ELECTRICAL & COMPUTER
ENGINEERING DEPARTMENT

COURSE CODE: ECE P124

Technical Report

Deadline: 30/06/2021

MEMBERS

Name:	*Anastasios-Ioulios*	*Achilleas*
Last name:	*Vagenas*	*Michalopoulos*
Email:	*avagenas@uth.gr*	*acmichalop@uth.gr*
Enrollment number:	*2496*	*2575*

Summary

The goal of this project is the implementation of a multithreaded memory allocator. The main characteristic of this allocator is the absence of locks (atomic compare & swaps instead), while preserving locality of reference and performance.

Key Words

Object Classes: The allocated objects are split into small and large. In case of small allocations, they use bins of different sizes. Large allocations are serviced directly by the back-end. Whenever the term is used, we always refer to small allocations, which is any allocation request below 2048 bytes.

Object Headers: For each allocated object, we use headers to identify the cleanup operation during free(). For object classes, we use a 1-byte header. For large allocations, a 16-byte header is used, with the last byte having the same format as the object class header.

CAS operation: A compare and swap atomic operation. Compared to locks, we are limited in the use of 32-bit and 64-bit operations only.

Pageblock: A pageblock is the main data structure that services allocations for the object classes (or bins). It is a multiple of pages in size and it comes in different sizes depending on the object class it services. Pageblocks of the same object class are kept in a doubly linked list, visible only to the owning thread.

Page Classes: Since pageblocks need to service multiple object classes, we also have different page classes, with the only difference between them, the number of pages they consist of. In our allocator for now, we use 3 different page classes with each class having a power of 2 number of pages.

Page Multiplier: A multiplier used to increase the size of the pageblocks, for each page class we have, compared to the default.

Requirements & Decisions

Base: The allocator has to conform to the system base requirements like other malloc()/free() implementations. Firstly, the minimum alignment requirement is 16-bytes, since the largest native type (long double) is 16-bytes. Secondly, no locks have to be used on the front-end part of the allocator, instead only CAS operations are allowed.

Performance: The allocator has to service common requested sizes, fast, with minimum memory overhead (in terms of allocator headers) and provide locality of reference. In order to address these, the following decisions were made:

- Headers

  Headers have to be made as small as possible at least for the object classes in order to have minimum memory overhead. For alignment reasons, headers can only be 1-byte at the minimum.

- Object classes

  Object classes have to cover a wide variety of allocation sizes, in order to avoid a large memory blowup. On the other hand, a large number of classes will negatively impact performance (decoding of object class, page faults, cache misses). For the reasons above, a balance must be kept in the maximum number of classes we can support (not too few - not too many).

- OS memory requests

  Memory requests through syscalls (e.g., brk, mmap) should be minimized, either with caching or large requests that manage the underlying memory with an appropriate policy.

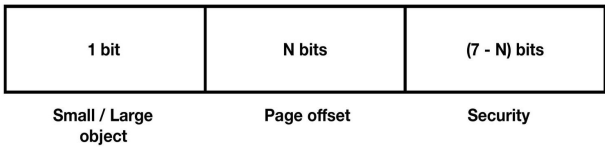
Flexibility & Upgradeability: While an obscure requirement, flexibility should be followed when designing the hierarchy, the structures and the functions of the allocator. Secondly, since the allocator is designed to be "flexible", it has to be also easily upgradable-parameterized. An example of this, is the page multiplier which adjusts the pageblock sizes.

ABA problem: In multithreaded computing, the ABA problem occurs during synchronization, when a location is read twice, has the same value for both reads, and "value is the same" is used to indicate "nothing has changed". However, another thread can execute between the two reads and change the value, do other work, then change the value back, thus fooling the first thread into thinking "nothing has changed" even though the second thread did work that violates that assumption. In our case, we might encounter this problem in our lock-free stack implementations.

Design Decisions

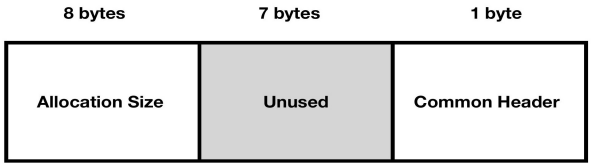
Front-End

Headers: Like we have said before we use headers to identify the object during freeing. In all allocations, we use a 1-byte common header with the following format:



- The MSB defines if the object type is large or small.
- The N bits define the distance of the object from the beginning of a page-block. The N can take values in the range of [2, 7]. This is only used if the object type is small.
- The rest bits define if there has been a corruption or a double invalid free.

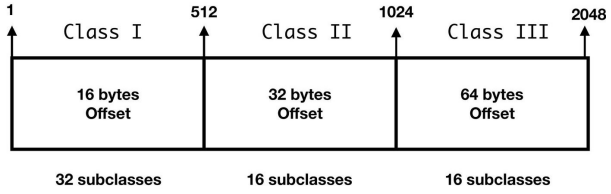
This header is all that we need for small type allocations (that belong to object classes), for large allocations an additional prefix header is also used:



- The **Common Header** is the header which was described previously.
- The **Allocation Size** is the size of a large type allocation.

In case of large type allocations, we use additional bytes for the size of objects, as the range of its values cannot be described in the 1-byte information of common header.

Classes: For the small type allocations, we separate them into object classes that service some sets of user memory requests. We have a total of 64 object classes, which are split in a 2-level hierarchy.



On the 1st level we have 3 major classes which have a number of sub-classes in them. Every sub-class that belongs in a major class has the same offset (difference from the next sub-class). The 2nd level then, is the point where the small allocation request is serviced. A sub-class services allocations that belong in a range of sizes (e.g., Sub-class2 $\Rightarrow$ [33,48] in bytes).

This design has some advantages:

- Alignment requirement for 16-bytes is automatically fulfilled, since all major class offsets are multiples of 16.

- Powers of 2 sized classes, which would cause large memory blowups, are avoided.
- The number of classes according to our intuition, seems adequate for a general purpose allocator and for small/"normal" allocations performs very well.
- The formula that generates the classes can be easily updated, for "medium" allocation requests (e.g., 0.5 pages - 8 pages).

But this design has some disadvantages, which we chose to live with:

- Extremely small user allocation requests (less than 16-bytes), there is significant space overhead (which could be up to 800%). In string-heavy generator programs, this might be a problem.
- Headers can cause a lot of extra space overhead for the first sub-classes.

Back-End

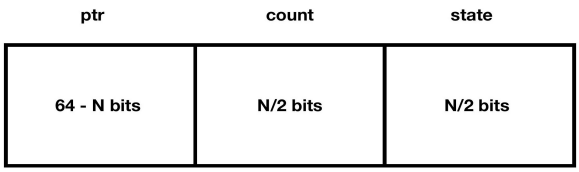
The back-end is the way we interface with the OS to request memory. In our case allocations/deallocations are made with the mmap()/munmap() system calls. In order to minimize the number of these expensive system calls we have 3 simple mechanisms:

- 2 levels of caching (Local/Global queues)
- Larger memory requests (Different page classes - Pageblocks)
- Memory sharing (Adoption policy - Global queues)

Local Free Queues: Each thread has a thread local set of queues that store completely free pageblocks of different classes. These do not require any synchronization, serve as the 1st level of caching and have a maximum limit of pageblocks they can hold.

Global Free Queues: These are set of queues that are visible to all the threads and hold completely free pageblocks. After a thread terminates, it stores every free pageblock from the local heaps and local free queues inside the global ones.

These queues, which are atomic counting stacks, require synchronization that is also ABA-safe. They serve as the 2nd and last level of caching (with a limit on the number of pageblocks they can hold), before we request memory directly from the OS.



- The **ptr** is the pointer to the pageblock.
- The **count** defines the number of pageblocks, which are stored in each queue, with respect on the limitation that we have defined.
- The **state** is used to address the ABA problem.

The 64-N bits are the effective bits of pointer. The rest N bits are parameterized. We decided to define the N equal to 24. Hence, we use 12 bits both for count and state.

Pageblocks' Adoption: When a thread terminates and still has allocated objects in some pageblocks, it sets these pageblocks as "orphaned". These "orphaned" pageblocks can then be adopted by other threads when they are freeing an object remotely.

However, the implementation of a pageblock's adoption might present some problems. There is the possibility of race conditions between threads during the remote freeing of an object, where a thread identifies that the pageblock is orphaned and tries to adopt it. In this way, we decided to create a structure which contains all the information needed for the remotely freed queue.

- The **Thread ID** defines the owner thread of the pageblock.
- The **Remotely freed offset** is queue head.
- The **Count** is the number of objects on the queue.

We want to avoid the race conditions between threads which make a free remotely and try to adopt an "orphaned" pageblock. So we define the number of bits for each field of the structure. Hence, we can achieve the synchronization needed between threads through CAS. Our only disadvantage is that the thread ID count is limited to 24-bit values and less, which means no more than 4 million threads have to be created during the process execution.

Experimental Evaluation

For the experimental evaluation, we used a combination of our own benchmarks, stress-ng and alloc-test. Our benchmarks are mostly micro-benchmarks that stress the allocator in very particular cases and check its integrity. Stress-ng and alloc-test are mostly for performance evaluation and comparison with other allocators.

Our own tests

- **Atomic-LIFO**: Tests the structure used for the global free queues where pageblocks are stored. It verifies correct operation and integrity of the main caching structure.
- **Class-integrity-malloc**: Stresses every object class and verifies alignment, headers and continuous allocation in the same class.
- **Class-integrity-realloc**: The same test as the above but instead of malloc, realloc is used.
- **Local-test**: This test generates threads and performs only local allocations/deallocations. Testes the front-end (pageblock local operations) and back-end (global caches).
- **Remote-test**: This test is a producer-consumer one, where a single thread allocates memory and the rest of the threads perform the deallocations. This stresses the synchronization of the remote free operation.
- **Shuffle-tests**: These 2 tests, are the integrity tests for randomized local only allocations/deallocations.
- **Adoption-test**: This test is the stressor for the adoption policy in the allocator. A producer thread, makes some allocations and is then terminated, with the allocations passed to the consumer threads. The consumer threads then compete with each other to steal the orphaned pageblocks. This test stresses the back-end heavily almost in its entirety.

Finally, we have also implemented our own stat collector function (**malloc_debug_stats()**), which collects statistics and information about the allocator, like:

- Kernel calls for memory
- Total allocations/deallocations/reallocations
- Total allocated/deallocated memory.
- Peak allocated memory.
- Pageblock steals (Adoption policy).
- The contents of a thread's local heap.

Something to note, using the above routine during the adoption-test, is the reduction of memory blowup. By implementing the adoption policy inside the allocator we saw a decrease in peak memory usage of about 50% in total (for the adoption-test).

Stress-ng

For this test, we used the following flags during execution of the stress-ng stressors:

- {- -malloc thread_num}: Run the malloc stressor for number of threads
- {- -malloc-ops ops_num}: How many allocations/reallocations/deallocations are to be made.
- {- -malloc-bytes bytes_num}: The maximum allocation size for each memory allocation/reallocation.
- {- -metrics}: Time measurements and performance measurements.
- {- -perf}: Collection of performance counters statistics through perf-stat.

In all cases, the number of malloc-ops is 10 millions. We use different number of threads (1-6) and different sizes of maximum allocations (for 1k, 5k, 10k and 20k). The main comparison for each test is the default malloc implementation. Finally, for this test we will use a page multiplier of 3.

		Time(s)	
Allocation Size	**Threads**	**Our allocator**	**glibc allocator**
1k	1	1.02	0.89
	2	0.56	0.47
	3	0.4	0.36
	4	0.33	0.29
	5	0.28	0.31
	6	0.26	0.41
5k	1	20.96	1.69
	2	4.32	0.74
	3	0.42	0.59
	4	0.33	0.39
	5	0.28	0.31
	6	0.26	0.27
10k	1	29.29	1.95
	2	11.81	0.9
	3	6.17	0.56
	4	2.68	0.43
	5	0.31	0.35
	6	0.28	0.32
20k	1	35.14	2.7
	2	16.6	1.14
	3	10.85	0.79
	4	7.53	0.57
	5	5.39	0.48
	6	4.06	0.47

We have made, from the results above, the following base observations:

- For a given allocation size, an increase on the number of threads leads to a decrease in time.
- For a given number of threads, an increase on the allocation size lead to an increase in time.
- Our allocator, for allocation sizes above 5k, time increases tremendously. This is due to the fact that the biggest object class is 2k in size, which means we have a lot of kernel calls.
- The glibc allocator, in general, performs better than our allocator.

Taking into account the base observations, our allocator performs really well in the cases we have designed it for.

For allocation sizes that fit inside object classes, our allocator while being slower initially, performs better than the glibc allocator for a larger number of threads. This would be even more evident for different pageblock sizes where modifications to the page multiplier were made. The page multiplier if left at its lowest (0), increases total time by 100% in some cases, while the performance benefits drop off starting at a multiplier of 3 and above.

For large allocations, since our allocator has no policy for large allocations sizes (in our case 2k and above), relying on kernel requests for memory is extremely expensive as it can be seen. This is even more evident in the case of 20k allocation size where our allocator has almost 15 times worse performance, on average, than glibc.

All these observations, were made clear by the performance counter stats (--perf flag). For example, for the 20k allocations, the statistics showed a large number of generated page faults for our allocator compared to the glibc one.

Alloc-test

For this test, the only modification we made was the maximum number of kernel threads used (our machine had 6 threads), inside the source code. We compared our results using the maximum number of threads against **ptmalloc** and **jemalloc** (**tcmalloc**, unfortunately, was not possible to install).

This test uses **new** and **delete** which also verifies that the operators are overloaded by the shared library ones (this was tested before, on our own test of course).

We run the test for all 3 allocators in our disposal, for a maximum object size of 16 bytes. Since the test supports individual object allocations of up to 31 bytes, we left it at the standard size (which was also the default). Some of the results are as follows:

Allocators	**Total time (ms)**	**RSS Max(Pages)**	**Memory overhead**
Ours	3194	131972	1.006866
Jemalloc	1927	132229	1.008827
Default (Ptmalloc)	2609	131938	1.006607

As, we can see from the results (we will not explain the procedure of the test just like stress-ng), our allocator while being competitive with the rest of the allocators, it still is far slower than the rest. The results used, are the largest metric out of the short summary metrics the app produced before termination.

Something to note though, is the memory overhead metric, which relates the memory allocated (by the operating system), against the requested memory by the app. In our case, our allocator is as efficient as the other allocators.

Finally, the performance difference of our allocator against ptmalloc is the same as in the case of the stress-ng test(for the 1k case, which is the fast path of our allocator).

Build

Our tests

For our testcases we have created a script, **run_test_alloc.sh**. The instructions which are executed:

```
make
./run_test_alloc.sh
```

With **make** the dynamic library is created (libxmalloc.so). With **./run_test_alloc.sh**, all testcases from test program are executed (inside the **/src** folder of the deliverable).

Alloc-test

In the deliverable file, inside the folder **/alloc-test**, we have the following:

- The modified script, **build_gcc.sh**, in order to link with our dynamic library object, where we have also left options for jemalloc/ptmalloc.
- A script (**./run_test.sh**) that modifies the environmental variables and executes the test.

To use the test just copy/overwrite the contents of **/alloc-test** folder inside the **/build** folder of the repository and execute these instructions:

```
./build_gcc
./run_test.sh
```

Finally, the shared library object should be left inside the **/build** folder where the executable of the test is generated (in our case when the overwrite happens we have a copy of the library too).

Stressng-test

In the deliverable file, inside the folder **"stressng-test"**, we have the following:

- A modified **Makefile**, in order to link with our dynamic library object (this is set by the **LOCAL_LIB** variable which can be easily set/reset).
- A script (**./run_test.sh**) that modifies the environmental variables and executes some of the tests during the experimental evaluation.

To use the test concerning memory allocations with our allocator, our **Makefile** should overwrite the default one provided, copy the dynamic library and the script in the same place. The instructions which are executed:

```
make
./run_test.sh
```