ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

AN OPERATING SYSTEM KERNEL FRAME ALLOCATOR IN RUST

SEMESTER PROJECT

Author: David Desboeufs
david.desboeufs@epfl.ch

Supervisors:
Edouard Bugnion
edouard.bugnion@epfl.ch

Charly Castes
charly.castes@epfl.ch

15th January 2023

Contents

1 Introduction

We aim to develop a simple frame allocator for x86-64 systems, also known as AMD64, which is the x86 architecture's 64-bit variant. We want to create a simple algorithm that can be proved with symbolic execution, such that it avoids path explosion. Moreover, unbounded loops are not allowed in the algorithm because they can lead to infinite execution.
The algorithm is written in Rust, which offers several advantages. Rust is designed to be memory safe (e.g. buffer overflows and data races) as it uses strong typing and a system of ownership and borrowing [1]. Furthermore, when compared to C code, it offers good performance [2].
Intel® x86-64 page tables come in three sizes: 4Kb, 2Mb (big page) and 1Gb (huge page) [3]. All of these sizes are used by our frame allocator.
Internal and external fragmentation should be minimal in a good allocator. Internal fragmentation is defined as allocating a larger block of memory than required [4]. External fragmentation is defined as the inability to allocate memory despite sufficient memory. This is due to several small blocks of free memory being distributed across the memory space [4].
Our algorithm is a modified version of the buddy allocator, with the particularity to divide memory blocks into 512 rather than two. We also provide deallocation safety features. Furthermore, we use Intel-specific instructions to improve the performance of our algorithm. Finally, we look at the external fragmentation when fine-tuning search parameters.

2 Algorithm Description

2.1 Context

Our goal is to allocate pages of 4Kb, 2Mb and 1Gb. We first consider the linked list allocator, with an additional block splitting mechanism to allocate 4Kb and 2Mb pages [5]. The main issue is that we cannot merge blocks afterward in a finite number of instructions. This is because free-linked lists have no notion of locality.
Our second option is the buddy allocator, which divides memory blocks into two recursively to accommodate requests. In our situation, this requires at least 18 levels in order to allocate the three Intel page sizes, 18 because 1Gb pages must be split 18 times to reach 4Kb (1Gb = 262144 * 4Kb and 2^{18} = 262144). The search is too expensive, we reduce these 18 levels to 3 with the idea to split memory blocks in 512 instead of 2.

2.2 Tree Representation

The buddy allocation technique inspired our frame allocator [6]. This consists of recursively splitting blocks in two to obtain the smallest possible block to satisfy a request.
Because blocks are divided into two parts, each one has a buddy. When a block is deallocated, it is merged with its buddy if this one is free (coalescing). This technique allows us to efficiently allocate and deallocate power of two size blocks of memory.

The buddy allocation technique can be illustrated with a full binary tree [7], with node values indicating whether or not a node is free (1 for free, 0 for used). Figure 1 shows the occupation tree, where two blocks are allocated: 8Kb and 4Kb, respectively (in red). Note that a parent block is free if its two children are also free (logical AND tree).

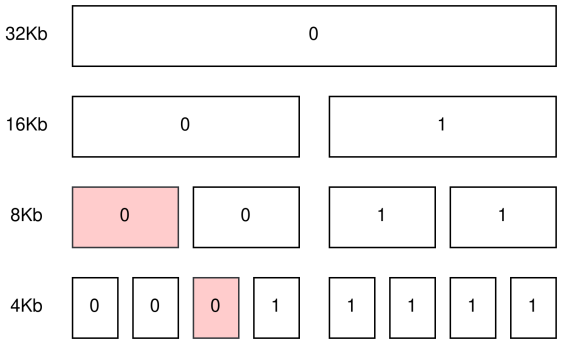

FIGURE 1
32Kb Buddy Allocator Representation

Top level does not provide any information about available 8Kb and 4Kb blocks, except when it is free. For example, if we want to allocate a 4Kb block, we cannot begin at the top of the tree, instead, we must go through each 4Kb block to find the first one that is free.

In order to know which child is free, each node has two bits instead of one, one bit per child. On the right, figure 2 depicts the new binary tree. Note that a node now corresponds to two blocks rather than a single block. When we see "01" in a node, we know that the right child is available. However, this new architecture does not completely solve the issue of locating a free 4Kb starting from the top level. The AND structure causes a problem because, if one child is used, we are left in the dark about the other.

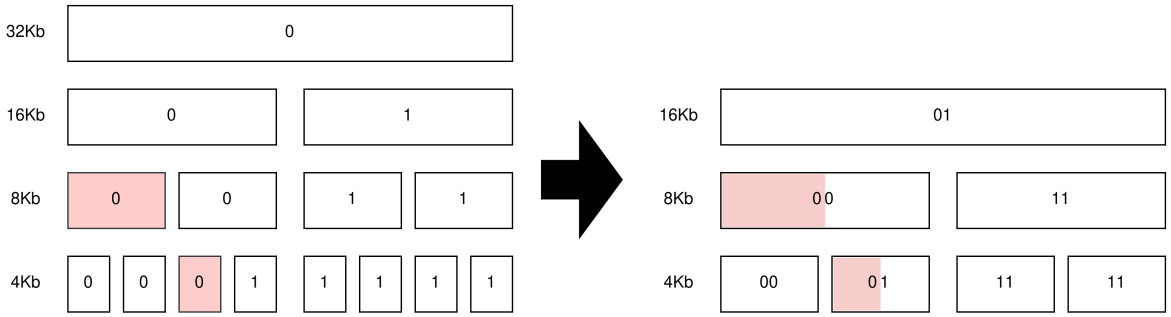

FIGURE 2
2 Bits per Node

2.3 DATA STRUCTURE

Our solution to the previously stated problem is to create an OR tree rather than an AND tree, which means that we have a 1 if at least one child is free. Figure 3 shows the new tree that can be used efficiently to find a free 4Kb block beginning at the top. To find a free 4Kb block, we must traverse three nodes, this corresponds to tree's height.

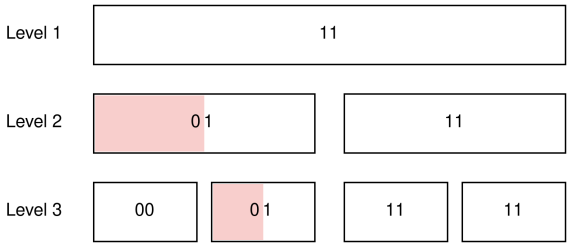

FIGURE 3
4Kb Tree

Because this tree can only be used to allocate 4Kb blocks, we must create a separate tree for each block size. Figure 4 depicts the two additional OR trees, they allow us to allocate 8Kb blocks (on the left) and 16Kb blocks (on the right). These three data structures make it easy to allocate the three block sizes. For each tree we begin at the level 1, and, if at least one of the two bits is 1, we have at least one free block of the desired size in memory. Note that because there is only 4Kb memory left on the left side, we see that both trees have level 1 equal to "01".

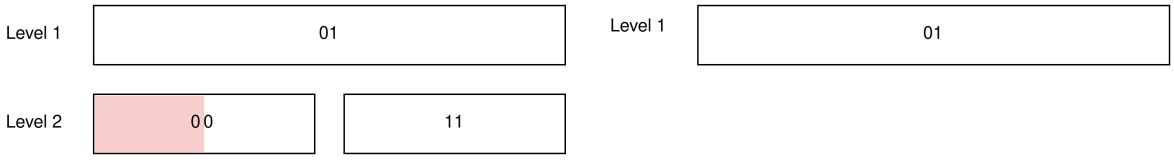

FIGURE 4
8Kb and 16Kb Trees

We can allocate blocks of 4Kb, 8Kb, and 16Kb with the previously created trees. When an allocation is completed, the three trees must be updated to reflect the changes. Observe that some changes are unnecessary, for example, when a 8Kb block is allocated, all levels of the 4Kb tree are updated. The level 3's update is irrelevant and can be ignored, when we set the parent bit (level 2) to 0, we will never visit the children. This reduces the number of updates we perform at each allocation. Figure 5 illustrates the new three data structures, which have the same configuration as before (8Kb and 4Kb allocated).

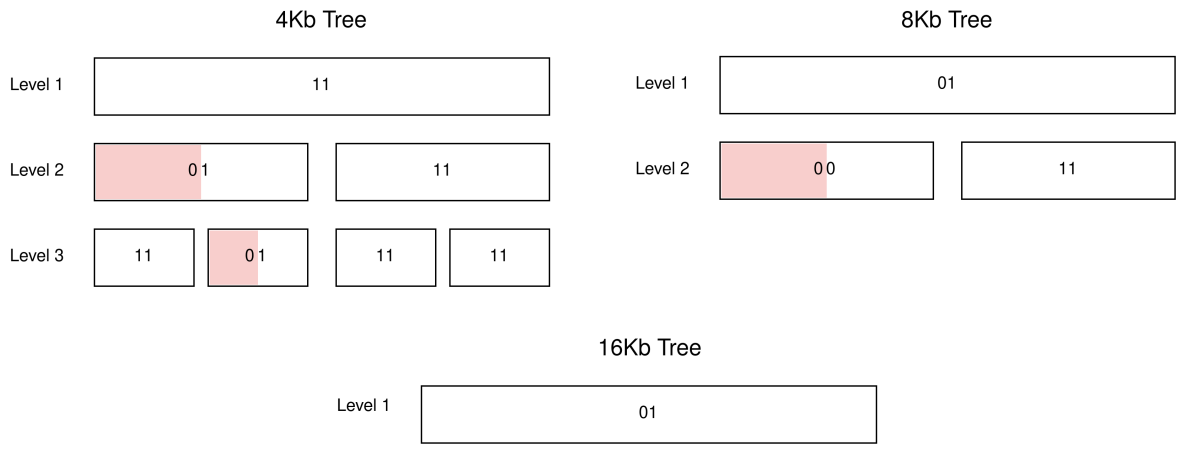

FIGURE 5
4Kb, 8Kb and 16Kb Trees

When we perform an allocation, we first search through the tree corresponding to the block size. If a block is available, we set the bit of the chosen block to 0 of the bottom level first. After that, we update the upper levels to maintain a consistent OR tree. To accomplish this, we check recursively if we need to change the upper bit; if the buddy is already used, then both are used, we set the parent bit to 0. We do this for all upper levels.
In the two other trees, we set to 0 the following bits: level 1 for 16Kb block allocation, level 2 for 8Kb and 4Kb block allocation (we ignore unnecessary updates). Updates are done recursively in the same manner as for the corresponding tree to the block size.
The same logic applies to block deallocation, except that we must set the upper blocks to 1 (free) in order to have a consistent OR tree.

2.4 AMD64 PAGE TABLE

We must address the allocation of 4Kb, 2Mb and 1Gb pages. With the previously described structure, we need a 4Kb tree of height 18 ($\log_2(512^2)$). However, there is a factor of 512 between 4Kb and 2Mb, and the same between 2Mb and 1Gb. Here comes the suggestion to have 512 children per node rather than the previously stated two. Each node contains 512 bits (one bit per child). The height of the 4Kb tree is now three. Figure 6 portrays the representation of a single node and its 512 children.

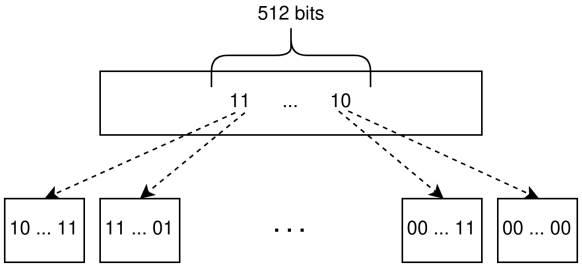

FIGURE 6
512-ary Node Representation

We eventually end up with three trees, one for each page size. We have a maximum capacity of 512Gb because each node contains 512 bits ($512^3 * 4Kb = 512Gb$).

2.5 Finding a Free Page

As we begin at the top level, we must decide which child to choose from the ones that are free. A simple technique is to always choose the rightmost one (search from right to left). To find the first free bit, we use two Intel-specific assembly instructions called *bit scan forward* and *bit scan reverse* rather than just performing a linear scan. We must divide the 512 bits into eight pieces because those instructions use 64-bit variables. When we spot a free bit while iterating over the eight pieces, we stop iterating. Section 4.3 illustrates the effect of changing the direction of searching on external fragmentation.
The option that minimizes external fragmentation is the following: search from right to left for 4Kb and 2Mb trees and search from left to right for the 1Gb tree. This comes from the fact that when a page of 1Gb is freed, a page of 4Kb or 2Mb may take its place, causing external fragmentation. This can be avoided if 1Gb pages are allocated beginning from the other side.

3 Rust Implementation

In this section, we will explore in detail the Intel-specific instructions for efficiently finding a free bit among the available 512 bits. We will take a look at how we store in memory the trees from the previous section. We will examine the functions required to construct our allocator. Finally, we will see that our algorithm provides safe deallocation through allocation size inference.

3.1 Background

Bit Scan Forward Instruction

We will look at how to efficiently find the first 1 in a sequence of bits using Intel-specific instructions. We have a sequence of 512 bits, which must be split into 8 parts of 64 bits each, as we use a 64-bit machine. We iterate through the eight parts, stopping the iteration when we find the first 1. Figure 7 shows the simplest method in Rust as well as its optimized assembly code on the right.

```rust
pub extern fn find_first_one(input: u64) -> u64 {
    let mut temp = input;
    for i in 0..64 {
        if temp & 1 == 1{
            return i;
        }
        temp >>= 1;
    }
    return 0;
}
```

```
example::find_first_one:
        xor     eax, eax
.LBB0_1:
        test    dil, 1
        jne     .LBB0_10
        test    dil, 2
        jne     .LBB0_9
        test    dil, 4
        jne     .LBB0_8
        test    dil, 8
        jne     .LBB0_7
        shr     rdi, 4
        add     rax, 4
        cmp     rax, 64
        jne     .LBB0_1
        xor     eax, eax
        ret
.LBB0_9:
        add     rax, 1
.LBB0_10:
        ret
.LBB0_8:
        add     rax, 2
        ret
.LBB0_7:
        add     rax, 3
        ret
```

Figure 7
Simple For Loop

We observe that the compiler uses a loop and rolling optimization technique, but we still have a linear search. In fact, the Intel instruction set includes instructions to determine the index of the first bit set to 1, *Bit Scan Forward* (BSF) and *Bit Scan Reverse* (BSR) [8]. Figure 8 depicts BSF's Rust code as well as its assembly code on the right.

```
pub extern fn bsf(input: u64) -> u64 {
    let mut pos: u64;
    unsafe {
        asm! {
            "bsf {pos}, {input}",
            input = in(reg) input,
            pos = out(reg) pos,
            options(nomem, nostack),
        };
    };
    return pos;
}
```

```
example::bsf:
        bsf     rax, rdi
        ret
```

FIGURE 8
Bit Scan Forward

Instead of directly using assembly code in Rust, we can use the *trailing_zeros* Rust method. This method returns the number of trailing zeros, which corresponds to the index of the first one. Figure 9 shows the *trailing_zeros* method code as well as its assembly code on the right. We observe that it is similar to the previous assembly code, with an additional check performed.

```
pub extern fn trailing_zeros(input: u64) -> u32 {
    input.trailing_zeros()
}
```

```
example::trailing_zeros:
        test    rdi, rdi
        je      .LBB1_1
        bsf     rax, rdi
        ret
.LBB1_1:
        mov     eax, 64
        ret
```

FIGURE 9
Trailing Zeros Method

TREE STORAGE

Trees are stored in simple arrays, figure 10 shows how they are flattened (binary tree as an example), where node content corresponds to its array index [9]. Given a node $index$, the left child corresponds to $2 * index + 1$ and the right child to $2 * index + 2$. This is similar for the 512-ary tree, except we multiply by 512 and add from 1 to 512 to iterate over all children.

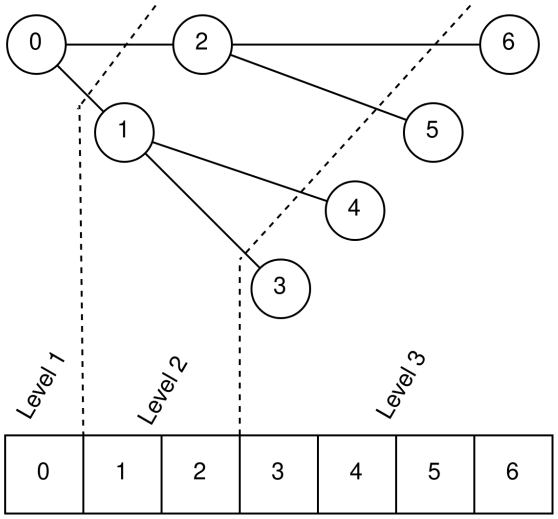

FIGURE 10
Flatten Binary Tree

3.2 IMPLEMENTED FUNCTIONS

Listing 1 illustrates the three functions dedicated to allocation and listing 2 shows the three functions dedicated to deallocation.

```
/**
 * Allocate 4Kb page
 * return None if allocation fails
 */
pub fn allocate_frame(&mut self) -> Option<usize> {...}

/**
 * Allocate 2Mb page
 * return None if allocation fails
 */
pub fn allocate_big_page(&mut self) -> Option<usize> {...}

/**
 * Allocate 1Gb page
 * return None if allocation fails
 */
pub fn allocate_huge_page(&mut self) -> Option<usize> {...}
```

LISTING 1
Allocation Functions

```
/**
 * Deallocate frame
 * nothing is done if frame was not previously allocated
 */
pub fn deallocate_frame(&mut self, frame_id: usize) {...}

/**
 * Deallocate big page
 * nothing is done if page was not previously allocated
 */
pub fn deallocate_big_page(&mut self, frame_id: usize) {...}

/**
 * Deallocate huge page
 * nothing is done if page was not previously allocated
 */
pub fn deallocate_huge_page(&mut self, frame_id: usize) {...}
```

LISTING 2
Deallocation Functions

The six pseudo-codes to allocate and deallocate pages are presented in the annexes. The entire code is available on github [10]. The strategy to allocate a frame can be summarized as searching for an available frame in the corresponding tree. Then we set the necessary bits to used in the three trees.
The strategy for deallocation can be summarized as determining whether or not the given address is valid. Then, in the three trees, set the necessary bits to free.

3.3 ALLOCATION SIZE INFERENCE

Deallocation is safe in the sense that it has no undefined behaviour when given an invalid address; it does nothing instead. In fact, we automatically determine which block size is allocated. Figure 11 illustrates the code diagram that enables us to determine the type of allocated page given an address. For example, when an address is not aligned, we can discard it: a 2Mb page must have an address that is a multiple of 512 and a 1Gb page must have an address that is a multiple of 512^2. Other cases are solved by inspecting the three data structures.
Furthermore, to distinguish which type of block is allocated, we use the principle described in figure 5: children are not automatically set to 0.

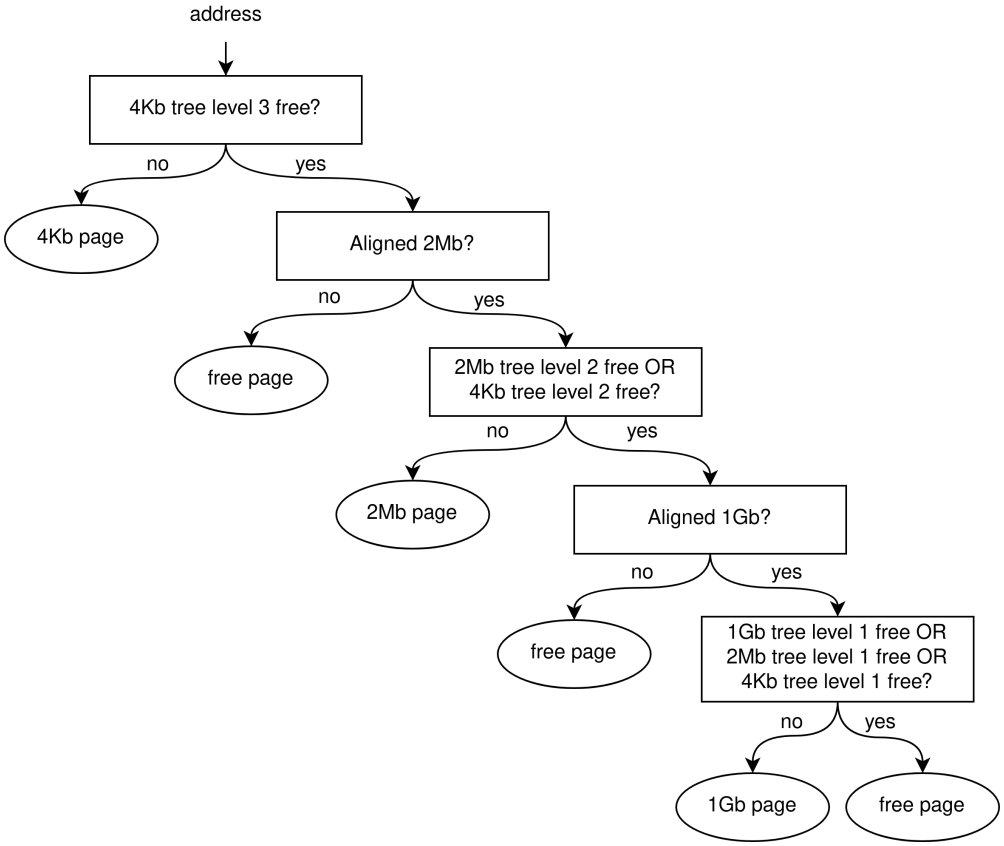

FIGURE 11
Code Diagram To Find Page Type

4 EVALUATION

In this section we evaluate the memory overhead of the additional structure, the advantage of using Intel-specific instructions and a measurement of external fragmentation given a predefined scenario. There is no comparison with other allocators such as the x86 page table allocator from Linux. There are too many dependencies to simply extract the standalone allocator's code. The benchmark only checks the behaviour of the algorithm against external fragmentation, which is not a real-world benchmark.
We measured allocation performance using a 2018 laptop equipped with an Intel core i7-8565U processor running at 1.80 GHz, and we were able to allocate 10 million pages per second.

4.1 Memory Overhead

Figure 12 depicts memory usage in relation to total memory available, we see that overhead accounts for $\sim 0.003\%$ of total available memory. This corresponds to approximately 1 bit per 4Kb page, so 1 bit per 4096 * 8 = 32768 bits. The 4Kb tree with its three levels is the largest contributor to overhead.

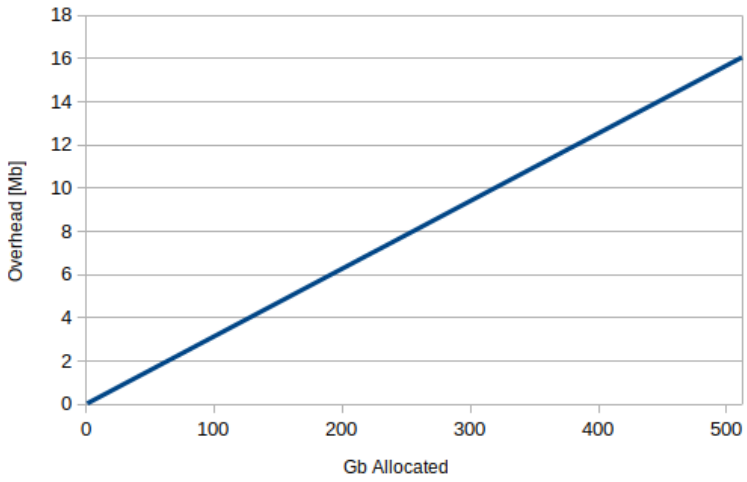

Figure 12
Memory Overhead

4.2 Finding First Bit Set

Figure 13 illustrates the average time of 1000 BSF and simple loop method iterations to find the first bit set from a 64-bit variable. We notice both BSF and *trailing_zeros* run in constant time.

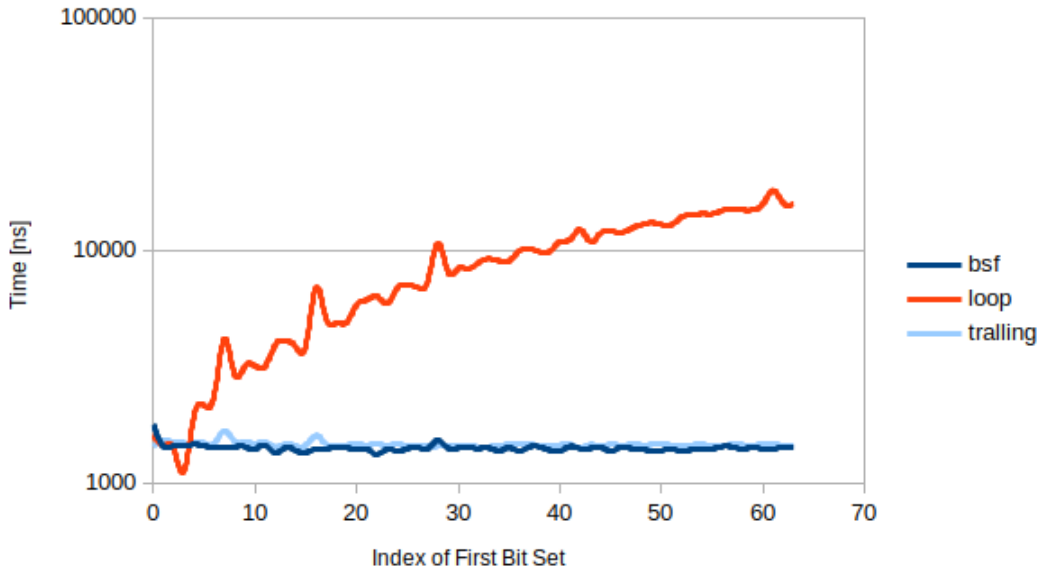

Figure 13
Benchmark BSF

Figure 14 depicts the average time of 1000 BSF and simple loop method iterations to search for the first bit set among a list of eight 64-bit variables totaling 512 bits. We continue to observe that the scaling of BSF-based methods is lower than that of the simple loop.

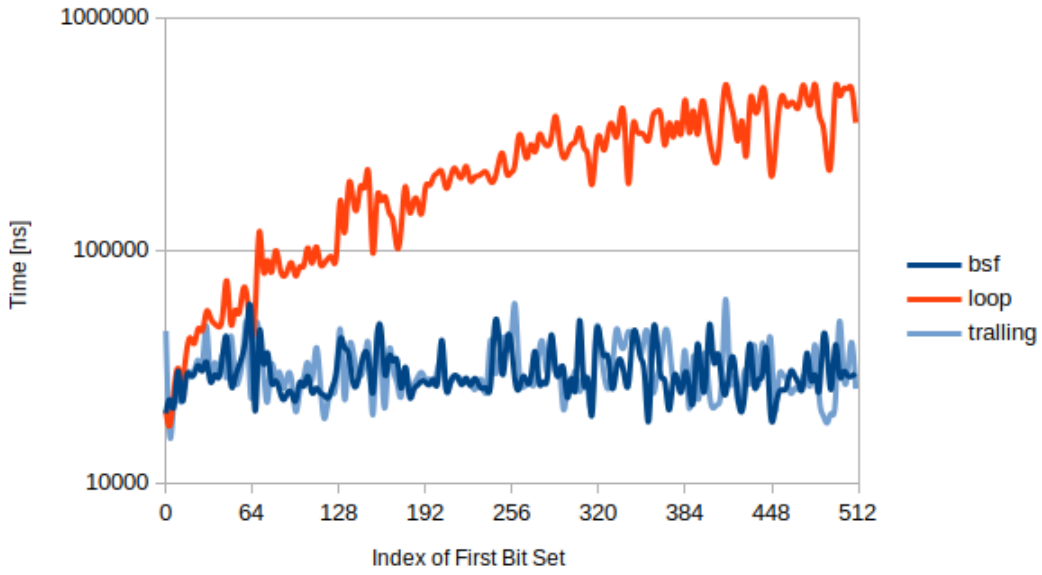

FIGURE 14
512-Bit Benchmark BSF

4.3 EXTERNAL FRAGMENTATION MEASUREMENT

The benchmark that follows evaluates the allocator's external fragmentation. The goal is to allocate and deallocate pages of varying sizes at random while measuring fragmentation. We want to reach a memory usage of 70%. To achieve this goal we use an inverse cumulative distributive Poisson function to determine the probability of doing an allocation, as illustrated in figure 15.

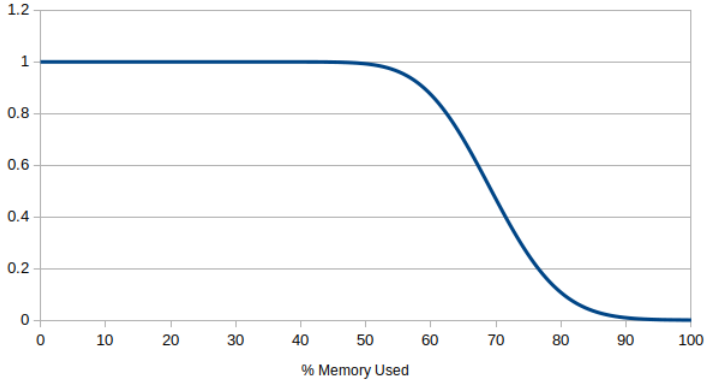

FIGURE 15
Inverse Poisson Cumulative Distribution

The goal is to have one-third occupations for each page size, probabilities are shown below:

$$P(allocate/deallocate\ 4Kb) = \frac{512^2}{512^2 + 512 + 1} \approx 0.9980$$
$$P(allocate/deallocate\ 2Mb) = \frac{512}{512^2 + 512 + 1} \approx 0.0019$$
$$P(allocate/deallocate\ 1Gb) = \frac{1}{512^2 + 512 + 1} \approx 0.3807 * 10^{-5}$$

Figures 16 and 17 illustrate the outcome of two billion successive random allocation and deallocation operations, with the x-axis representing the evolution with the new allocations and deallocations. Figure 16 shows the percentage of allocated blocks for each size, as well as the blocks that can only be allocated to 2Mb and 4Kb. One can see the external fragmentation on the top of the figure with the $4Kb\_free$ and $2Mb\_free$, indicating that this portion of the memory cannot be allocated to 1Gb pages. The goal is to have a large majority of $1Gb\_free$ for unused blocks because one $1Gb\_free$ can be used to allocate each of the three page sizes. The $4Kb\_free$ memory zone can only be used to allocate 4Kb pages. There is at most approximately 15% fragmentation, which seems reasonable.
Figure 17 illustrates the spatial representation of memory blocks. We observe 1Gb blocks are placed on the opposite side, since, as mentioned in section 2.5, the search direction for 1Gb pages is reversed.

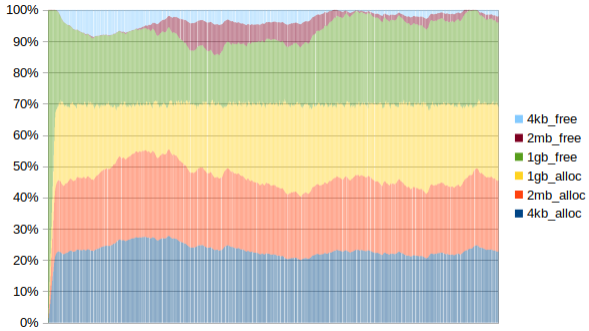

FIGURE 16
Quantitative Memory Representation

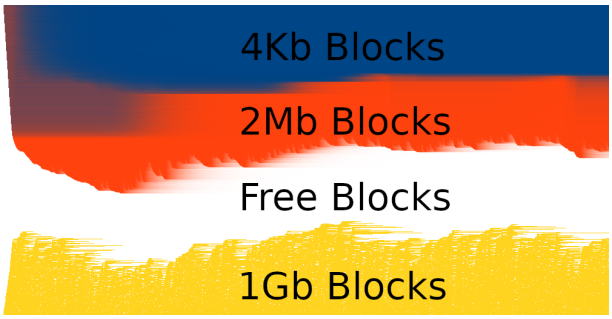

FIGURE 17
Spatial Memory Representation

As a point of comparison, figures 18 and 19 illustrate the case where all pages are allocated to the same side. We observe a higher fragmentation due to competition between 1Gb pages and other pages.

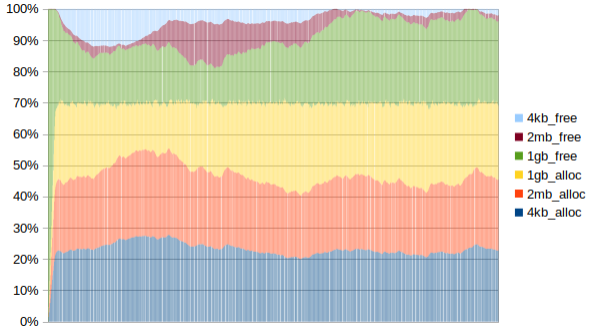

FIGURE 18
Quantitative Memory Representation

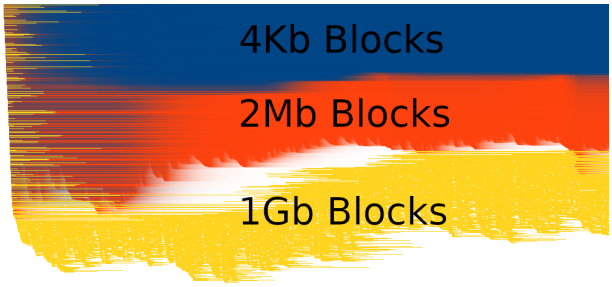

FIGURE 19
Spatial Memory Representation

5 CONCLUSION

We have an efficient algorithm that allows us to allocate three different block sizes in few operations, in a simple manner. Furthermore, the code contains no unbounded loops. With an Intel core i7-8565U running at 1.80 GHz, a 2018 laptop can allocate 10 million pages per second. The code consists of approximately 500 lines of code (LoC) [10].
The *bit scan forward* x86-64 assembly instruction improves allocator performance. Memory overhead is kept at 0.003%. Because allocations are done in a dummy manner, external fragmentation may occur. Furthermore, without the need of an additional memory structure, deallocations are safe because we have an allocation size inference.

5.1 FUTURE WORK

We observe a non-negligible external fragmentation during our benchmark, we could imagine a better algorithm which selects the block that creates the least fragmentation rather than the rightmost one.
To make our allocator more functional, we could implement a multi-threaded version of our algorithm.
The performance of our allocator could also be measured by the number of cycles required to complete each task.

REFERENCES

[1] *Rust*. 2022. URL: https://www.rust-lang.org/ (visited on 26th Dec. 2022).

[2] Yi Lin et al. 'Rust as a Language for High Performance GC Implementation'. In: *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*. ISMM 2016. Santa Barbara, CA, USA: Association for Computing Machinery, 2016, pp. 89–98. ISBN: 9781450343176. DOI: 10.1145/2926697.2926707. URL: https://doi.org/10.1145/2926697.2926707.

[3] Intel®. 'Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3 Part 1'. In: 2022. Chap. 4.1.1.

[4] Greg Gagne Abraham Silberschatz Peter Baer Galvin. *Operating System Concepts*. 2013. ISBN: 978-1-118-06333-0.

[5] Daan Leijen, Benjamin Zorn and Leonardo de Moura. 'Mimalloc: Free List Sharding in Action'. In: *Programming Languages and Systems*. Ed. by Anthony Widjaja Lin. Cham: Springer International Publishing, 2019, pp. 244–265. ISBN: 978-3-030-34175-6.

[6] J.M. Chang and E.F. Gehringer. 'A high performance memory allocator for object-oriented systems'. In: *IEEE Transactions on Computers* 45.3 (1996), pp. 357–366. DOI: 10.1109/12.485574.

[7] William D. McQuain. *Full and Complete Binary Trees*. 2009. URL: https://courses.cs.vt.edu/~cs3114/Fall10/Notes/T03a.BinaryTreeTheorems.pdf (visited on 26th Dec. 2022).

[8] Intel®. 'Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2 Part 1'. In: 2022. Chap. 3-125.

[9] Romolo Marotta et al. 'A Non-blocking Buddy System for Scalable Memory Allocation on Multi-core Machines'. In: *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. 2018, pp. 164–165. DOI: 10.1109/CLUSTER.2018.00034.

[10] David Desboeufs. *Frame Allocator*. https://github.com/Walterieux/vmxvmm_report. 2023.

[11] B. Randell. 'A Note on Storage Fragmentation and Program Segmentation'. In: *Commun. ACM* 12.7 (July 1969), 365–ff. ISSN: 0001-0782. DOI: 10.1145/363156.363158. URL: https://doi.org/10.1145/363156.363158.

[12] Ghods, Amir Reza. 'A Study of Linux Perf and Slab Allocation Sub-Systems'. MA thesis. 2016. URL: http://hdl.handle.net/10012/10184.

6 ANNEXES

6.1 ALLOCATE 4KB PAGE

Algorithm 1 $allocate\_frame()$

▷ Search free 4Kb page
$l1\_idx \leftarrow search\_first\_bit\_set(Tree4Kb,\ 0,\ LEVEL\_1)$
if $l1\_idx$ not valid **then**
    return None ▷ No available memory
end if

$l2\_child\_node \leftarrow compute\_node\_child(l1\_idx,\ LEVEL\_2)$
$l2\_idx \leftarrow search\_first\_bit\_set(Tree4Kb,\ l2\_child\_node,\ LEVEL\_2)$

$l3\_child\_node \leftarrow compute\_node\_child(l2\_idx,\ LEVEL\_3)$
$l3\_idx \leftarrow search\_first\_bit\_set(Tree4Kb,\ l3\_child\_node,\ LEVEL\_3)$

▷ 4Kb tree: set bits to 0
$set\_bit\_to\_zero(Tree4Kb,\ l3\_idx,\ LEVEL\_3)$
if 512 bits of $l3\_child\_node$'s 4Kb tree are used **then** ▷ OR operation
    $set\_bit\_to\_zero(Tree4Kb,\ l2\_idx,\ LEVEL\_2)$
    if 512 bits of $l2\_child\_node$'s 4Kb tree are used **then** ▷ OR operation
        $set\_bit\_to\_zero(Tree4Kb,\ l1\_idx,\ LEVEL\_1)$
    end if
end if

▷ 2Mb tree: set bits to 0
$set\_bit\_to\_zero(Tree2Mb,\ l2\_idx,\ LEVEL\_2)$
if 512 bits of $l2\_child\_node$'s 2Mb tree are used **then** ▷ OR operation
    $set\_bit\_to\_zero(Tree2Mb,\ l1\_idx,\ LEVEL\_1)$
end if

▷ 1Gb tree: set bit to 0
$set\_bit\_to\_zero(Tree1Gb,\ l1\_idx,\ LEVEL\_1)$

return $Some((l1\_idx\ <<\ 18)\ +\ (l2\_idx\ <<\ 9)\ +\ l3\_idx)$

6.2 ALLOCATE 2MB PAGE

Algorithm 2 $allocate\_big\_page()$

▷ Search free 2Mb page
$l1\_idx \leftarrow search\_first\_bit\_set(Tree2Mb,\ 0,\ LEVEL\_1)$
if $l1\_idx$ not valid **then**
  return None ▷ No available memory
end if

$l2\_child\_node \leftarrow compute\_node\_child(l1\_idx,\ LEVEL\_2)$
$l2\_idx \leftarrow search\_first\_bit\_set(Tree2Mb,\ l2\_child\_node,\ LEVEL\_2)$

▷ 2Mb tree: set bits to 0
$set\_bit\_to\_zero(Tree2Mb,\ l2\_idx,\ LEVEL\_2)$
if 512 bits of $l2\_child\_node$'s 2Mb tree are used **then** ▷ OR operation
  $set\_bit\_to\_zero(Tree2Mb,\ l1\_idx,\ LEVEL\_1)$
end if

▷ 4Kb tree: set bits to 0
$set\_bit\_to\_zero(Tree4Kb,\ l2\_idx,\ LEVEL\_2)$ ▷ Note $LEVEL\_3$ is not modified
if 512 bits of $l2\_child\_node$'s 4Kb tree are used **then** ▷ OR operation
  $set\_bit\_to\_zero(Tree4Kb,\ l1\_idx,\ LEVEL\_1)$
end if

▷ 1Gb tree: set bit to 0
$set\_bit\_to\_zero(Tree1Gb,\ l1\_idx,\ LEVEL\_1)$

return $Some((l1\_idx\ <<\ 18)\ +\ (l2\_idx\ <<\ 9))$

6.3 ALLOCATE 1GB PAGE

Algorithm 3 $allocate\_huge\_page()$

▷ Search free 1Gb page
$l1\_idx \leftarrow search\_first\_bit\_set(Tree1Gb,\ 0,\ LEVEL\_1)$
if $l1\_idx$ not valid **then**
  return None ▷ No available memory
end if

▷ 1Gb tree: set bit to 0
$set\_bit\_to\_zero(Tree1Gb,\ l1\_idx,\ LEVEL\_1)$

▷ 4Kb tree: set bit to 0
$set\_bit\_to\_zero(Tree4Kb,\ l1\_idx,\ LEVEL\_1)$

▷ 2Mb tree: set bit to 0
$set\_bit\_to\_zero(Tree2Mb,\ l1\_idx,\ LEVEL\_1)$

return $Some((l1\_idx\ <<\ 18))$

6.4 DEALLOCATE 4KB PAGE

Algorithm 4 $deallocate\_frame(frame\_id)$

if frame was not previously allocated **then**
    return
end if

▷ Extract level indices
$l3\_idx \leftarrow frame\_id \,\&\, 0x1FF$
$l2\_idx \leftarrow (frame\_id >> 9) \,\&\, 0x1FF$
$l1\_idx \leftarrow (frame\_id >> 18) \,\&\, 0x1FF$

▷ Free the 3 Level of 4Kb Tree
$set\_bit\_to\_one(Tree4Kb,\ l1\_idx,\ LEVEL\_1)$
$set\_bit\_to\_one(Tree4Kb,\ l2\_idx,\ LEVEL\_2)$
$set\_bit\_to\_one(Tree4Kb,\ l3\_idx,\ LEVEL\_3)$

▷ if all 4Kb are free, free upper level for 2Mb
if $are\_all\_free(Tree4Kb,\ l3\_idx,\ LEVEL\_3)$ **then**
    $set\_bit\_to\_one(Tree2Mb,\ l1\_idx,\ LEVEL\_1)$
    $set\_bit\_to\_one(Tree2Mb,\ l2\_idx,\ LEVEL\_2)$
end if

▷ if all 2Mb are free, free 1Gb
if $are\_all\_free(Tree2Mb,\ l2\_idx,\ LEVEL\_2)$ **then**
    $set\_bit\_to\_one(Tree1Gb,\ l1\_idx,\ LEVEL\_1)$
end if

6.5 DEALLOCATE 2MB PAGE

Algorithm 5 $deallocate\_big\_page(frame\_id)$

if frame was not previously allocated **then**
  return
end if

▷ Extract level indices
$l2\_idx \leftarrow (frame\_id >> 9)\ \&\ 0x1FF$
$l1\_idx \leftarrow (frame\_id >> 18)\ \&\ 0x1FF$

▷ Free the 2 Level of 2Mb Tree
$set\_bit\_to\_one(Tree2Mb,\ l1\_idx,\ LEVEL\_1)$
$set\_bit\_to\_one(Tree2Mb,\ l2\_idx,\ LEVEL\_2)$

▷ Free the 2 upper Level of 4Kb Tree
$set\_bit\_to\_one(Tree4Kb,\ l1\_idx,\ LEVEL\_1)$
$set\_bit\_to\_one(Tree4Kb,\ l2\_idx,\ LEVEL\_2)$

▷ if all 2Mb are free, free 1Gb
if $are\_all\_free(Tree2Mb,\ l2\_idx,\ LEVEL\_2)$ **then**
  $set\_bit\_to\_one(Tree1Gb,\ l1\_idx,\ LEVEL\_1)$
end if

6.6 DEALLOCATE 1GB PAGE

Algorithm 6 $deallocate\_huge\_page(frame\_id)$

if frame was not previously allocated **then**
  return
end if

▷ Extract level index
$l1\_idx \leftarrow (frame\_id >> 18)\ \&\ 0x1FF$

▷ Free all the Level 1
$set\_bit\_to\_one(Tree1Gb,\ l1\_idx,\ LEVEL\_1)$
$set\_bit\_to\_one(Tree4Kb,\ l1\_idx,\ LEVEL\_1)$
$set\_bit\_to\_one(Tree2Mb,\ l1\_idx,\ LEVEL\_1)$