University of Siena
Department of Engineering

Internship Report : Memory Allocator
17 oct – 3 nov 2016

Presented to Prof. Roberto Giorgi
By AbdAllah Aly Saad

Table of Contents

1. Objective (Abstract)

Write a fast and optimized memory allocator for allocating memory blocks in an efficiently matter. The memory allocator should be thread-safe, with detailed error reporting.

At this stage, a basic memory allocator has been developed without considering much of optimization or multi-threaded environment requirements, As the sole purpose of the developed memory allocator is to understand how memory allocators work and designed, how to detect bottlenecks and optimized them.

2. Design Overview

In order to write an efficient memory allocator there are several criterias to be covered [DL2000], some are :

- Maximizing Compatibility : The allocator should follow the ANSI/POSIX standards and be plug-compatible with other allocators.
- Minimizing Space : The allocator should minimize the fragmentation as much as possible and reuse available memory chunks without obtaining more memory from the system.
- Minimizing Time : functions for allocator operations such as ***malloc()*** and ***free()*** should run as fast as possible.
- Maximizing Locality : by guaranteeing locality of chunks by getting allocated near to each other, which helps to minimize page and cache misses during memory execution.
- Maximizing Error Detection : Provide the possibility for detecting memory corruption such as overwriting memory, multiple frees or any other issue.

However, The primary goal for any memory allocator must be minimizing space wastage [DL2000] by minimizing fragmentation both internal and external. The allocator presented here trys to achieve this strategy by following the best-fit search approach. Although some of the optimizations could also contribute to this wastage such as chunks alignment by skipping over some bytes to achieve allocate aligned chunks [DL2000].

While efficiency is about the performance of the allocator itself,the allocator also has to be also effective in the sense that, it should be able to reuse all the freed blocks allocated by the user or release it to the OS.

It is also important to make sure that the wasted memory is minimal, which Is the memory allocated by the memory allocator but not used [TLSF1]. It is also a must to keep track of released memory blocks in order to reuse them to answer later allocation requests in order to avoid memory exhaustion [TLSF1].

[TLSF1] also specifies two general categories of memory allocators, the first is explicit allocation and deallocation, which is done by explicitly calling the memory allocator primitives such as ***malloc()*** and ***free()***, the second is implicit memory deallocation or garbage collection, where the memory allocator is responsible for collecting the memory blocks that have been allocated but are not needed anymore. Different categories of memory allocators are presented in the following table :

Sequential Fits	The most basic memory allocation mechanism, by sequentially or free blocks in a singly or doubly linked list, such as first-fit which searches the free list for the first block whose size is equal or greater than the requested size, next-fit, and best-fit which selects the block with the exact or closest size to the requested one.
Segregated Free Lists	Uses a set of free lists with different predefined sizes or size range. an example is fast-fit, which uses an array for small-size free lists and a binary tree for larger sizes.
Buddy Systems	A specific class of segregated free lists with only $\boldsymbol{log_2(H)}$ lists, with ***H*** being the heap size, as the heap could be only split in powers of two. This restriction provides efficient splitting and merging operations, however it also causes high memory fragmentation. One example is the Binary-buddy, which has always been considered as a real-time allocator.
Indexed Fits	Indexing of free blocks by Using advanced data structures.
Bitmap Fits	Using bitmaps to rapidly locate free blocks without performing an exhaustive search. An example is half fit.
Hybrid allocators	Use different mechanisms to overcome some of the flaws of the memory allocator. One example is DLmalloc which implements a good fit jointly with some heuristics to speed up the operations as well as to reduce fragmentation.

Table 1, memory allocators categories

Some allocators are difficult to be assigned to a specific category as they use more than one mechanism [TLSF]. Another important concept of memory allocators issues is fragmentation, which is not being able to reuse free memory, and they are defined as two categories [TLSF1] as shown in the following table :

Internal fragmentation	Happens when the allocator returns a memory block with a size larger than the one that was requested because of block round-up, memory alignment, inability to handle the remaining memory, etc. It is caused only the the allocator implementation [TLSF1].
External fragmentation	When free memory is available but the free blocks are not large enough to satisfy the allocation request. It is caused due to a combination of the allocation policy and the user request sequence [TLSF1].

Table 2, types of memory fragmentation

2.1 Design Choices

The allocation policy is a best-fit, where a memory allocation request should be assigned a memory block of the same size as requested. By analyzing the flaws or challenges faced by every memory allocator, a list of strategies for different scenarios is made, these strategies may not be the best as they have to be bench-marked and analyzed and they are not part of the final design unless proven to be optimal or near optimal, while Other strategies should be added to the later modified and improved memory allocator as shown in section 5. The table below summarizes each scenario and the proposed strategy:

Internal Fragmentation	Using best-fit strategy should as lower internal fragmentation as much as possible.
Locating a free block	Just like Dlmalloc, a mapping function is used to locate a suitable list, while the Dlmalloc uses a quick and optimized function [TLSF1].

3. Implementation

“A key factor in an allocator is the data structure it uses to store information about free blocks” [TLSF1]. The free block data structure ***memBlock*** is used to hold information such as the size class and the next memory aligned ***memBlock.*** When a memory block is requested, ***findBlock()*** function is used to search for a free list of the requested size class, if found then the list is searched for a free block otherwise a new list is added, if a free block is found, then it is returned, otherwise, the ***malloc()*** calls ***sbrk()*** to request more memory from the system. Once a memory block is freed using ***free()***, a list of the size class should be available and the free block is added to this list. Free lists are inserted in an increasing order to simplify the lists crawling.

```
#include <unistd.h> //sbrk
#include <stdio.h> // printf
#include <string.h> //memset
#include <pthread.h> // pthread_mutex_t

void* base;
pthread_mutex_t malloc_lock;

// basic memory block allocated/free
typedef struct  memBlock memBlock;
struct memBlock
{
	int size; // lowest bit is not a sign but mark for free or not
	memBlock* nextMemBlock;
};
```

```c
// explicit free blocks list of a specific class/size
typedef struct  blockList blockList;
struct blockList
{
	int size;
	memBlock* headBlock;
	blockList* nextList;
};

blockList* headBlockList;

memBlock* findBlock(int size)
{
	blockList* head = headBlockList;

	while (head && !(head->size == size)) // search for a list of this specific class/size
	{
		head = head->nextList;
	}

	if(head && head->headBlock) // list found with an available block
	{
		memBlock* found = head->headBlock;
		 head->headBlock = found->nextMemBlock;
		return found;
	}
	else if (!head) // no list is found at all of this class/size !
	{
		// add a new list head with this new size class !
		blockList* newList = (blockList*)sbrk(0);

		if ( sbrk( sizeof(blockList)) == (void*) -1)
		{
			#ifdef DEBUG
			fprintf(stderr, "couldn;t create a new list with no available memory\n");
			#endif
			return NULL;
		}
		newList->size = size;
		newList->headBlock = NULL;
		newList->nextList = NULL;

		// inster it into the linked list :)
		if(!headBlockList)
		{
			headBlockList = newList;
			#ifdef DEBUG
			fprintf(stderr, "new list is added\n");
			#endif
		}
```

```
          else if (headBlockList->size > size)
          {
               newList->nextList = headBlockList;
               headBlockList = newList;
          }
          else
          {
               head = headBlockList;
               while (head)
               {
                    if(!head->nextList) // it is the last
                        {
                              head->nextList = newList;
                              break;
                        }
                    else if(head->nextList->size > size) // if a bigger size then we take its place
                    {
                         newList->nextList = head->nextList;
                         head->nextList = newList;
                         break;
                    }
                    head = head->nextList;
               }
          }
          #ifdef DEBUG
          fprintf(stderr, "instereted new class list \n");
          #endif
     }
     return NULL;
}

void* malloc(unsigned int size)
{
     #ifdef DEBUG
      fprintf(stderr, "calling malloc\n");
     #endif

     pthread_mutex_lock(&malloc_lock);
     memBlock* block = findBlock(size);

     #ifdef DEBUG
     fprintf(stderr, "done finding block\n");
     #endif

     if(block == NULL)
     {
          block = (memBlock*)sbrk(0);
          if ( sbrk( sizeof(memBlock) + size) == (void*) -1)
          {
               pthread_mutex_unlock(&malloc_lock);
```

```
                return NULL;
            }
            block->size = size;

            pthread_mutex_unlock(&malloc_lock);
            return (void*) (block+1);
        }

        pthread_mutex_unlock(&malloc_lock);
         return (void*) (block+1);
}

void free(void* data)
{
        pthread_mutex_lock(&malloc_lock);
        if(data)
        {

        //    printf("%s\n", "calling free");
        //       fflush(stdout);
            #ifdef DEBUG
            fprintf(stderr, "calling free\n");
            #endif

            //memBlock* block = ((memBlock*) data) + sizeof(memBlock);
            memBlock* block = (memBlock*) data-1;
            blockList* head = headBlockList;

            while (head && !(head->size == block->size))
            {
                head = head->nextList;
            }

            if(head)
            {
                block->nextMemBlock = head->headBlock;
                head->headBlock = block;
            }
        }
        #ifdef DEBUG
        fprintf(stderr, "end free\n");
        #endif
        pthread_mutex_unlock(&malloc_lock);
}

void* calloc(size_t num, size_t nsize)
{
        size_t size;
        if (!num || !nsize)
        {
            return NULL;
        }
        size = num * nsize;
        // check mul overflow
        if (nsize != size / num)
        {
            return NULL;
        }

        void* block = malloc(size);

        if(block)
        {
            return memset(block, 0, size);
        }
        return NULL;
}

void* realloc(void *block, size_t size)
{
        if (!block || !size)
        {
            return malloc(size);
        }
        memBlock* memblock = (memBlock*) block-1;

        if (memblock->size >= size)
        {
            return block;
        }

        void* ret = malloc(size);
        if (ret)
        {
            memcpy(ret, block, memblock->size);
            free(block);
        }
        return ret;
}

void _init(void)
{
        #ifdef DEBUG
        printf("%s\n", "Running our malloc library");
        fflush(stdout);
        #endif
         base = sbrk(0);
         headBlockList = NULL;
}
```

Code listing 1 : a basic and simple sequential memory allocator with best fit policy

4. Results

“Every new allocator has to be analyzed, tested and compared with others in order to show and measure the intended improvements [TLSF1]”

Using a specific memory allocator is done by modifying the environment variable *LD_PRELOAD* to include the desired memory allocator, which is In the form of a shared library, causing the symbols contained in this shared library to be loaded before any executable giving precedence to the symbols contained in this library [LMEM]. Therefore without the need to modify the code of the already compile programs, by just providing ***malloc()***, ***free()*** and other set of memory allocating functions in any memory allocator, all is needed is to compile and load that memory allocator before the one linked to the program.

As stated in [TLSF1], Many allocator spatial efficiency measuring ways exist, such as plots of heap size, maps of busy/free blocks, amount of times the allocator calls the brk system call. For most of these ways, Valgrind will be used to provide memory checks, heap analyzing and call graphs to visualize the performance and spatial efficiency of each memory allocator.

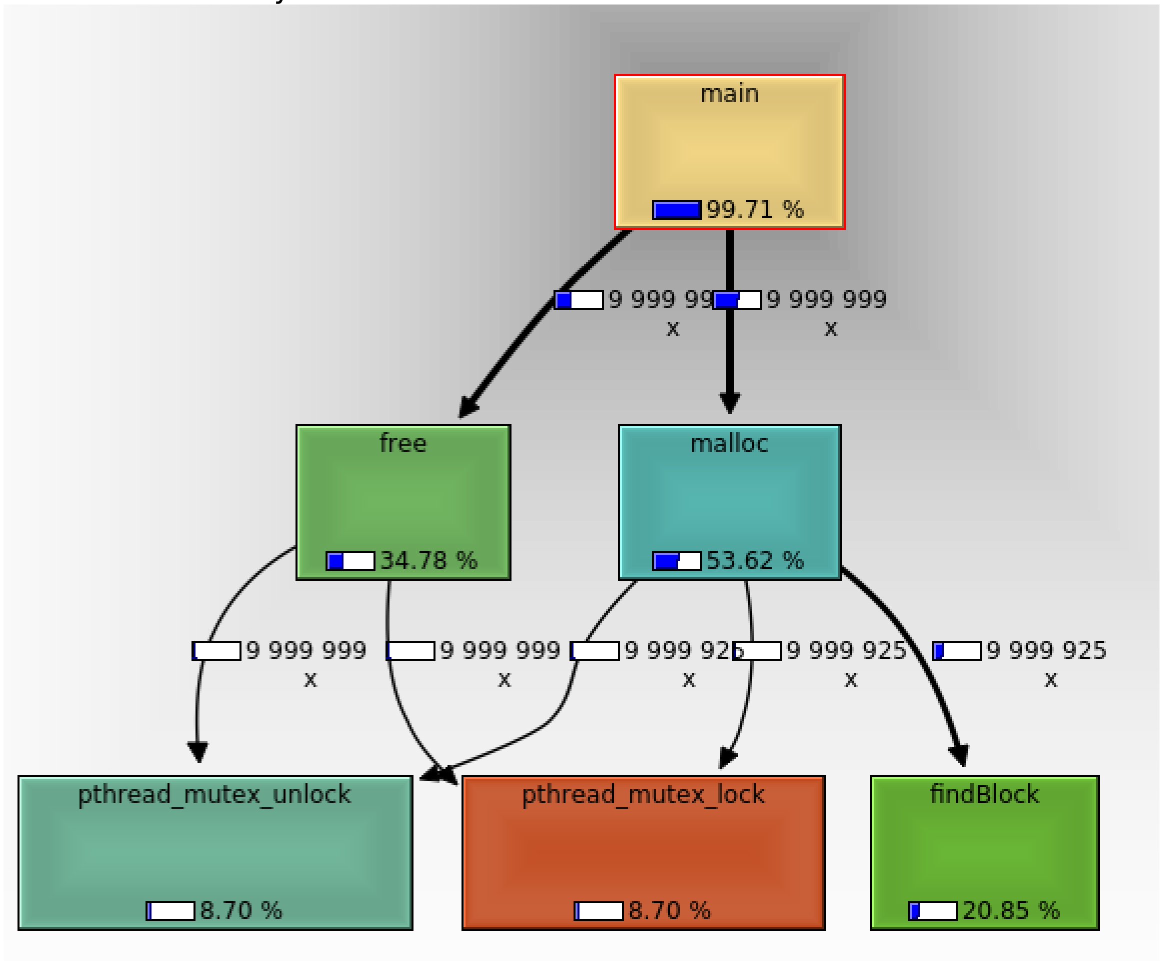
Following is a simple synthetic benchmark which allocates 100000 char on the heap and then frees the allocated chars, this is done on 100 times to take the average of CPU ticks of all the 100 iterations. CPU ticks or CPU time is calculated using the C++ std’s function std::clock() which is not related to the wall clock but to the CPU time [CPPClock]. This benchmark is used to produce a call graph in order to analyze the costs of calling each function of the memory allocator.

```
#include <iostream>
#include <ctime> // std::clock
#include <stdlib.h>
#include <string.h>

int* chars[100000];

int main()
{
        std::clock_t allocduration = 0;
        std::clock_t deallocduration = 0;
        std::clock_t start;
        int iterations = 100;
        for (int k =0; k <iterations; k++)
        {
            start = std::clock();
            for (int i = 0; i < 100000; ++i)
            {
                chars[i] = (int*) malloc(1);
            };
            allocduration += std::clock() - start;
```

```
                start = std::clock();
                for (int i = 0; i < 100000; ++i)
                {
                        free(chars[i]);
                };
                deallocduration += std::clock() - start;
        }
        std::cout << "heap allocation took " << allocduration/iterations << " clock ticks\n";
        std::cout << "heap deallocation took " << deallocduration/iterations << " clock ticks\n\n";
}
```

Code listing 2 : a simple synthetic benchmark

In the following call graph, it is clear that the function ***findBlock()*** takes a great amount of instructions just to search for a free block of a given size class, while ***pthread_mutex_lock()*** and ***pthread_mutex_unlock()*** contribute to the instructions count of the total calls with 17.40% , this graph call only covers ***malloc()*** and ***free()*** as the synthetic benchmark only uses those two functions.



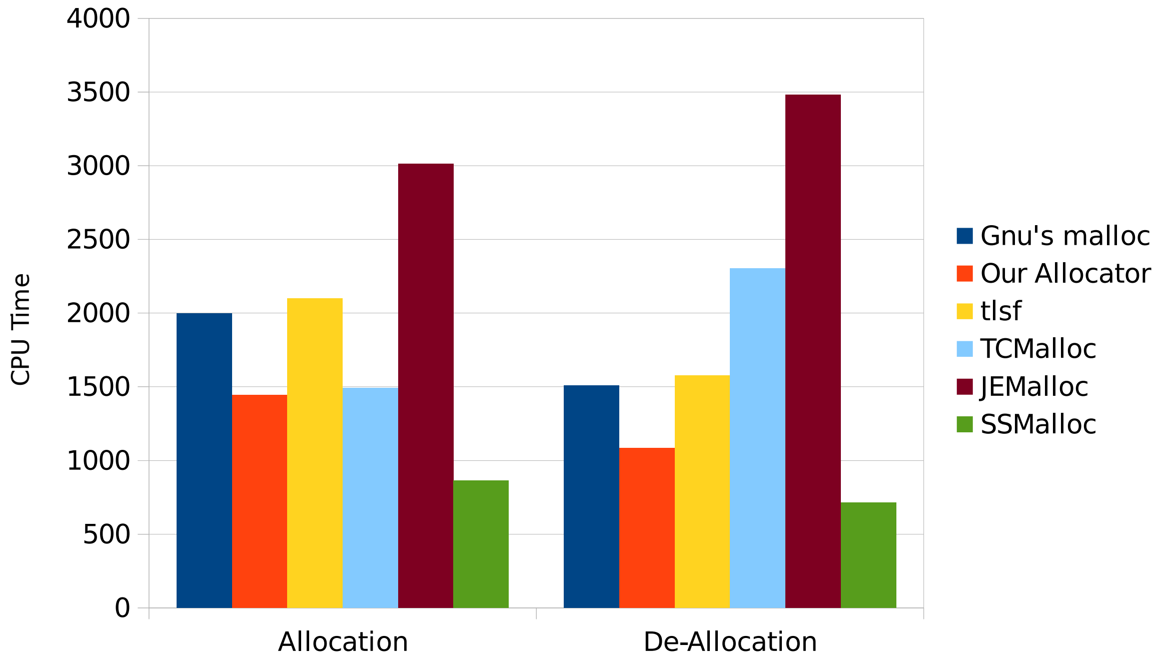
Graph 1 : a call graph from the benchmark listed in code listing 1

4.1 Performance Comparison

A performance comparison of the most trendy and used memory allocators in the competing arena, such as Google’s TCMalloc, which uses thread local caches of free objects allowing memory allocation requests of a certain size range to be satisfied without the needs of locks. By performing most allocations and deallocations without synchronization overhead, it is becomes optimized for large-scale multi-threaded applications [TCMalloc], JEMalloc, TLSF, which is a real time fast memory allocator with little fragmentation [TLSF1], GNU’s libc malloc, ssmalloc and finally our simple allocator.

The same benchmark shown in *graph 2* is used with the above mentioned memory allocators. The values of the Y axis represents the CPU time spent during the benchmarking, in order to convert it to seconds, it should be divided by ***CLOCKS_PER_SEC*** on the used machine [CLOCKMAN].

All of the benchmarks were done on a quad core `Intel(R) Core(TM) i5-3337U CPU @ 1.80GHz machine with 8 GB RAM.`



graph 2 : a comparison chart of running the benchmark, numbers represent cpu ticks (cpu time)

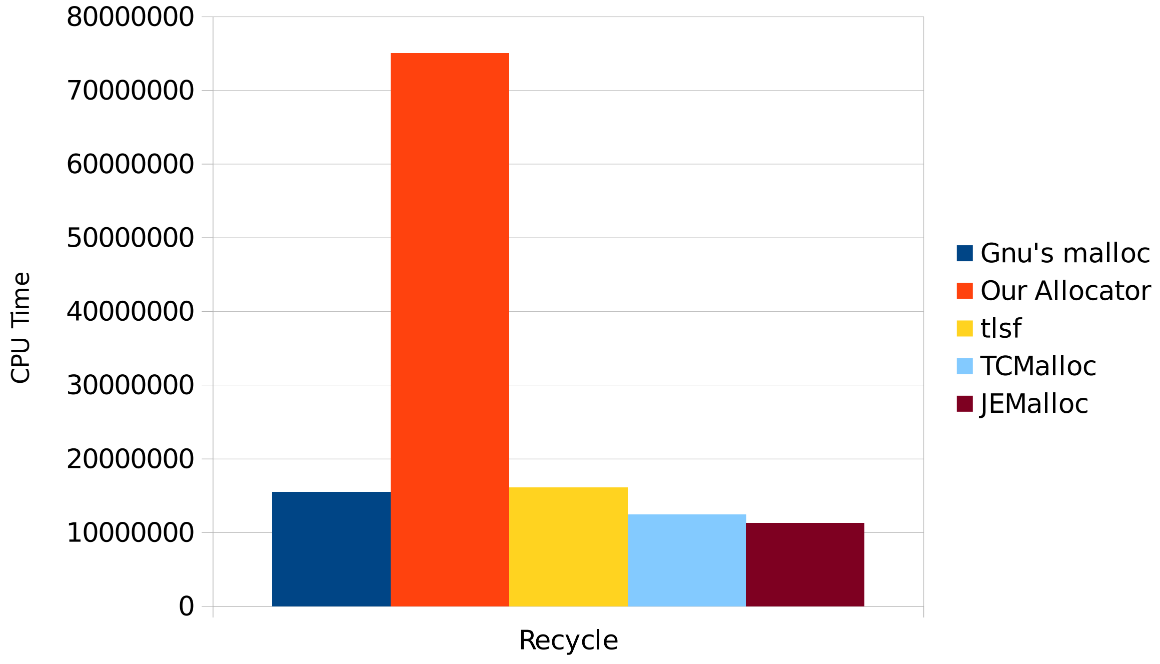
Another synthetic benchmark is the Recycle benchmark which is defined in [RECYCLE] as “Recycle: This is a custom synthetic microbenchmark that stresses the ability of multithreaded allocators to efficiently perform simultaneous memory management operations by multiple threads, for objects that are created and destroyed locally by each thread. Each thread allocates a total of 107 objects, the size of which is selected randomly from a given range. The benchmark can simulate different memory reuse patterns. Each thread deallocates all the objects it has allocated after every rate allocations, rate being a user-provided parameter. Recycle is not expected to scale with more processors (in terms of execution time reduction), since its workload is also scaled with the number of threads." With a small modification added to measure the cpu ticks consumed , the following code listing is used :

```
/*
 * recycle.c
 *
 * Copyright (C) 2007  Scott Schneider, Christos Antonopoulos
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301  USA
 */

#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <ctime>
#include <iostream>

/* EDB DISABLED #include <numa.h> */

size_t min_size;
size_t max_size;
int iterations = (int)1e8;
int rate;

double random_number()
{
	static long int seed = 547845897;
	static long int m = 2147483647;		// m is the modulus, m = 2 ^ 31 - 1
	static long int a = 16807;		// a is the multiplier, a = 7 ^ 5
	static long int q = 127773;		// q is the floor of m / a
	static long int r = 2836;		// r is m mod a

	long int temp = a * (seed % q) - r * (seed / q);

	if (temp > 0) {
		seed = temp;
	}
	else {
		seed = temp + m;
	}

	return (double)seed / (double)m;
}

void* simulate_work(void* arg)
{
	unsigned long** reserve = (long unsigned int**)malloc(rate * sizeof(void*));
	int i;
	int j;
	double rand;
	size_t object_size;

	for (i = 0; i < iterations; ++i) {

		if (i % rate == 0 && i != 0) {
			for (j = 0; j < rate; ++j) {
				free(reserve[j]);
			}
		}

		rand = random_number();
		object_size = min_size + (rand * (max_size - min_size));
		reserve[i % rate] = (long unsigned int*)malloc(object_size);
	}

	free(reserve);

	return NULL;
}

void numa_start(void);

int main(int argc, char* argv[])
{
	pthread_t* threads;

	//numa_start();

	if (argc < 5) {
		printf("correct usage: recycle <num threads> <min alloc size> <max alloc size> <alloc rate>\n");
		exit(0);
	}

	int num_threads = atoi(argv[1]);
	min_size = atoi(argv[2]);
	max_size = atoi(argv[3]);
	rate = atoi(argv[4]);

	iterations /= num_threads;

	std::clock_t start = std::clock();

	threads = (pthread_t*)malloc(num_threads * sizeof(pthread_t));
```

```
    int i;
    for (i = 0; i < num_threads-1; ++i) {
        pthread_create(&threads[i], NULL, simulate_work, NULL);
    }

    simulate_work(NULL);

    for (i = 0; i < num_threads-1; ++i) {
        pthread_join(threads[i], NULL);
    }

    std::clock_t duration = std::clock() - start;
    std::cout << "recycle  took " << duration << " clock ticks\n";

    return 0;
}
```

code listing 3 : Reycle.c benchmark

SSMalloc is not included as it segfaults, although Recycle benchmark is used in [SSMalloc] to benchmark SSMalloc and compare to other memory allocators. The recycle bechmark is ran with 2 threads and 5 bytes maximum allocation.



graph 2 : the recycle benchmark,numbers represent cpu ticks (cpu time)

5. Further Improvements and Discussion (Status)

A bitmap will be used to improve the indexing of free lists, as ***freeBlock()*** is a bottleneck of the current implementation, another improvement is to add “Combined free blocks”, where different combinations of continuous memory blocks of different sizes are computed at the time of freeing a block and crawling to the next and the previous lists. if a virtual size is requested, the virtual free block is made to satisfy the request. Free memory should be released back to the system to avoid heap exhaustion once a limit which is proportional to the total heap size available in the system is hit and should be dynamically updated. Adding multi-threading capability through a lock-less or lock method because the current mutex locks are not practical.

6. References

[DL2000] A memory allocator, Doug Lea, http://gee.cs.oswego.edu/dl/html/malloc.html

[LMEM] Inside memory management : The choices, tradeoffs, and implementations of dynamic allocation, Jonathan Bartlett, http://www.ibm.com/developerworks/library/l-memory/

[FFMEM] https://pavlovdotnet.wordpress.com/2008/03/11/firefox-3-memory-usage/

[TLSF1] Masmano, M., Ripoll, I., Balbastre, P. et al. A constant-time dynamic storage allocator for real-time systems, Real-Time Syst (2008) 40: 149. doi:10.1007/s11241-008-9052-7

[SSMalloc] Ran Liu and Haibo Chen. 2012. SSMalloc: a low-latency, locality-conscious memory allocator with stable performance scalability. In *Proceedings of the Asia-Pacific Workshop on Systems* (APSYS '12). ACM, New York, NY, USA, , Article 15 , 6 pages. DOI=http://dx.doi.org/10.1145/2349896.2349911

[OTCMalloc] Sangho Lee, Teresa Johnson, and Easwaran Raman. 2014. Feedback directed optimization of TCMalloc. In Proceedings of the workshop on Memory Systems Performance and Correctness (MSPC '14). ACM, New York, NY, USA, , Article 3 , 8 pages. DOI=10.1145/2618128.2618131 http://doi.acm.org/10.1145/2618128.2618131.

[TCMalloc] S. Ghemawat and P. Menage. Tcmalloc : Thread-caching mal-loc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html

[CPPClock] N. Josuttis, "5.7. Clocks and Timers | The C++ Standard Library: Utilities", *InformIT*, 2016. [Online]. Available: http://www.informit.com/articles/article.aspx?p=1881386&seqNum=2. [Accessed: 04- Nov- 2016]

[RECYCLE] Scott Schneider, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. 2006. Scalable locality-conscious multithreaded memory allocation. In *Proceedings of the 5th international symposium on Memory management* (ISMM '06). ACM, New York, NY, USA, 84-94. DOI=http://dx.doi.org/10.1145/1133956.1133968

[CLOCKMAN] "Clock(3) - Linux Man Page". N.p., 2016. Web. 4 Nov. 2016.