ID1206 Seminar 2 Report

Cem Cebeci

November 28, 2019

1 Benchmark Description

The same benchmark program will be used to assess different implementations of the `palloc` and `pree` methods. The benchmark should be mimicking the memory requests of an average user program as closely as possible. Since it is not very easy to determine how an average program uses the memory, I decided to follow the assumptions made in the "my malloc" assignment and use the same memory request characteristics as that assignment.

1.1 Benchmark Implementation

The benchmark uses a buffer of `void` pointers. The length of the buffer should be passed as the main method's first argument. At each iteration, we pick a random pointer in the buffer (with a uniform distribution). If that pointer points to a memory address, we `pree` the memory address and set the pointer to `NULL`. If the pointer is `NULL`, we call `palloc` to allocate a new block of some size and store the new block's address in our pointer. The number of iterations the benchmark should perform should be passed as the main method's second argument.

Naturally, we should determine the size of our new block before calling `palloc`. This is done randomly as well. Clearly, we need to pick a maximum and a minimum block size to obtain interpretable results. These two values are defined in `rand.c` as `MAX` and `MIN`. Furthermore, the sizes of the requested blocks are not distributed uniformly. Whenever we need pick a size, we do so complying with:

$$size = \texttt{MAX}/e^r$$

where r is distributed uniformly between 0 and $ln(MAX/MIN)$.

After the specified number of iterations are performed, the benchmark should report its observations. Out interest is in the length of the free block list and

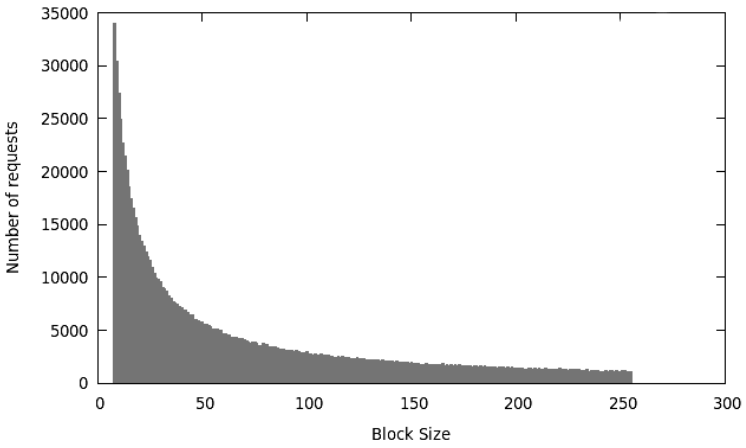

Figure 1: A histogram of the distribution with 10000 samples

the distribution of the sizes of the free blocks. Being a user program, the benchmark can not (and should not) know about the state of the free list `palloc` and `pree` maintain for themselves. To solve that issue, I have added a method `flist_data` to `ptmall.c` and `ptmall.h`. `flist_data` returns the length of the free block list and an array describing the distribution of the free blocks.

Then, the benchmark

- Prints the **number of** `palloc` **and** `pree` **calls** on the console.
- Prints the **free block list length** on the console.
- Prints the **total free space** on the console.
- Writes the **distribution of free blocks** to `dist.dat`.

1.2 Benchmark Characteristics

The benchmark implementation has, as the summary of Section 1.1, three main characteristics:

1. It requests much more small blocks than big blocks.
2. It frees almost every block eventually.
3. At a given time, the benchmark has around $buffersize/2$ blocks allocated.

The first characteristic is created by the distribution of the block sizes, the second characteristic is created by the fact that any non-null pointer the process goes through is freed and the third characteristic is a result of the uniform selection of buffer locations.

2 The First Implementation

2.1 Description

The `palloc` and `pree` methods both work with a large pre-allocated block of memory. This serves two purposes. Firstly, once the system is initialized, it will not have to make any more system calls, reducing the runtime dramatically. Secondly, the system will be completely abstracted from the operating system once it is initialized, giving us complete freedom with the large memory we had the operating system allocate for us.

Here are some implementation details worth mentioning for the first implementation:

- The system has a 24 Byte overhead for each block.
- The total memory this system can allocate is 64KB, that includes the overhead as well.
- Every block allocated by this system will be 8 Byte aligned.
- The `pree` method runs in constant time but the `palloc` is not guaranteed to.
- After a while, the system will most likely have a large amount of internal fragmentation.

2.1.1 The Freelist

The system has a linked list that keeps track of all of the free blocks available, this linked list will be referred to as the freelist. To maintain the freelist, we attach 24 Byte headers to the start of each memory block, which we keep completely hidden from the user. When `palloc` is called, it will find and return the first block that is of sufficient length, split it if it is large enough, detach it from the freelist and return the block to the user. `palloc` also initializes the system when it is first called. When `pree` is called, it will add the block to the freelist. The rest of the methods serve the purpose of maintaining the freelist.

2.1.2 The Implicit List

The system maintains another data structure that is not crucial for the first implementation but will be important for the coalescing process. This second data structure will be referred to as the implicit list. Because it's elements do not have explicit links to each other but they can calculate their neighbours' addresses using size information about themselves and their predecessors. Therefore we can operate with this structure as if it was a linked list. The methods in the first implementation are implemented in a way that preserves the structure of the implicit list as well. Though it is in no way important or useful to the first implementation.

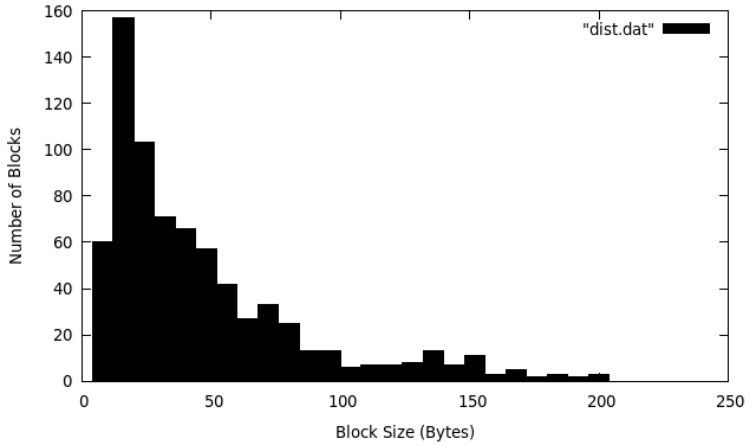

Figure 2: A histogram of free block size distribution

2.2 Observations

Here are some observations after runnning several benchmarks with different sizes:

- As Figure 2 shows, most of the blocks in the freelist are of small size.
- The program runs out of memory when loop count exceeds around 2000.
- The loop count at which the program runs out of memory changes very slowly with respect to the buffer size.
- Freelist's length is usually around %80 of the number of calls to `pree`.

2.3 Interpretation

The first implementation we built is terribly inefficient. About %80 of the blocks we freed are not being used again. In the end, we end up with a program that has 40 KB of it's 64 KB allocated memory free but can not allocate memory for a block of 16 bytes.

Our observations suggest that this waste of memory is due to severe internal fragmentation. The number of allocated blocks we have at a time does not change our limitations that much, therefore the problem must be that we can not find sufficiently large free blocks in the free list. That explanation is also supported by the fact that a great portion of our free blocks are very small blocks.

The cause for the extreme internal fragmentation our implementation faces is that it splits large blocks whenever it can but never merges the small ones.

Even if we were to increase our total memory size, the system is bound to break that memory into small blocks and end up with a severely fragmented memory.

Without any doubt, the first upgrade we have to make to this system is to introduce the **coalescing** of the blocks. As the system is bound to waste its memory without that mechanism.

3 Modifications

In this section, we will design, implement and test some modifications on the first implementation. The goals we will try to accomplish are:

- Reduce internal fragmentation.
- Reduce the memory overhead, in the form of headers and alignment.
- Keep `palloc` and `pree` running in constant time.

3.1 Coalescing Free Blocks

As discussed in Section 2.3, having some way of merging small free blocks is crucial to the memory efficiency of our system. Merging consequent free blocks is relatively simple to implement and fast. Our implementation, in fact, will be able to merge blocks in $O(logn)$ time. Merging nonconsequent blocks, however, would require us to move blocks around to have the free blocks together. Therefore we will merge only consequent blocks, wherever possible.

The `merge` method is called whenever we insert a new block to the freelist and it merges the newly added block with all the suitable blocks around it. It does so in a recursive fashion, therefore it does not in constant time but still it runs in $O(logn)$ time, which is acceptable.

3.2 Multiple Free Lists

As an extension to our implementation, we will keep a number of additional freelists. One of our assumptions about the average program's behaviour is that it requests small blocks much more frequent than large blocks. Then, allocating blocks of optimal sizes to small requests should reduce the internal fragmentation dramatically. That can be achieved by keeping freelists that contain blocks exactly of sizes of multiples of 8 (since our allocation is 8-Byte aligned). It should make our `find` method somewhat faster as well, since we know exactly where to finds blocks of sufficient size for small sizes.

In order to implement this, we need to change free list to be an array of lists and modify the `detach` and `insert` methods to find the appropriate list to insert(detach) the nodes to(from). We should also change the `find` method to check the additional lists.

It is worth mentioning that how we maintain the freelist(s) is completely independent from the implementation of `merge`.`merge` detaches the blocks to be merged as it merges them and `insert` adds the final merged block to the freelist(s).

4 Results

We will now run our benchmark on our new system and measure the success of our attempts to increase its memory efficiency. The following results are obtained from a benchmark with a buffer size of 300 and a loop count of 1,000,000.

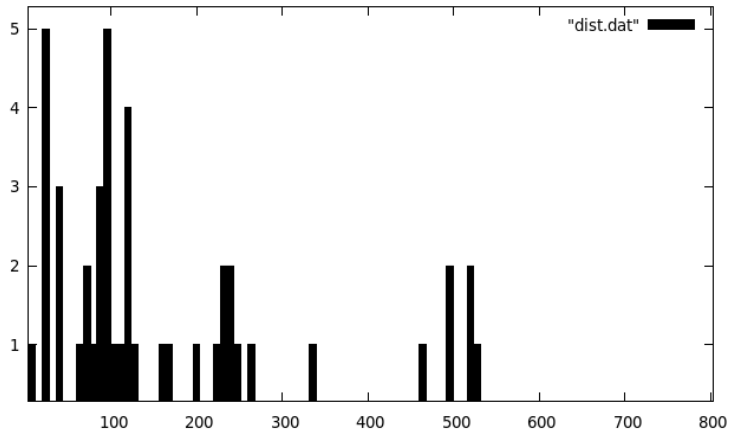

Figure 3: A histogram of free block size distribution

- The freelist's (combined) length is 50.
- The system's free block distribution is as shown in Figure 3, very big blocks are not shown in the figure, the average block length is 925 Bytes.
- The system still has 46KB has free memory, with a great portion of it in the form of large blocks.
- The system's ability to allocate memory is limited by the buffer size, not the loop count, which is about 2K blocks.

With those observations, we can conclude that we have solved the severe fragmentation problem we had in the first implementation. The system is still not optimal and it has lots of possible upgrades, but it is usable and has very high memory utilization. And it still runs all of its services in O(logn) time.