eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Degree in Computer Engineering

Computer architecture

End of degree work

Analysing the use of a SLAB allocator for user-space programs

Author

Iker Galardi Santiago

2022

eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Degree in Computer Engineering

Computer architecture

End of degree work

Analysing the use of a SLAB allocator for user-space programs

Author

Iker Galardi Santiago

Director(s)

Jose A. Pascual Saiz

Summary

Modern general purpose memory allocators have become fast and less prone to fragmentation, but still, there has been a movement into using memory allocators specifically tailored to a workload, as recently happened on operating system kernels like Linux or FreeBSD's. This new specifically tuned allocator is called the *SLAB allocator* and has shown significant speedups on the allocation/deallocation primitives of the memory subsystem.

Although this allocator has been designed for operating system kernels, given that it is specifically tailored to small temporary buffers (commonly used on hardware drivers), other workloads with the same requirements could benefit from it. Knowing that the allocator is particularly successful with small size allocations, a hybrid approach using the *SLAB allocator* and the traditional memory subsystem for larger allocations could bring considerable performance improvements to certain applications.

In this project, the aim is to create a from scratch implementation of the *SLAB allocator* for user-space and to test the potential performance improvements of the allocation/deallocation primitives using both synthetic and real workloads.

Contents

List of Figures

1. CHAPTER

Introduction

Memory allocations are the most common operations among computer programs, and thus, the speed in which a program allocates memory becomes a very important part, as it can heavily affect the overall performance of the application. Even though many solutions have been proposed over the years, this topic can be considered still under research, as operating systems and software alike are very sensitive to allocation time.

Traditionally, general purpose allocators such as the GNU allocator [gnu,] and *Jemalloc* [Evans, 2006] have been used in almost all scenarios, as they don't require any configuration and are not tailored to specific workloads. But in recent years there has been a move into using custom memory allocators that can take advantage of specific allocation patterns, sizes or system needs. There are several examples of custom allocators that help in specific use cases like *the SLAB allocator* [Bonwick et al., 1994] used on operating system kernels, *Hoard* [Berger et al., 2000] used for heavily multithreaded applications or *the smart allocator* [Ramakrishna et al., 2008] suited for embedded real-time systems.

All the prior mentioned allocators use what's called *the heap* in order to request and return memory. However, the current programming model divides the memory into the before mentioned *heap* (also known as *dynamic memory*), used for allocations not known at compile time; and the *stack*, that's used for compile time known allocation.

The *stack* operates like the *stack* data structure, by pushing and poping values following the *LIFO* (last in first out) principle. This simple set of operations makes it suitable for temporary and automatically managed memory, as these allocation times are very fast and the temporary values can be easily removed after the scope finished. But due to the limited

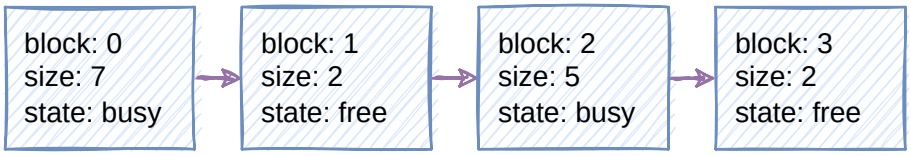

Figure 1.1: Example of a block list

operation set and size, the stack becomes unusable for objects that dynamically change sizes or objects with more complex lifetimes.

In contrast, the *heap* is a more complex section that programs use for bigger chunks of memory or dynamic allocations. This section does not have a specific structure, as each memory allocator has specific needs or approaches. Traditionally, the heap operates by two operations: `malloc(size_t size)`, which is used to requesting memory given a size (also called `new` in more modern languages); and `free(void* pointer)`, which can be used to return a previously allocated buffer (also called `delete` in more modern languages). These routines use allocation algorithms (also known as *memory allocators*) to structure the memory, ask more memory to the operating system or return unused memory.

A memory allocator can be easily understood as a list of memory blocks that is used to keep track of what memory is free or allocated (as it can be seen in Figure 1.1). Every time a program requests memory, the allocator will traverse the list looking for a block of memory with enough space.

The previously mentioned algorithm can be used as a simple memory allocator, but as can be expected, has certain issues that make it unsuitable for most use cases. One of the issues is what's called memory fragmentation. Taking the state of the block list of Figure 1.1, if a program requests 4 bytes, even though in total 4 bytes of memory are available to the requester, as they are not contiguous, they cannot be assigned, leading to waste of memory.

Furthermore, the allocation speed is a very important factor, because a frequent operation as memory allocation being slow can lead to very questionable application performance. In order to reduce allocation times, several allocators have used more complex data structures like trees or bitmaps, which by its nature have much lower search time complexity. Others, in contrast, have taken the approach of having a better list management in order to reduce traversal like the *SLAB allocator* [Bonwick et al., 1994].

Still, having a single algorithm for all possible situations can lead to inappropriate perfor-

mance and memory usage. That's why in recent years, there has been a movement into custom memory allocators that take advantage of their specific needs in order to squeeze more performance and try to reduce memory fragmentation. One important example of a custom allocator in the operating systems' world has been the aforesaid *SLAB allocator*, which is in use on the most popular operating system kernels. As *Bonwick* explains in his article [Bonwick et al., 1994], this allocator was designed to improve the allocation time and reduce memory fragmentation of temporary buffers in drivers, taking advantage of common allocation sizes.

Taking this into account, the aim of the project is to create a scratch built implementation of the aforementioned *SLAB allocator* for user-space programs and test if the custom allocator can bring performance improvements.

2. CHAPTER

The aims of the project

The SLAB allocator is the allocator introduced in SunOS 5.4 by *Jeff Bonwick* to improve the performance of the memory allocation/deallocation subsystem and to reduce memory fragmentation. The allocator employs same sized slots in order to reduce memory fragmentation, and with the help of clever free list management allocation and free-ing of memory can be done in constant time. This makes the allocator very suitable for temporary buffers and, due to its nature, the allocator is very effective with small sized memory (1/8 of the size of a page, usually 4 KB).

The main goal of this project is to study the usage of the SLAB allocator on user-space applications by implementing the allocator and by creating custom `malloc` and `free` functions that use the SLAB allocator internally in order to hide the peculiar API. As the allocator shines with small sized allocations, the custom `malloc` and `free` functions will use the system's implementation on larger allocation sizes.

At first, as most implementations of the aforementioned allocator are built for operating system kernels, they are specifically bound to internal kernel virtual memory systems and can't easily be ported to user-space. Thus, a custom allocator needs to be built from scratch in order to use kernel system calls instead of the virtual memory subsystem to get and return pages from the application's address space. The implementation will differ in some ways with the allocator described by *Bonwick* [Bonwick et al., 1994] because certain aspects of the original allocator cannot be implemented in user-space.

Although user-space programs could directly use the API given by the SLAB allocator, which could already be an improvement, most applications do not want to use custom

APIs for allocating memory, so a hybrid approach where small allocations go through the SLAB allocator and the rest go through system's allocator will be investigated. This hybrid approach will use the traditional `malloc` and `free` interface and implement a list of slabs that the implementation can use in order to speed up allocation times.

In order to see the improvements (or lack there of), a set of benchmarks should be created in the interest of properly comparing both allocators and analyze possible bottlenecks that could arise in the design and implementation of the custom hybrid allocator.

All this being said, these are the specific objectives of the project:

- Build a modified user-space version of the SLAB allocator.
- Test the performance of the SLAB allocator.
- Integrate the SLAB allocator as a fast path over `malloc`.
- Test the performance and possible bottlenecks of the integration

3. CHAPTER

Design and implementation

This chapter describes how the slab allocator was implemented in user-space and how the mechanism to automatically use slabs was implemented in a low overhead way. As the slab allocator's API is not how current C programs interact with the heap, a small bridge had to be constructed to utilize slabs transparently. The source code of the implementation and the benchmarks (see Chapter 4) can be found in GitHub[1]

3.1 Original slab allocator

The slab, as described by *Bonwick*, is an object cache used to allocate and construct objects in a fast manner. Each slab has a *front end* and a *back end*, each for a specific need: the *front end* interface is used by the user to allocate and free memory; and the *back end* interface is used by the slab in order to grow or shrink its size based on kernel events (such as memory pressure) or capacity (in order to grow it's size when it's getting full). Both of those interfaces can be seen in Figure 3.1.

Internally, the slab maintains a reference count and a list of free buffers. Each buffer is handled by a structure called `bufctl` which maintains the linkage and buffer pointer (as it can be seen in Figure 3.2). Still, this logical layout can not be directly transferred into small sized buffers, as it's `bufctl` would occupy as much as the buffers by themselves. Thus, for small size slabs, the buffer itself serves as the linkage by allocating an extra word.

[1] https://github.com/ikergalardi/userspace-slab-allocator

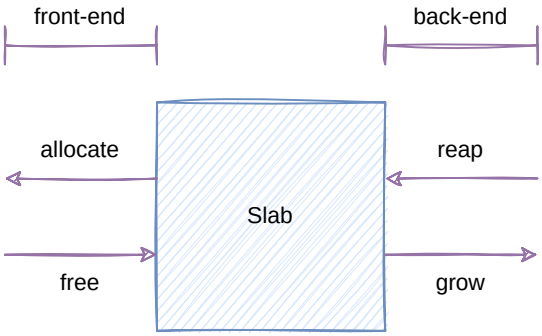

Figure 3.1: Interfaces of the original slab allocator

3.2 User-space slab allocator

The aforementioned architecture works well in the kernel, as the allocator can be tightly integrated with the kernel and receive events and statistics about memory pressure. But as user-space lacks those statistics and tight integration, the backend becomes useless. Thus, for the user-space allocator, the decision to completely remove the backend was done as it would greatly simplify the implementation. The mechanisms for growing and shrinking are still relevant in user-space, so that responsibility has been changed to another structure called *slab pool*.

The implementation is laid out in memory as shown in Figure 3.3 occupying the whole page for allocations. The header stores basic information such allocation size on that slab, the pointer to the start of the `bufctl` structure, a pointer to the start of the allocable buffers and a magic number in order to identify if a page is used as a slab or by other things. The `bufctl` structure is used as a linked list of buffers and to store its status.

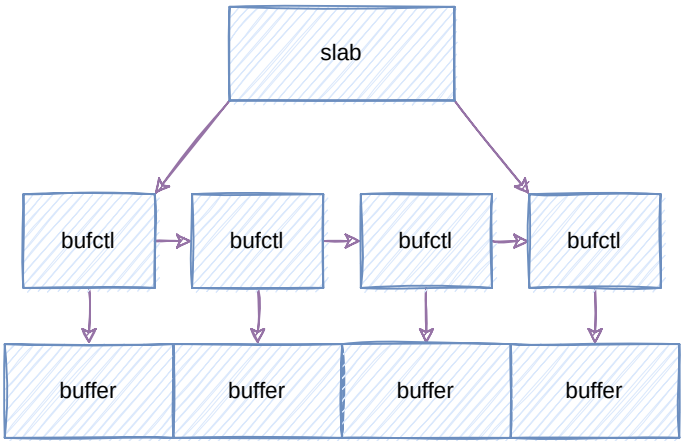

Figure 3.2: Logical view of the original slab

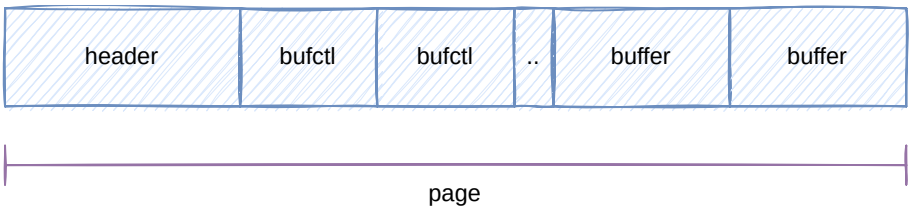

Figure 3.3: Structure of slab allocator's page

Even though physically the `bufctl` structure is laid out as an array, internally it works like a linked list by having the previous and next `bufctl`'s index (apart from buffer's state, busy or free). This architecture leads to a simple optimization to achieve constant time allocations while maintaining a small sized structure. By knowing the index on the physical array of the structure, the buffer it points to can be easily computed, while the list can be reordered in order to have free buffers first and make allocations in constant time. This strategy is what the original slab allocator uses in order to reduce allocation times, but the user-space implementation differs by grouping all the `bufctls` instead of having each `bufctl` next to it's buffer.

As mentioned before, in order to achieve constant time allocations, the `bufctl` list order is modified for every allocation or deallocation to maintain free buffers first. This is done by moving allocated buffers to the end and moving by moving freed buffers to the start of the list. This way, if the first `bufctl` is busy, means that the whole slab is full and another needs to be created. Pseudocode shown on Listing 3.1 explains the way allocations and deallocations are performed.

```
1  void* allocate(slab) {
2    slab.bufctl[0].isfree = false;
3    ptr = ptr_of_bufctl(slab.bufctl[0])
4    move_bufctl_to_end(slab.bufctl[0])
5    return ptr;
6  }
7  void deallocate(slab, ptr) {
8    index = get_index_of_bufctl(ptr);
9    slab.bufctl[index].isfree = true;
10   move_bufctl_to_start(slab.bufctl[index]);
11 }
```

Listing 3.1: Allocation/deallocation algorithms of the *slab*

With that reordering, we can assume that the list will be on a state similar to what is shown

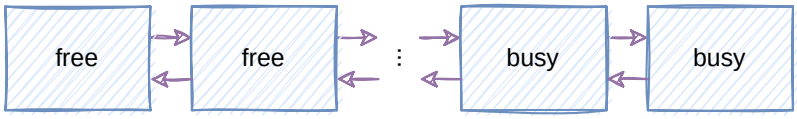

Figure 3.4: Example list state for a slab

on Figure 3.4. Knowing that, we can assume that if the first slab is busy, the whole slab is full. Thus, the algorithm can allocate inside the slab in constant time.

As the implementation has lost the capabilities of growing and shrinking, that responsibility is passed to a new structure called *slab pool*, that will manage lists of slabs and will grow or shrink it as it needs.

3.3 The slab pool

As mentioned on the section before, the slab itself has lost the capabilities of growing and shrinking in order to simplify the implementation. So, in order to add the same functionality, the structure called *slab pool* has been created.

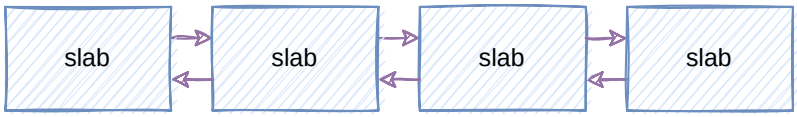

Figure 3.5: Logical view of a slab pool

The *slab pool* is a doubly linked list of slabs (as it can be seen in the Figure 3.5) that is able to allocate and free slabs to remove unnecessary slabs or allocate more when needed. It provides the same API as the slab, but this time, whenever there is not enough space to allocate a new buffer, it automatically creates new slabs.

As with the slab itself, the *slab pool* tries to maintain favorable ordering in order to reduce the time searching for slabs with available space. A similar strategy has been taken, but taking into account that a slab can have three states (compared to the two states of a buffer, busy or full): empty, partial or full. In order to have a situation similar to what's shown on Figure 3.6, partial slabs are considered empty as they can still allocate at least one buffer. Actions taken for each allocation/deallocation are shown on the Listing 3.2.

At first, checking for both the first and the second slab for space seems strange; but it's important to note that when several slabs are appended at once (as explained later), the first slab can become full while the rest are not touched.

```
void* allocate(pool) {
  if(pool.slab[0].hasspace)
    return allocate(pool.slabs[0])

  move_slab_to_end(pool.slabs[0])
  if(pool.slab[0].hasspace)
    return allocate(pool.slabs[0])

  append_new_slabs_at_start()
  return allocate(pool.slabs[0])
}
void deallocate(pool, ptr) {
  index = get_index_of_slab(ptr);
  deallocate(pool.slabs[index]);
  move_slab_to_start(pool.slabs[index]);
}
```

Listing 3.2: Allocation and deallocation algorithm for *slab pool*

It is worth mentioning the cost of a system call on program runtimes, specially when the operating system kernel has to do heavy work or multiple small work. This was a problem for the slab pool, as allocating pages one by one made so that when applications did allocations heavily, the *slab pool* would spend most of it's time mapping pages into the virtual address space. Other small problem was when the application had many temporary buffers, as this would lead to a sequence of unnecessary mapping/unmapping that could be avoided. To solve this, the slab pool maps several pages in a single `mmap` system call and with the help of a simple heuristic tries not to unmap parts of the virtual address space unnecessarily.

The aforementioned heuristic keeps track of how many slabs have still enough space to allocate on them, and with that knowledge tries to keep enough slabs in order to reduce unnecessary slab creation and destruction. When the heuristic considers that there are enough empty slabs, the next slabs that get empty are automatically destroyed. During testing, there were many scenarios where the allocation pattern lead to destroy a slab and immediately create another one, by keeping a buffer of slabs available that kind of behaviour is mostly mitigated.

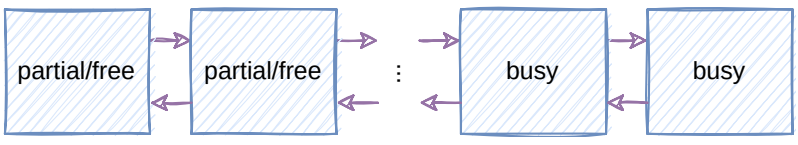

Figure 3.6: Example list state for a slab

3.4 Putting it all together

Now that the main structures are explained, we can put it all together in order to provide the typical `malloc` / `free` interfaces. For this, several slab pools are needed for each size. As previously mentioned, the idea behind this bridge is to provide a fast path over the system memory allocator, so having the correct configuration of slab pools becomes important for the specific use case.

At the start of the program, the allocator creates several slab pools of different sizes, and whenever the `smalloc` function is called, the allocator searches in the slab pools in order to find a pool with enough size. In the case that no suitable pool is found, the allocator uses the system `malloc` function as a last resort.

For the `sfree` function, the allocator searches for a magic number at the start of the page, in order to see if the allocated pointer was provided by the system allocator. If that's the case, the allocator simply calls system's `free` and returns. If that's not the case, the allocator goes through all the slabs until one of them is able to deallocate the pointer.

Summarizing, both functions `smalloc` and `sfree` follow the traditional heap manipulation API, and thus, they can be easily integrated into any program.

4. CHAPTER

Experimental set-up

This chapter describes how the user-space slab allocator was benchmarked and compared to glibc's allocator by creating a synthetic benchmark that could be configurable for different size allocation patterns, and how real world applications were ported to use the slab allocator.

4.1 Synthetic benchmarks

The benchmarking process is going to be divided into two distinct parts: performance testing using synthetic benchmarks and performance testing using a real world application. Synthetic benchmarks are traditionally used to simulate workloads and test the performance in those simulated cases. In this case, the main use for the synthetic benchmarks is to test favorable and not so favorable cases in order to properly study the the potential benefits and drawbacks. Using real world world applications helps see if the tested performance benefits and drawbacks are reflected in real world workloads.

The first synthetic benchmark created was the *random size allocation benchmark* that makes allocations in mass, selecting the size with the help of the beta distribution. The beta distribution takes two parameters in order to give different forms. For example, if the parameters given are $\alpha = 2$ and $\beta = 20$, the form that the distribution takes can be seen in the Figure 4.1.

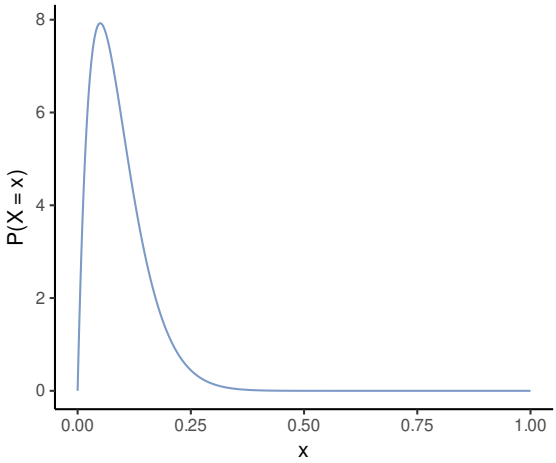

Figure 4.1: Beta distribution given the parameters $\alpha = 2$, $\beta = 20$

With the help of the GNU Scientific Library[1], we can choose at runtime the allocation size, given a maximum and a minimum. With this, we can model different allocation size scenarios in order to test the speed benefits when targeting small sizes (favorable to the slab allocator) and to test the penalties when targeting bigger allocation sizes (which won't even touch the slab allocator in most cases). Several parameters have been selected for size distribution and allocation lifetime. The selected values try to:

- Show the speedup when allocations always go through the slab allocator.
- Show a more "realistic" situation where some values go through system allocator and other through the slabs.
- Show the performance penalties when the slabs are not touched.
- Show the behavior of the allocator on different allocation patterns.

Knowing the aforementioned intentions, the selected parameters can be seen on Figure 4.2. With the first and the last distributions, we can test both extremes, when every allocation uses the slab allocator and no allocation uses the slab allocator. Meanwhile, the second distribution is a middle ground, a more "realistic" workload for the allocator.

Apart from allocation sizes, the order in which allocations and deallocations are made can take effect on performance. So another benchmark was created in order to address this. This second synthetic benchmark will create an array of operations that later will be

[1] https://www.gnu.org/software/gsl/

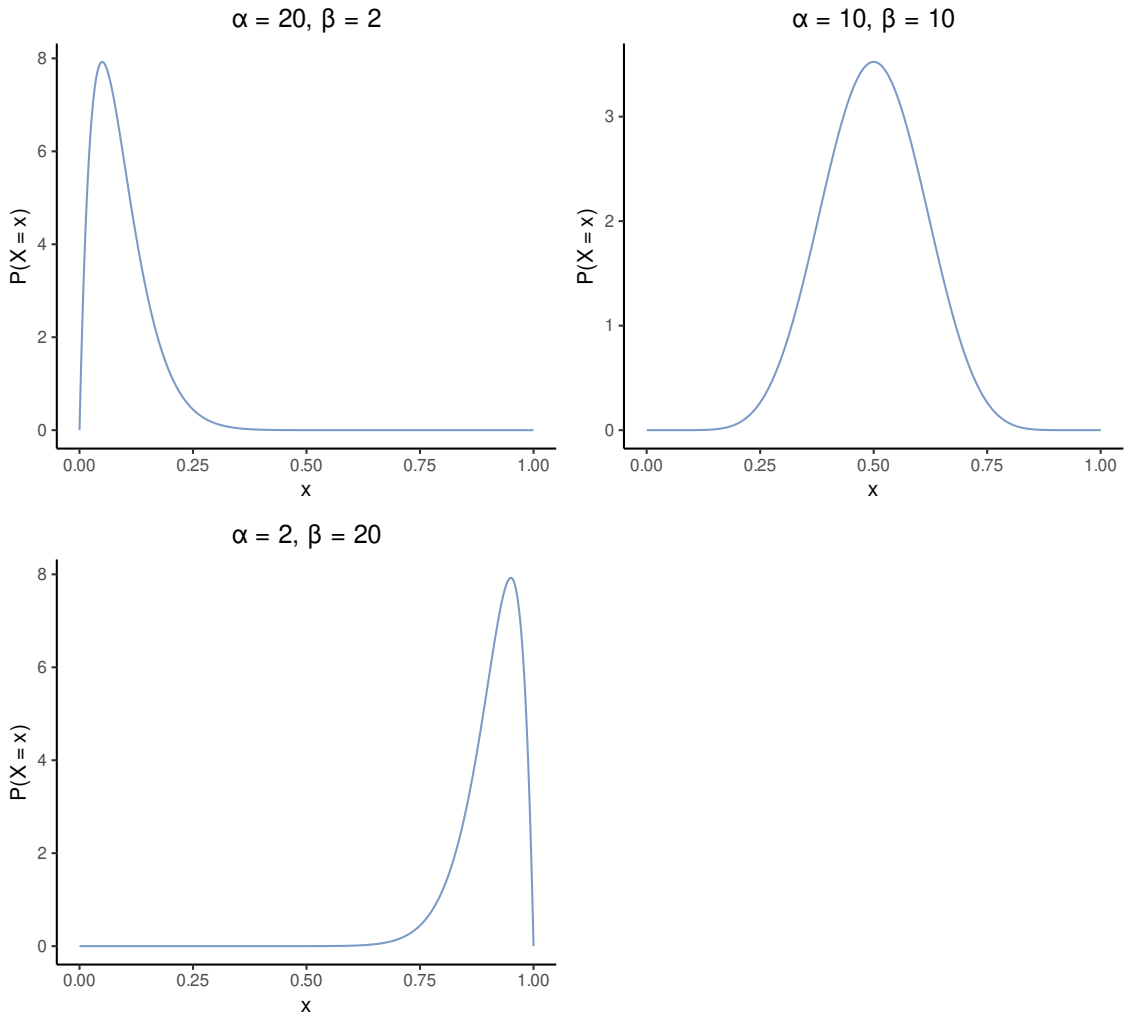

Figure 4.2: Distributions used for size selection.

performed. The parameter given to the benchmark is the average spacing of the `malloc` and its respective `free` is called *lifetime* (denoted by the letter d). When creating the operations array and a `malloc` operation is placed, it's respective `free` operation will be placed d operations later.

In the case of the lifetime synthetic benchmark, the distance values $20, 200, 2000, 20000$ have been selected according to some patterns observed on other applications like *cfrac* and *dash*.

4.2 Cfrac

It is important to always benchmark with real world applications, for this, the chosen real world application has a beneficial allocation pattern, as when those patterns do not occur the slabs aren't even touched, and those are easily tested synthetically (patterns that don't use slabs will simply use glibc's allocator, so the performance differences can be tested more easily synthetically).

The used application is Cfrac [Collins, 1999], which is a general purpose factorization algorithm used for Microsoft's mimalloc benchmark suite [2] in order to test the performance of an allocator for small short-lived allocations. As the pattern and sizes in which this algotithm uses are the best case scenario for the slab allocator in theory, this benchmark will be used in order to test real world performance benefits.

4.3 Benchmarking environment

In order to have a proper environment for benchmarking, a server using a *Intel(R) Xeon(R) CPU E5-1607 v3 @ 3.10GHz* CPU and *Ubuntu 22.04 LTS* was used. It is important to use a server in order to gather benchmarking information, as servers typically do not have as much operating system noise as desktops or laptops have (unpredictable background processes like automatic updates, changing power needs on battery, etc.).

It is important to note that changes in hardware should not affect much on the results. Of course, execution time may vary from processor to processor as certain processors are faster or slower, but metrics like acceleration factor should not be affected as no processor specific tweaks have been made.

[2] https://github.com/daanx/mimalloc-bench

5. CHAPTER

Analysis of the results

This chapter describes the obtained results after running the aforementioned benchmarks. The showed results are the average of 10 executions of the same benchmark and its standard deviation.

5.1 Allocation size synthetic benchmark

Figure 5.1 shows the results obtained after executing the benchmark on the allocation size synthetic benchmark. As theorized, when allocations use the slab allocator as a fast path, performance improvements can be seen. Meanwhile, as expected, when the slab allocator does not even get used as requested memory size slabs are not created, a small performance penalty can be observed. This benchmark shows the potential benefits the slab allocator gives over more traditional allocators like *glibc*'s one.

This benchmark can be used to see the potential benefit that this hybrid allocator has over *glibc*'s memory allocator. As this benchmark allocates and deallocates buffers at once, the benchmark can hide performance bottlenecks when handling all the slab orderings. Still, the results show promise of the performance improvements this allocator can provide to applications.

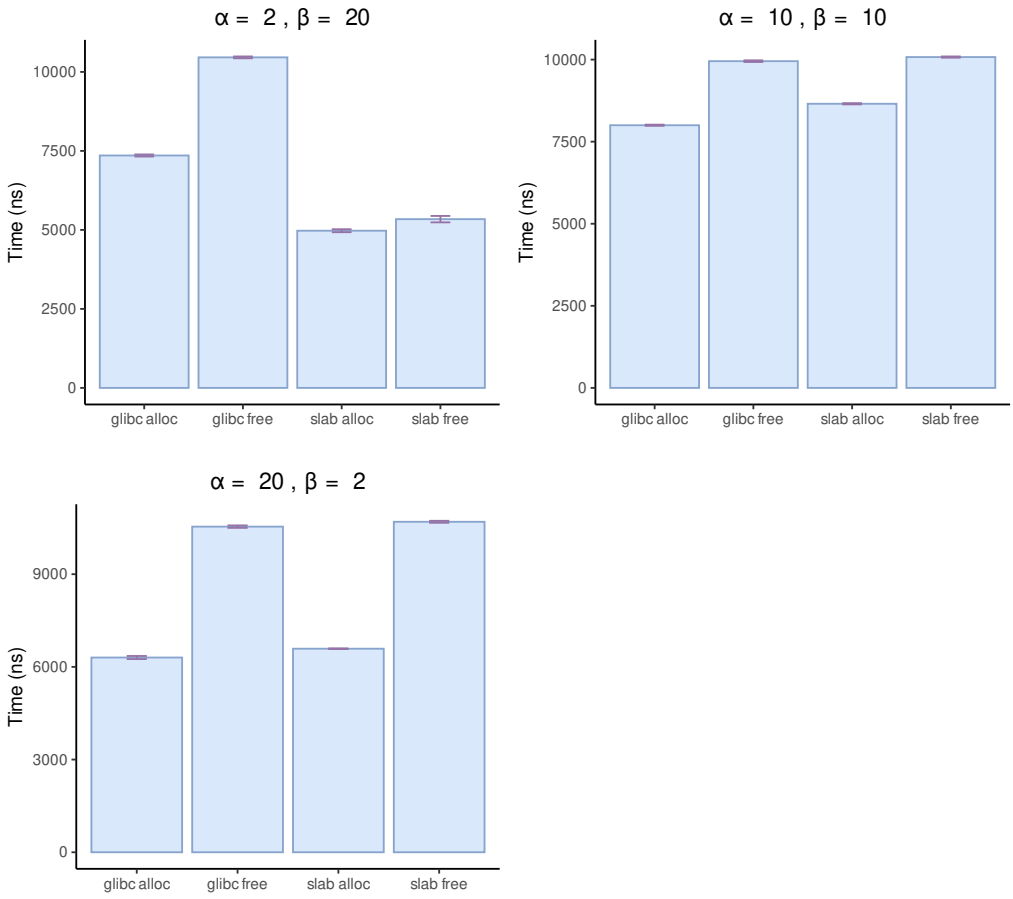

Figure 5.1: Size distribution synthetic benchmark results

5.2 Allocation lifetime synthetic benchmark

Figure 5.2 shows the results obtained after executing the benchmark on the allocation size synthetic benchmark. With these results the main problem of the tested implementation arises, as when complex allocation patterns arise the allocator struggles to keep up with the performance given by the *glibc* allocator.

This benchmark shows performance bottlenecks not seen on the benchmark before. Using the *perf*[1] tool we can inspect that the main problem was list management on the *slab pool*. The *slab pool*, as explained before, tries to maintain an optimal order within all the slabs to avoid traversing long lists of slabs, but continuous reordering of the slab list slows down the execution. To improve performance, maybe not reordering the cases when the optimal order is already there could be a good option.

[1] https://perf.wiki.kernel.org/index.php/Main_Page

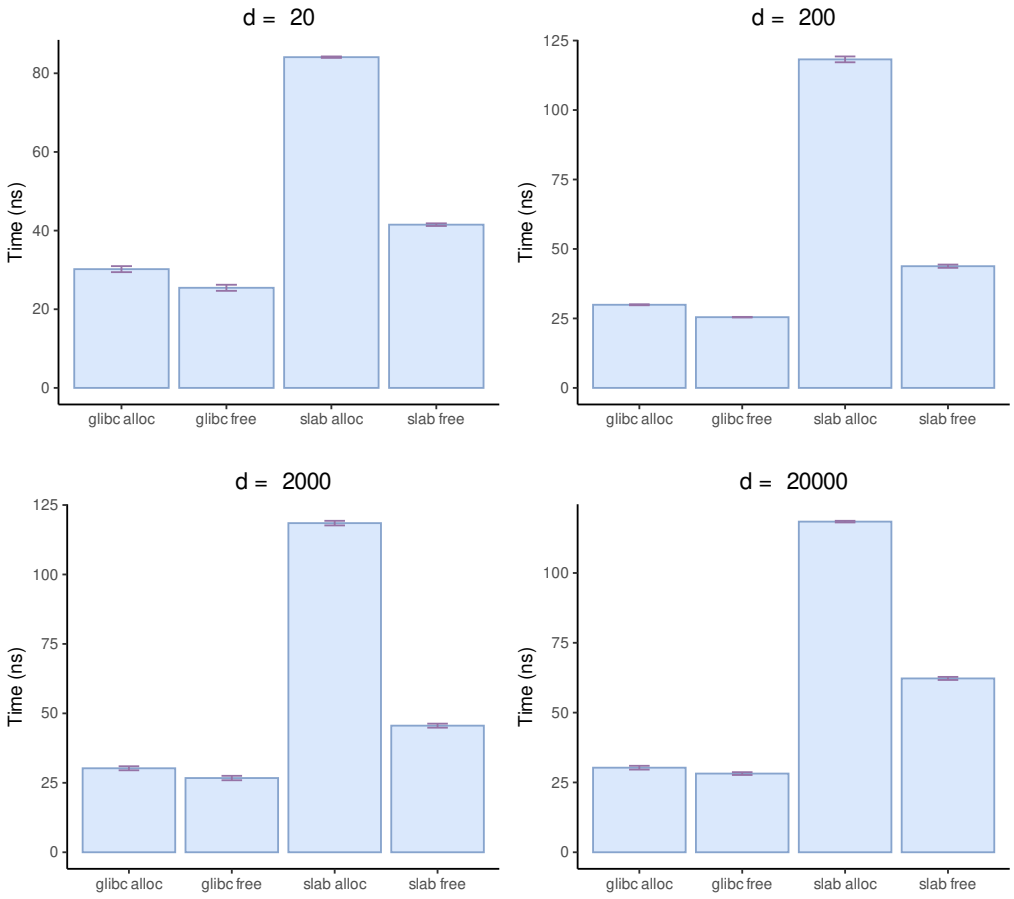

Figure 5.2: Allocation lifetime synthetic benchmark results

5.2.1 Cfrac algorithm benchmark

Figure 5.3 shows the results obtained after executing the benchmark on the *cfrac* algorithm. This benchmark does not show any preference of allocator, as the allocation time does not affect the overall performance of the algorithm. This benchmark is a good indication that the allocation time does not affect performance on many applications.

The lack of execution time differences was stunning at first, but after carefully analyzing the impact that allocation time had on the application it was clear. Most of the time the *Cfrac* application was simply doing its calculations and only 0.1% of the application time was the actual allocation/deallocation time. Thus, the performance difference between both allocators was insignificant.

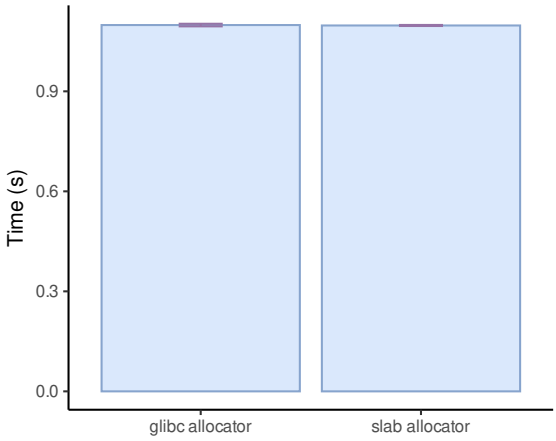

Figure 5.3: Cfrac benchmark results

6. CHAPTER

Conclusions

As seen on the results, the implemented slab allocator does not show the performance benefits theorized. This lack of performance comes from two distinct places: time and implementation details. Even though the slab itself is much faster than the standard allocator, when layers of slab management come into place on real world applications or random pattern benchmarks, the performance lacks as the concrete implementation is not mature enough. This could be solved by just having more development and testing time.

The size synthetic benchmark shows the promise that the algorithm has, as mass allocation followed by mass deallocation does not let the implementation have to reorder the slabs much and the performance benefits of the non-resizable slab shines through. Knowing this, we can determine that with enough development time, the hybrid approach studied on this project could help improve performance on certain workloads.

Even though the shown results aren't good, the fact that certain workloads could benefit from significant performance improvements (like the results shown on *random size benchmark*). The flexible nature of the bridge between the *SLAB allocator* and the traditional API shows that using a better configuration specifically tailored to an application or with better tuning of the heuristic could drastically improve allocation times on an application, hopefully improving the general performance. Still, as said before, the allocator itself needs some work in order to show the theorized performance gains.

7. CHAPTER

Future work

As this project's purpose was to test the potential benefits of using the slab allocator in user-space, the project can be considered *finished*. Still, to be able to use the available source code for production, several points need to be considered:

- Project testing has been limited. Although tests are available, there are not many neither good for production. So before even trying to improve performance or add features, a better testing framework should be considered.
- Multithreading support is missing, so multithreaded applications can cause race conditions on the allocator and give unpredictable results. Locks remove race conditions, but global or per slab pool locks should be avoided, as they could cause performance penalties.
- Implement debugging features for use-after-free-like bugs. The original SLAB allocator article mentions several debugging mechanisms implemented on SunOS, but a better integration with user-space would be also beneficial (aborting of application, showing stack trace, etc.).

Testing and multithreading aside, several SLAB specific features were discarded as they did not directly fit what it was tested. Still, undermentioned features can potentially improve performance and ease of use:

- Investigate slab coloring. Slab coloring is proposed on the SLAB allocator article in order to improve cache and bus utilization, and thus, performance.

- Improvements to the heuristic should be made, as the current implementation is quite simple. Maybe kernel specific statistics could be gathered periodically through system calls and try to remove as many unmappings as possible and return memory only when necessary.

Bibliography

[gnu,] The gnu allocator. https://www.gnu.org/software/libc/manual/html_node/The-GNU-Allocator.html. Accessed: 2022-04-02.

[Berger et al., 2000] Berger, E. D., McKinley, K. S., Blumofe, R. D., and Wilson, P. R. (2000). Hoard: A scalable memory allocator for multithreaded applications. *ACM Sigplan Notices*, 35(11):117–128.

[Bonwick et al., 1994] Bonwick, J. et al. (1994). The slab allocator: An object-caching kernel memory allocator. In *USENIX summer*, volume 16. Boston, MA, USA.

[Collins, 1999] Collins, D. C. (1999). Continued fractions. *The MIT Undergraduate Journal of Mathematics*, 1:11–20.

[Evans, 2006] Evans, J. (2006). A scalable concurrent malloc (3) implementation for freebsd. In *Proc. of the bsdcan conference, ottawa, canada*.

[Ramakrishna et al., 2008] Ramakrishna, M., Kim, J., Lee, W., and Chung, Y. (2008). Smart dynamic memory allocator for embedded systems. In *2008 23rd International Symposium on Computer and Information Sciences*, pages 1–6.