**Heap Allocation**

The most adaptable allocation strategy is heap allocation. Memory can be allocated and deallocated at any moment, when it is required, based on the needs of the user. Memory is dynamically allocated to variables using heap allocation, which then claims the memory back when the variables are no longer in use [1].

In kernel there are two types of variables, and they are Local variables and Static Variables

**Local variables**

Stack data structure (call stack) is which supports push and pop operations that the local variables are stored. Local variables, return address and the parameters of the called function of each function entry were pushed by the compiler:

The call stack is displayed in the example above after the outer function has called the inner function. The local variables of outer first are visible in the call stack. The parameter 1 and the function's return address were pushed on the inner call.

Then the inner gets the control which pushed its local variables

When the inner function returns, its section of the call stack is popped, leaving only the outer function's local variables:

**Static variables**

Static variables are kept in a dedicated area of memory that is not part of the stack. The linker assigns this memory location at compile time, and the executable is coded using this information. Statics have a static lifetime and can always be linked from local variables because they last the entire duration of the program:

In the example above, the call stack is cleared when the inner function returns. The &Z[1] reference remains valid after the return because the static variables are stored in a different memory region that is never deleted.

**Dynamic Memory**

Because of these static and local variables’ limitations like:

* Both variables having a fixed size and the dynamically growing elements cannot store as a collection
* Local variables are live on the call stack and when the surrounding function returns, they were destroyed
* Static variables are living until e=the end of the program so when the memory no longer needed there’s no way to reuse or reclaim

So, because of these limitations programming languages uses heap for storing variables as a third memory. The allocate and deallocate functions of the heap enable dynamic memory allocation at runtime. A free memory space of the given size is returned by the allocate function which can be used to hold a variable. When the deallocate function is used with a reference to the variable, the variable remains in existence until it is released [2].

**Common Errors**

There are two typical bug kinds with more serious repercussions, aside from memory leaks, which are regrettable but don't leave the software open to attackers.

Use-after-free vulnerability – after calling deallocate on a variable accidentally continue to use it

Double-free vulnerability – accidentally freeing a variable twice

Rust makes use of a concept called ownership to ensure the accuracy of dynamic memory operations at build time. As a result, there isn't any performance overhead and the above-mentioned vulnerabilities can be avoided without the need of trash collection. Another benefit of this strategy is that, like with C or C++, the programmer retains fine-grained control over the utilization of dynamic memory.

**Allocations in Rust**

The Rust standard library offers abstraction types which implicitly call allocation and deallocate rather than allowing the programmer to individually invoke these functions. Box, the abstraction for a heap-allocated value, is the most crucial type. It offers a constructor function called Box::new that accepts a value, calls allocate with the size of the value, and then transfers the value to the heap's newly created slot. The Box type implements the Drop trait to call deallocate when it exits scope in order to release the heap memory once more.

So, the rust ownership assigns lifetime abstracts for the references. Also, Like garbage collected languages such as Python or Java do, they also provide total memory safety, preventing use-after-free problems. It also ensures thread safety, making it even safer in multi-threaded programs than those languages. Most crucially, there isn't any runtime overhead compared to manually implemented memory management in C because all these tests take place at compile time [2].

**The Allocator Interface**

Adding a dependence on the built-in alloc crate is the first step in the heap allocator implementation process. This is a subset of a standard library which also includes the collection and allocation types, much like the core crate.

We don't have to change the Cargo.toml, in contrast to typical dependencies. The alloc crate is included in the standard library with the Rust compiler, therefore the compiler is already aware of the crate. We tell the compiler to try to include it by adding the extern crate declaration.

Then the compiler include the alloc crate in our kernel. When tring to compile this occurs two errors

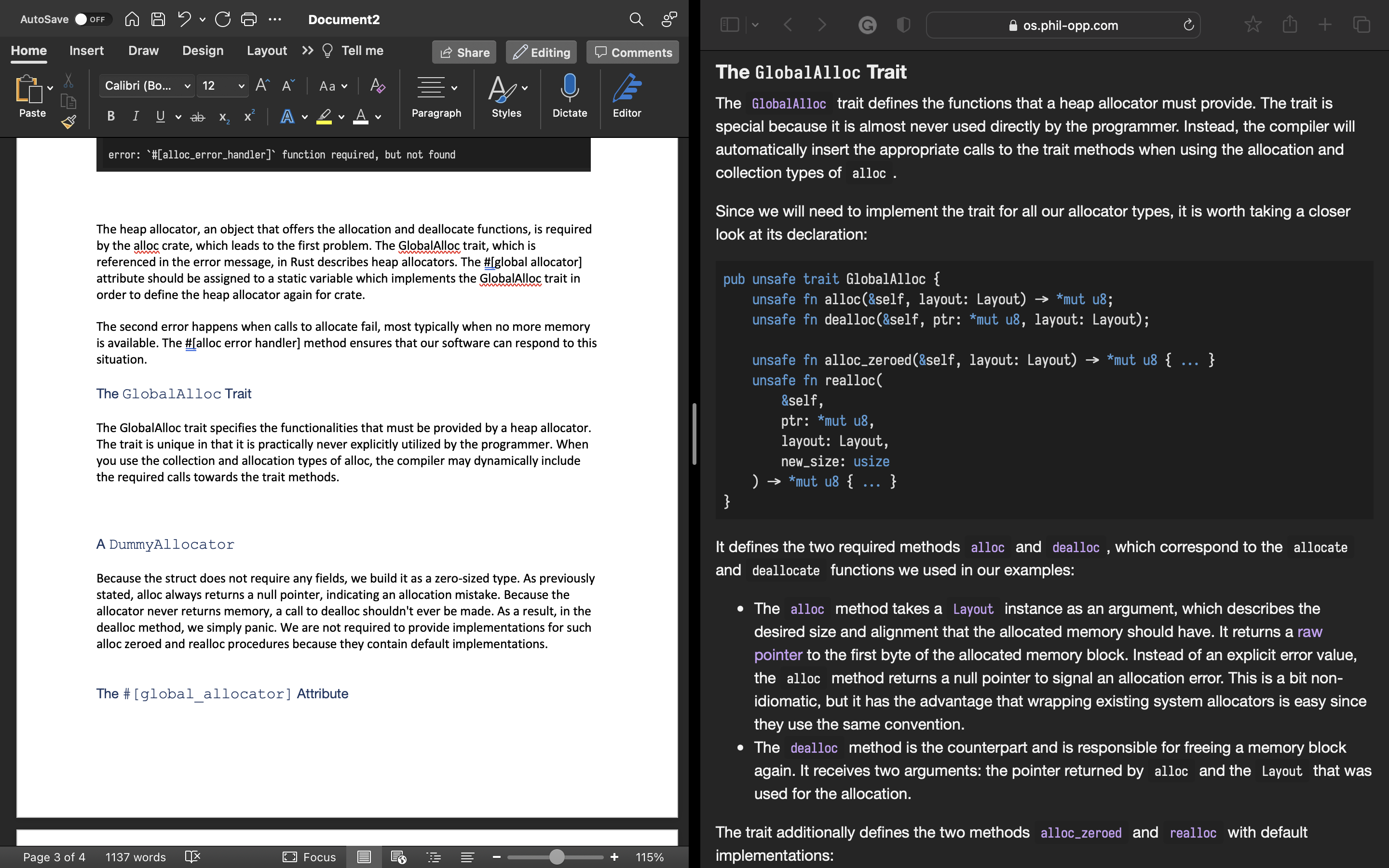


The heap allocator, an object that offers the allocation and deallocate functions, is required by the alloc crate, which leads to the first problem. The GlobalAlloc trait, which is referenced in the error message, in Rust describes heap allocators. The #[global allocator] attribute should be assigned to a static variable which implements the GlobalAlloc trait in order to define the heap allocator again for crate.

The second error happens when calls to allocate fail, most typically when no more memory is available. The #[alloc error handler] method ensures that our software can respond to this situation.

### The GlobalAlloc Trait

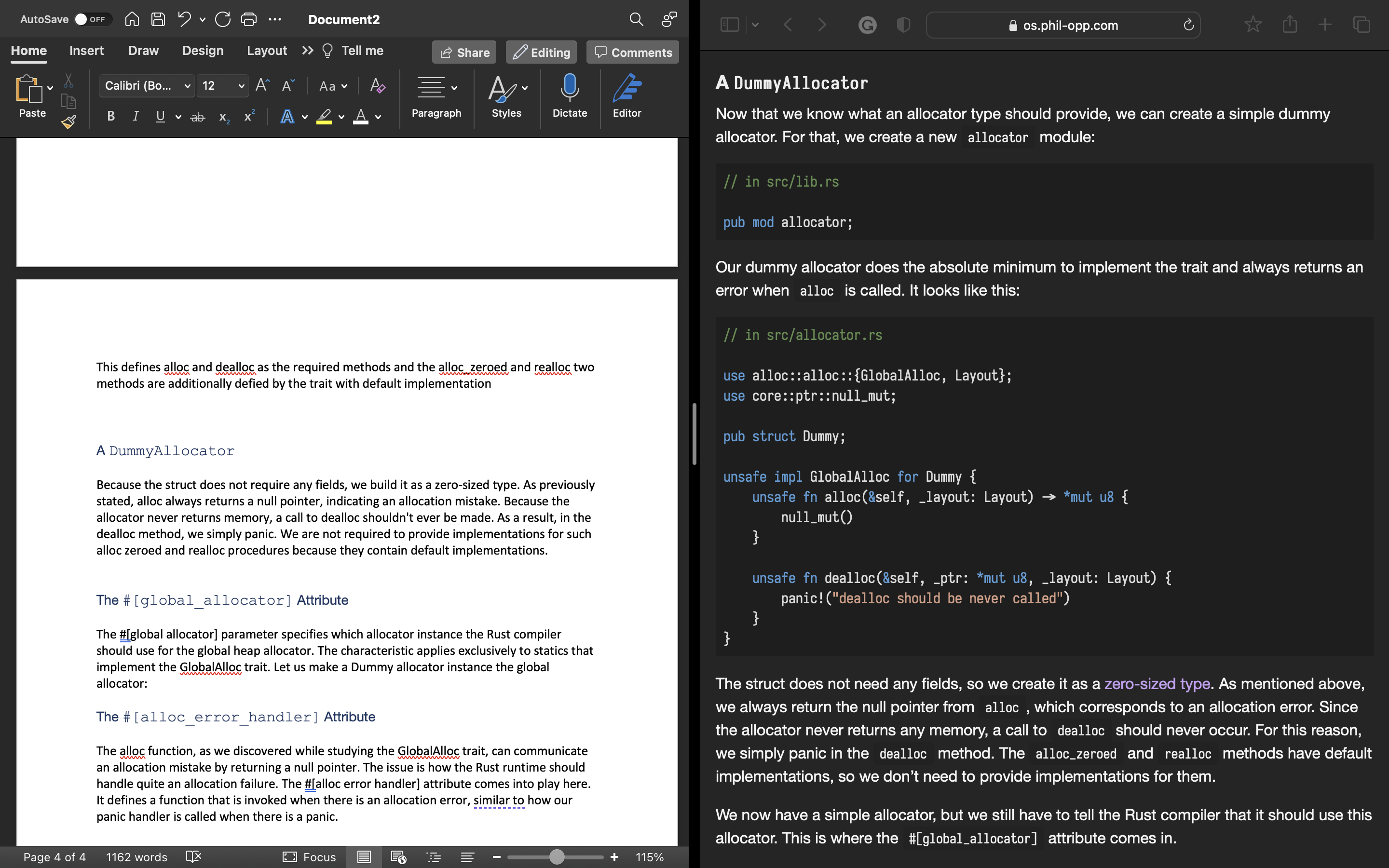
The GlobalAlloc trait specifies the functionalities that must be provided by a heap allocator. The trait is unique in that it is practically never explicitly utilized by the programmer. When you use the collection and allocation types of alloc, the compiler may dynamically include the required calls towards the trait methods [3].



This defines alloc and dealloc as the required methods and the alloc\_zeroed and realloc two methods are additionally defied by the trait with default implementation

### A DummyAllocator

Because the struct does not require any fields, we build it as a zero-sized type. As previously stated, alloc always returns a null pointer, indicating an allocation mistake. Because the allocator never returns memory, a call to dealloc shouldn't ever be made. As a result, in the dealloc method, we simply panic. We are not required to provide implementations for such alloc zeroed and realloc procedures because they contain default implementations.



### The #[global\_allocator] Attribute

The #[global allocator] parameter specifies which allocator instance the Rust compiler should use for the global heap allocator. The characteristic applies exclusively to statics that implement the GlobalAlloc trait. Let us make a Dummy allocator instance the global allocator:

### The #[alloc\_error\_handler] Attribute

The alloc function, as we discovered while studying the GlobalAlloc trait, can communicate an allocation mistake by returning a null pointer. The issue is how the Rust runtime should handle quite an allocation failure. The #[alloc error handler] attribute comes into play here. It defines a function that is invoked when there is an allocation error, similar to how our panic handler is called when there is a panic.

## Creating a Kernel Heap

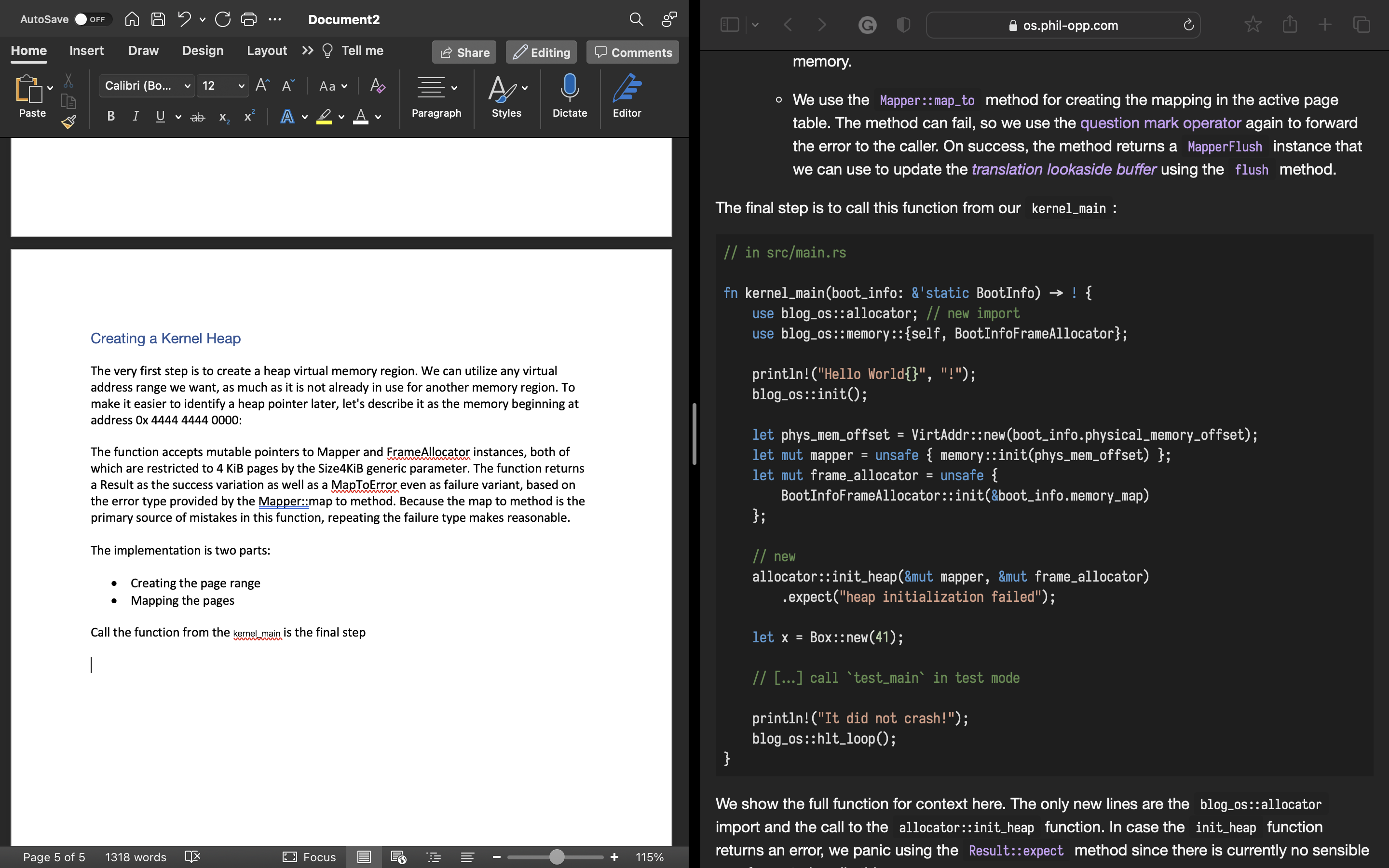
The very first step is to create a heap virtual memory region. We can utilize any virtual address range we want, as much as it is not already in use for another memory region. To make it easier to identify a heap pointer later, let's describe it as the memory beginning at address 0x 4444 4444 0000:

The function accepts mutable pointers to Mapper and FrameAllocator instances, both of which are restricted to 4 KiB pages by the Size4KiB generic parameter. The function returns a Result as the success variation as well as a MapToError even as failure variant, based on the error type provided by the Mapper::map to method. Because the map to method is the primary source of mistakes in this function, repeating the failure type makes reasonable.

The implementation is two parts:

* Creating the page range
* Mapping the pages

Call the function from the kernel\_main is the final step



**Allocator Designs**

Previously added basic support for heap allocations to our kernel by creating a new memory region in the page tables and manage the memory by using linked\_list\_allocator crate. In allocator designs creating own heap allocator from scratch and discuss different allocator designs and these help to improve performance.

**Design Goals**

An allocator's task is to control the heap memory that is accessible. It must return unused memory on alloc calls and maintain track of memory freed by dealloc in order for it to be reused. epesially , it should never allocate memory that is currently in use someplace else, as this would result in erratic behavior and there are many secondary design goal. Those requirements would make finding good allocators difficult [2].

Lets see three of possible kernel allocator designs:

## Bump Allocator

This initializes in a linear fashion and merely maintains track of the amount of granted bytes and allocations. The concept behind a bump allocator is to allocate memory linearly by increasing ("bumping") a next variable that links to the beginning of the unused memory. At the start, next will be equal to the heap's start address.

### Implementation

The heap start and heap end fields keep track of the heap memory region's lower and higher bounds. The caller must guarantee that these addresses are correct, else the allocator will return invalid memory. As a result, the init function must be hazardous to call.

For keep this interface identical towards the allocator supplied by the linked list allocator crate, we elected to construct a separate init method rather than executing the initialization straight in new. This allows the allocators to be altered without requiring any extra code changes.Graphical user interface, text

Description automatically generated

### Implementing GlobalAlloc - Because no limits checks or alignment modifications are performed, this approach is not yet safe. Because there are some errors with alloc and dealloc methods. The problem is bump allocator’s essential principles are the updating next on every allocation

#### GlobalAlloc and Mutability. – The #[global allocator] attribute is added to the a static that defines the GlobalAlloc trait. Because static variables in Rust are immutable, there isn't any way to call a function that takes &mut self on the static allocator. As a result, all GlobalAlloc methods only accept an immutable &self reference.

#### A Locked Wrapper Type. - For the bump allocator, implementing the GlobalAlloc trait with the aid of the spin::Mutex wrapper type. The idea is to implement the trait for the wrapped spin::MutexBumpAllocator> type rather than the BumpAllocator directly. But this doesn’t word because trait implementations for types declared in other crates are not permitted by the Rust compiler

#### Implementation for Locked<BumpAllocator> - h this is using for implement GlobalAlloc for the bump allocator.

Performance- The main advantage with bump allocation is its speed. In contrast to other allocator designs (see below), which must actively search for a suitable memory block and conduct different accounting operations on alloc and dealloc, a bump allocator can really be optimized to only a few assembly instructions. As a result, bump allocators are useful for optimizing allocation performance, such as when developing a virtual DOM library [2].

## Linked List Allocator

When implementing allocators, a typical approach for keeping a record of an arbitrary number of available memory areas is to use these areas as backing storage. This capitalizes on the fact that the regions are all still mapped to a virtual address and supported by a physical frame, however the stored information is no longer required. We can maintain track of an infinite number of released regions by storing information about the freed region within the region itself.

### Implementation

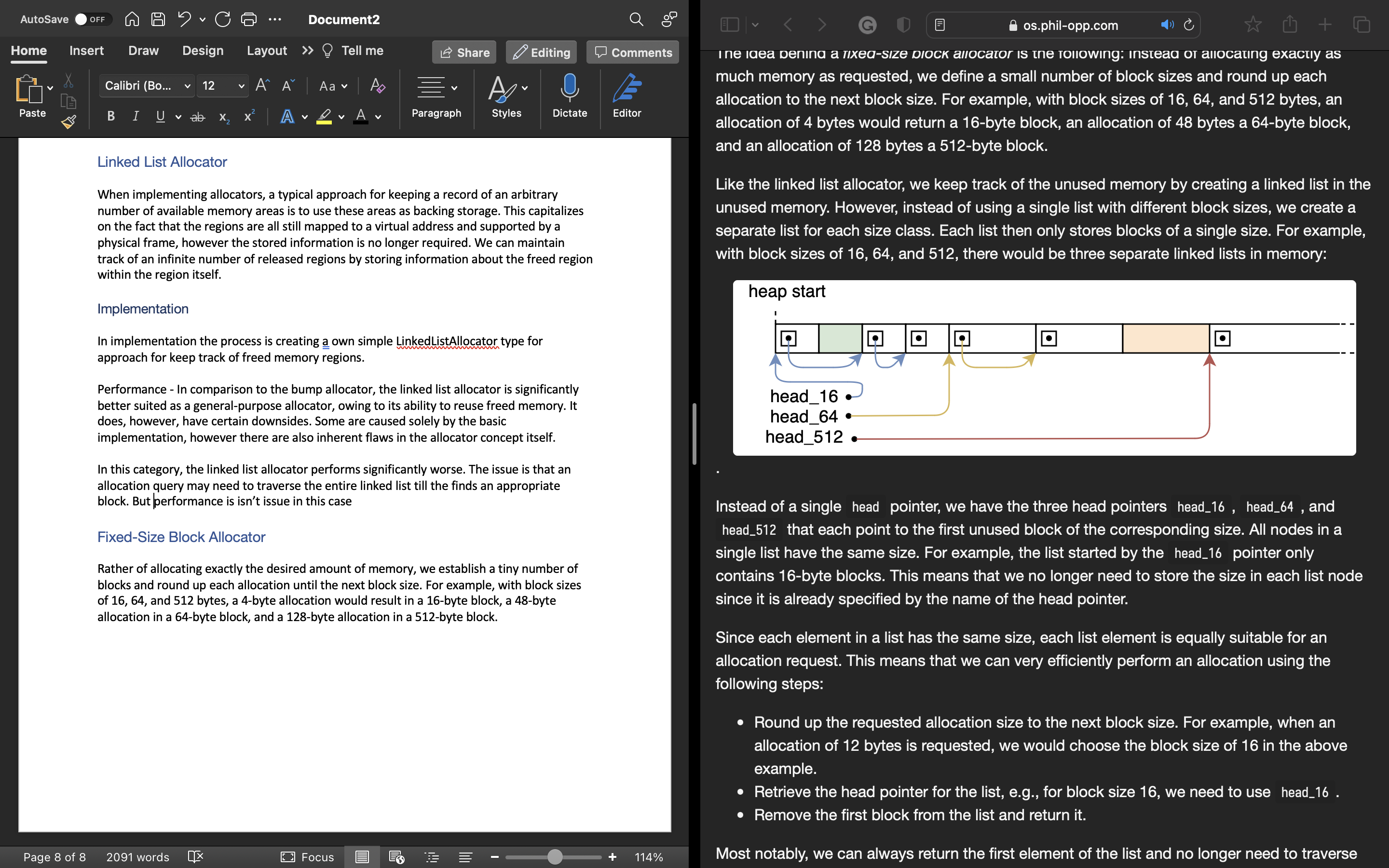
In implementation the process is creating a own simple LinkedListAllocator type for approach for keep track of freed memory regions.

Performance - In comparison to the bump allocator, the linked list allocator is significantly better suited as a general-purpose allocator, owing to its ability to reuse freed memory. It does, however, have certain downsides. Some are caused solely by the basic implementation, however there are also inherent flaws in the allocator concept itself.

In this category, the linked list allocator performs significantly worse. The issue is that an allocation query may need to traverse the entire linked list till the finds an appropriate block. But performance is isn’t issue in this case [2].

## Fixed-Size Block Allocator

Rather of allocating exactly the desired amount of memory, we establish a tiny number of blocks and round up each allocation until the next block size. For example, with block sizes of 16, 64, and 512 bytes, a 4-byte allocation would result in a 16-byte block, a 48-byte allocation in a 64-byte block, and a 128-byte allocation in a 512-byte block.



There are three head pointers head\_16 , head\_64 and head\_512

**Block Sizes and Wasted Memory**

Rounding up based on the block sizes , a lot of memory can be lose. In the worst case half of allocation size and in the average cases quarter of the allocation size of memory waste limit can be limit.

#### Fallback Allocator

Given the rarity of large allocations (>2 KB), particularly in operating system kernels, it may make sense to use a specialized allocator for these allocations. Because only a few allocations of such magnitude are planned, the linked list would remain short and the (de)allocations would remain quite fast.

#### Creating new Blocks

When fulfilling all allocation requests some point, the linked block size becomes empty. At this stage, there are two options for creating new unused blocks of a specified size to satisfy an allocation request:

# Async/Await

async/.await are Rust language extensions that allow you to cede control of the current thread instead of blocking, allowing other code to run while waiting for an operation to complete.

Async can be used in two ways: async fn and async blocks. Each one returns a result that satisfies the Future trait:

Text

Description automatically generated

As we learned in the first chapter, async bodies and other futures are inactive until they are run. The most frequent method for running a Future is to.await it. When.await is used on a Future, it attempts to complete it. If the Future is blocked, control of the current thread is relinquished. When more progress can be achieved, the executor can grab up the Future and begin running it, allowing the.await to resolve.

## [async Lifetimes](https://rust-lang.github.io/async-book/03_async_await/01_chapter.html#async-lifetimes)

This means that an async fn's future must be.awaited while its non-'static arguments are still valid. This is not an issue in the common scenario of.awaiting the future right after running the function (as in foo(&x).await). This may be an issue if you are storing the future or passing it to another job or thread.

One typical solution for converting an async fn with references-as-arguments into a'static future is to wrap the arguments in an async block:

## [async move](https://rust-lang.github.io/async-book/03_async_await/01_chapter.html#async-move)

As with conventional closures, async blocks and closures support the move keyword. An async move block will assume ownership of the variables it references, allowing it to outlive the current scope while denying other programs the opportunity to share those variables:

## [.awaiting on a Multithreaded Executor](https://rust-lang.github.io/async-book/03_async_await/01_chapter.html#awaiting-on-a-multithreaded-executor)

Because a Future can move between threads when utilizing a multithreaded Future executor, all variables used in async bodies must be able to do the same. await has the possibility to switch to a new thread.

This means that using Rc, &RefCell, or any other types that do not implement the Send trait, including references to types that do not implement the Sync trait, is not safe.

(Caveat: these types can be used as long as they are not in scope during a call to.await.)

Similarly, holding a standard non-futures-aware lock across an.await may cause the threadpool to lock up: one job may take out a lock,.await, and surrender to the executor, allowing another task to attempt to take the lock and produce a deadlock. To avoid this, use the Mutex from futures::lock instead of the Mutex from std::sync [2].

Futures

A future reflects a resource which is not yet accessible.This might be an integer calculated by another process or a file obtained from the network. Instead of waiting until the value is ready, futures allow you to continue execution until the value is required.

Working with Futures

Another more accurate method would be to pause the current thread until the future becomes accessible. This is,  obviously, only feasible if there are threads, therefore this method is not suitable for the kernel, at certainly still not. But on systems that enable blocking, it is frequently undesirable since it converts an asynchronous operation into a synchronous activity, so limiting the potential performance gains.

# References

|  |  |
| --- | --- |
| [1] | Ginni, 8 Nov 2021. [Online]. Available: https://www.tutorialspoint.com/what-is-heap-allocation. |
| [2] | 2020. [Online]. Available: https://os.phil-opp.com/async-await/. |
| [3] | "rust," 2020. [Online]. Available: https://doc.rust-lang.org/std/alloc/trait.Allocator.html. |
| [4] | https://os.phil-opp.com/async-await/. [Online]. |