to construct such an algorithm and how such an algorithm would work are included below:

Considerations:

1. The reasons for the need for a lockfree algorithm is that multiple threads, can contend for the same resources, leading to linear increase in runtime (and possibly more) with increasing number of threads. There are several mechanisms through which this occurs
   (a) Lock contention - the naive way of using locks would be to implement a global lock, leading to linearly scaling runtimes + overhead of locking/unlocking. Other forms of locking will also lead to longer wait times, such as spin waits.
   (b) False sharing, cache contention - When multiple threads use memory close in location/memory far apart in location, it causes false sharing/cache eviction which degrades performance

Basic intuition:

1. The heap, is, without engineering, a multi producer and multi consumer resource. Such resources inherently need to implement locks in order to avoid race conditions.
2. However, other design patterns such as single producer, multi consumer have lock-free designs which will eliminate lock contention.
3. To implement single producer, multiple consumer, there would need to be single producer of free blocks, and multiple consumers of it
   (a) This would use a single-producer, multiple consumer linked list/pool
   (b) There would be a single thread responsible for recycling free blocks into a shared pool
   (c) The multiple consumer threads can avoid using locks, using a similar mechanism to how linked lists may be inserted/modified by multiple consumers using CAS and atomic operations.

Description of algorithm

1. Each thread will start with its own (large) chunk of memory. This is its own area for splitting and allocating memory if two conditions are met
   (a) The requested allocation is not larger than some threshold
   (b) There is remaining area in this thread local storage
2. Each time a piece of memory is freed, it is recycled by a singular thread working in the backgroud which will garbage collect free memory. This free memory is placed into a queue like object.
3. When allocations of large memory size is made, mmap may be called instead, and if threads are out of memory, they may pull from this single producer multiple consumer queue of the garbage collector thread

Performance considerations

1. To minimize false sharing, allocations can be made to align with memory boundaries for cache lines. Additionally, threads will largely work in their own thread local memory (only pulling from garbage collector queue whenever they run out)
2. One may pre-allocate regions of thread local memory for allocations of specific, common sizes, depending on program needs
3. Threads that frequently ask for allocations from the SPMC queue may have their own thread local space adaptively increased. However, this isn't guaranteed to be continuous, so it would lead to added complexity in managing thread local storage.

4 Discussion of Performance

As a working version of the lockless was not achieved, and performance should be compared relatively, it would incorrect for me to point to the results of the locked version in terms of their magnitude. However, several speculations are made as to what may be observed performance differences:

For reasonable parameterizations of the lockless version, for any reasonable number of threads (say between 4 and 20), runtime might scale like

$$T_{\text{lockless}} = T_0 \frac{W}{W_0} \frac{1}{r \cdot t}$$

Where T_0 is the benchmarked time for single thread, no locking or threading overhead, W is the amount of allocation being done, W_0 is the original amount of allocation being done, r is an efficiency factor between 0.5 and 0.9, and t is the number of threads.

For $W = t \cdot W_0$, we have

$$T_{\text{lockless}} = \frac{1}{r} T_0$$

due to the inherent overhead of managing such a structure.

However, for a naive locked approach, the time would be

$$T_{\text{locked}} = \alpha W + T_0 \frac{W}{W_0}$$

Where α describes the overhead with each global lock/unlock operation. The equation comes from the fact the global lock causes things to be serial, and with each allocation there is lock/unlock overhead.

For $W = t \cdot W_0$, this translates into

$$T_{\text{locked}} = \alpha t W_0 + t T_0$$

.

In other words, it scales linearly plus an overhead factor due to excessive locking.

One would expect that for a lesser number of threads (1 to 4), locked might perform better due to overhead of managing the lockless version, and similarly with an excessive number of threads (20+), it might perform better than the locked version, but it will observe performance degradation and we no longer observe a constant time

$$T_{\text{lockless}} = \frac{1}{r} T_0$$

but rather

$$T_{\text{lockless}} = \frac{1}{r} T_0 + g(t)$$

where $g(t)$ is a monotonically increasing function associated with the overhead of managing so many threads.