北京理工大学

本科生毕业设计（论文）外文翻译

外文原文题目： **Paging Implementation**

中文翻译题目： **分页实现**

rCore 模块化改进的设计与实现

学　　院：	计算机学院
专　　业：	计算机科学与技术
班　　级：	07111703
学生姓名：	石文龙
学　　号：	1120173592
指导教师：	陆慧梅

原文内容：

Paging Implementation

Abstract

This post shows how to implement paging support in our kernel. It first explores different techniques to make the physical page table frames accessible to the kernel and discusses their respective advantages and drawbacks. It then implements an address translation function and a function to create a new mapping.

The previous post gave an introduction to the concept of paging. It motivated paging by comparing it with segmentation, explained how paging and page tables work, and then introduced the 4-level page table design of x86_64. We found out that the bootloader already set up a page table hierarchy for our kernel, which means that our kernel already runs on virtual addresses. This improves safety since illegal memory accesses cause page fault exceptions instead of modifying arbitrary physical memory.

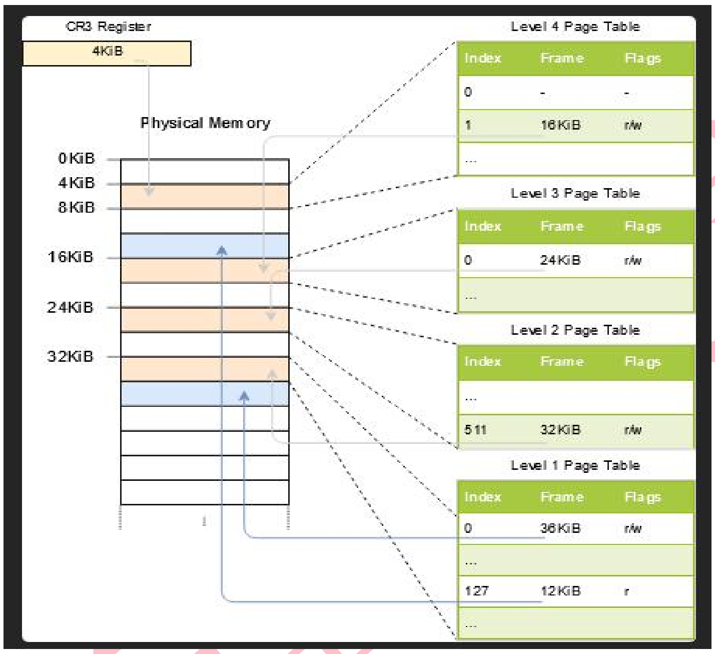
The post ended with the problem that we can’t access the page tables from our kernel because they are stored in physical memory and our kernel already runs on virtual addresses. This post explores different approaches to making the page table frames accessible to our kernel. We will discuss the advantages and drawbacks of each approach and then decide on an approach for our kernel.

To implement the approach, we will need support from the bootloader, so we’ll configure it first. Afterward, we will implement a function that traverses the page table hierarchy in order to translate virtual to physical addresses. Finally, we learn how to create new mappings in the page tables and how to find unused memory frames for creating new page tables.

directory

1. Accessing Page Tables

Accessing the page tables from our kernel is not as easy as it may seem. To understand the problem, let’s take a look at the example 4-level page table hierarchy from the previous post again:



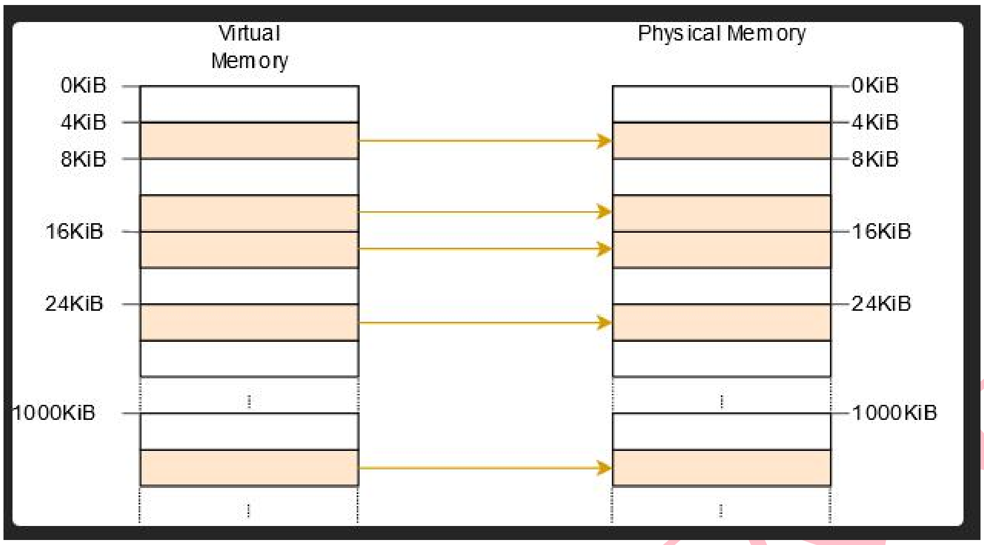
The important thing here is that each page entry stores the physical address of the next table. This avoids the need to run a translation for these addresses too, which would be bad for performance and could easily cause endless translation loops.

The problem for us is that we can’t directly access physical addresses from our kernel since our kernel also runs on top of virtual addresses. For example, when we access address 4 KiB we access the virtual address 4 KiB, not the physical address 4 KiB where the level 4 page table is stored. When we want to access the physical address 4 KiB, we can only do so through some virtual address that maps to it.

So in order to access page table frames, we need to map some virtual pages to them. There are different ways to create these mappings that all allow us to access arbitrary page table frames.

1.1 Identity Mapping

A simple solution is to identity map all page tables:



In this example, we see various identity-mapped page table frames. This way, the physical addresses of page tables are also valid virtual addresses so that we can easily access the page tables of all levels starting from the CR3 register.

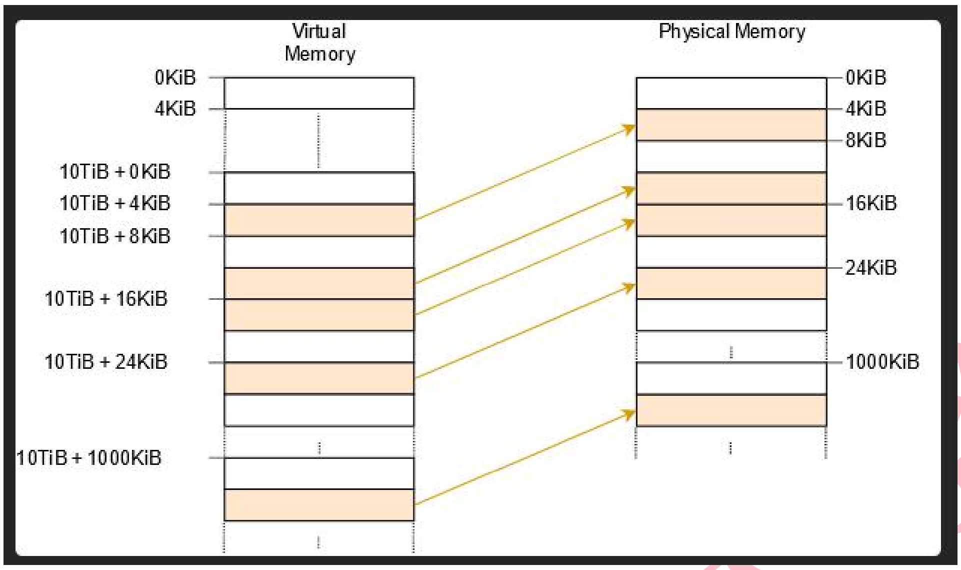
However, it clutters the virtual address space and makes it more difficult to find continuous memory regions of larger sizes. For example, imagine that we want to create a virtual memory region of size 1000 KiB in the above graphic, e.g., for memory-mapping a file. We can’t start the region at 28 KiB because it would collide with the already mapped page at 1004 KiB. So we have to look further until we find a large enough unmapped area, for example at 1008 KiB. This is a similar fragmentation problem as with segmentation.

Equally, it makes it much more difficult to create new page tables because we need to find physical frames whose corresponding pages aren’t already in use. For example, let’s assume that we reserved the virtual 1000 KiB memory region starting at 1008 KiB for our memory-mapped file. Now we can’t use any frame with a physical address between 1000 KiB and 2008 KiB anymore, because we can’t identity map it.

1.2 Map at a Fixed Offset

To avoid the problem of cluttering the virtual address space, we can use a separate memory region for page table mappings. So instead of identity mapping page table frames, we map them at a fixed offset in the virtual address space. For example, the offset could be

10 TiB:

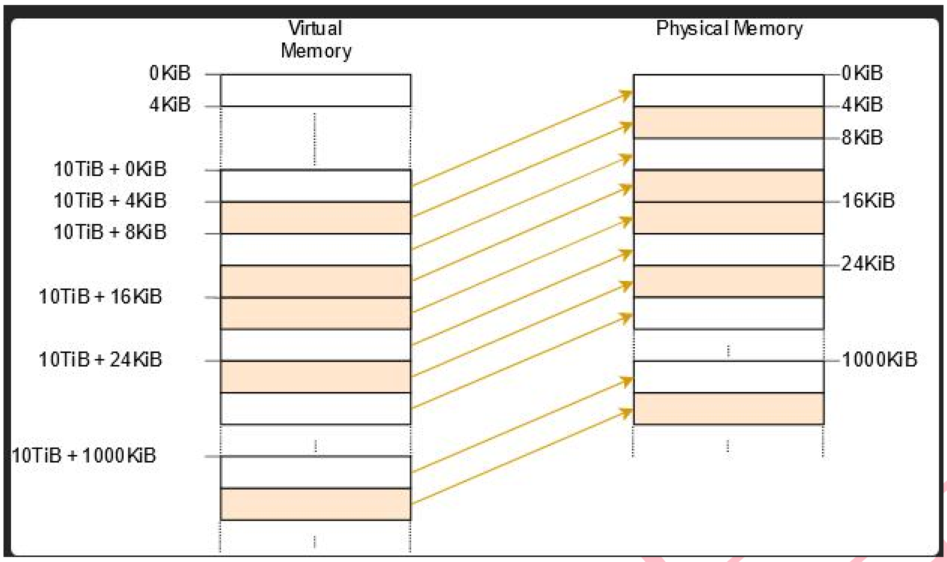


By using the virtual memory in the range 10 TiB..(10 TiB + physical memory size) exclusively for page table mappings, we avoid the collision problems of the identity mapping. Reserving such a large region of the virtual address space is only possible if the virtual address space is much larger than the physical memory size. This isn't a problem on x86_64 since the 48-bit address space is 256 TiB large.

This approach still has the disadvantage that we need to create a new mapping whenever we create a new page table. Also, it does not allow accessing page tables of other address spaces, which would be useful when creating a new process.

1.3 Map the Complete Physical Memory

We can solve these problems by mapping the complete physical memory instead of



only page table frames:

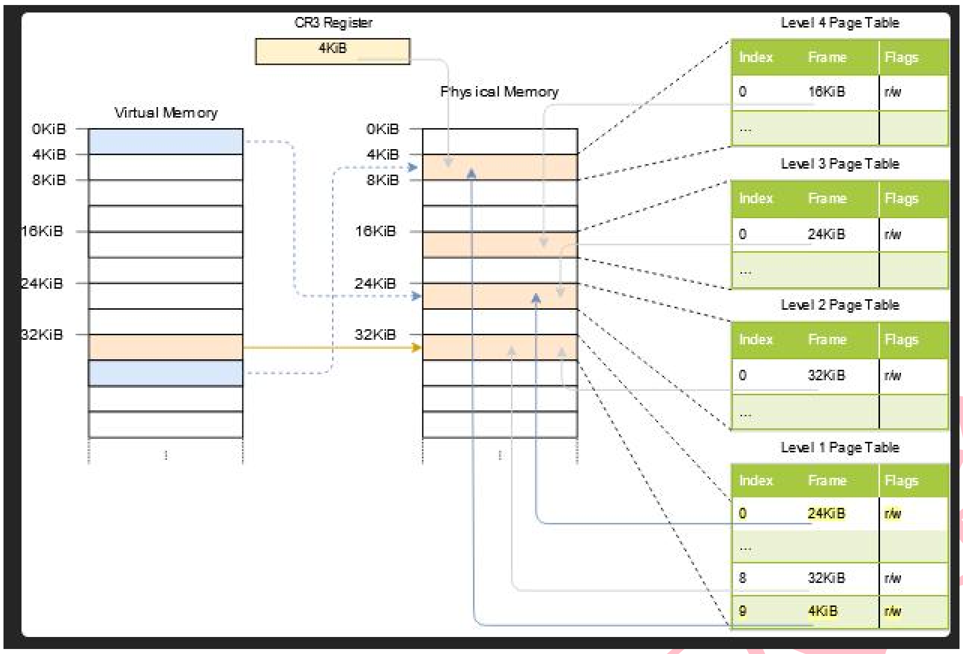
This approach allows our kernel to access arbitrary physical memory, including page table frames of other address spaces. The reserved virtual memory range has the same size as before, with the difference that it no longer contains unmapped pages.

The disadvantage of this approach is that additional page tables are needed for storing the mapping of the physical memory. These page tables need to be stored somewhere, so they use up a part of physical memory, which can be a problem on devices with a small amount of memory.

On x86_64, however, we can use huge pages with a size of 2 MiB for the mapping, instead of the default 4 KiB pages. This way, mapping 32 GiB of physical memory only requires 132 KiB for page tables since only one level 3 table and 32 level 2 tables are needed. Huge pages are also more cache efficient since they use fewer entries in the translation lookaside buffer (TLB).

1.4 Temporary Mapping

For devices with very small amounts of physical memory, we could map the page table frames only temporarily when we need to access them. To be able to create the temporary mappings, we only need a single identity-mapped level 1 table:



The level 1 table in this graphic controls the first 2 MiB of the virtual address space. This is because it is reachable by starting at the CR3 register and following the 0th entry in the level 4, level 3, and level 2 page tables. The entry with index 8 maps the virtual page at address 32 KiB to the physical frame at address 32 KiB, thereby identity mapping the level 1 table itself. The graphic shows this identity-mapping by the horizontal arrow at 32 KiB.

By writing to the identity-mapped level 1 table, our kernel can create up to 511 temporary mappings (512 minus the entry required for the identity mapping). In the above example, the kernel created two temporary mappings:

By mapping the 0th entry of the level 1 table to the frame with address 24 KiB, it created a temporary mapping of the virtual page at 0 KiB to the physical frame of the level 2 page table, indicated by the dashed arrow.

By mapping the 9th entry of the level 1 table to the frame with address 4 KiB, it created a temporary mapping of the virtual page at 36 KiB to the physical frame of the level 4 page table, indicated by the dashed arrow.

Now the kernel can access the level 2 page table by writing to page 0 KiB and the level 4 page table by writing to page 36 KiB.

The process for accessing an arbitrary page table frame with temporary mappings would be:

Search for a free entry in the identity-mapped level 1 table.

Map that entry to the physical frame of the page table that we want to access.

Access the target frame through the virtual page that maps to the entry.

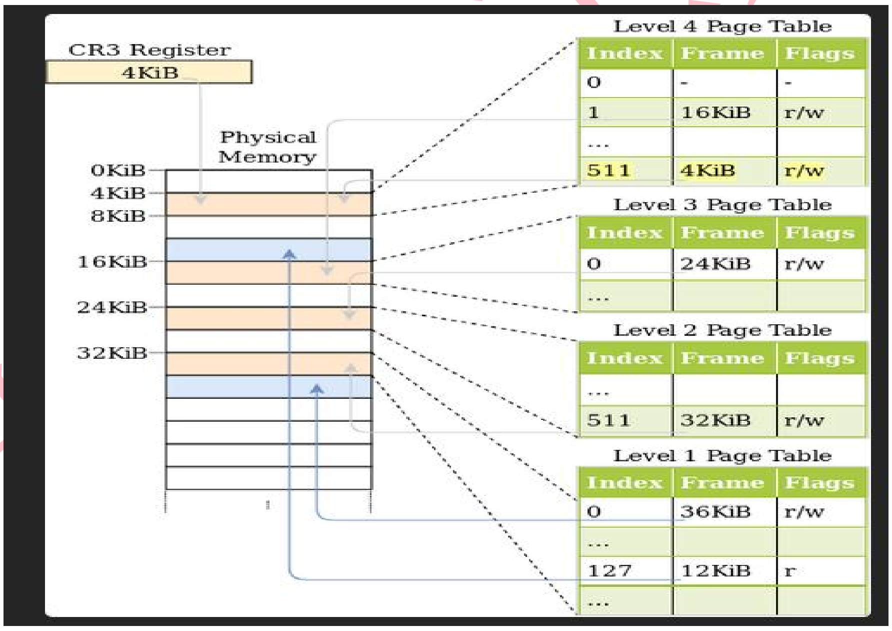
Set the entry back to unused, thereby removing the temporary mapping again.

This approach reuses the same 512 virtual pages for creating the mappings and thus requires only 4 KiB of physical memory. The drawback is that it is a bit cumbersome, especially since a new mapping might require modifications to multiple table levels, which means that we would need to repeat the above process multiple times.

1.5 Recursive Page Tables

Another interesting approach, which requires no additional page tables at all, is to map the page table recursively. The idea behind this approach is to map an entry from the level 4 page table to the level 4 table itself. By doing this, we effectively reserve a part of the virtual address space and map all current and future page table frames to that space.

Let's go through an example to understand how this all works:



Level 4 Page Table

Index	Frame	Flags
0	-	-
1	16KiB	r/w
...		
511	4KiB	r/w

Level 3 Page Table

Index	Frame	Flags
0	24KiB	r/w
...		

Level 2 Page Table

Index	Frame	Flags
...		
511	32KiB	r/w

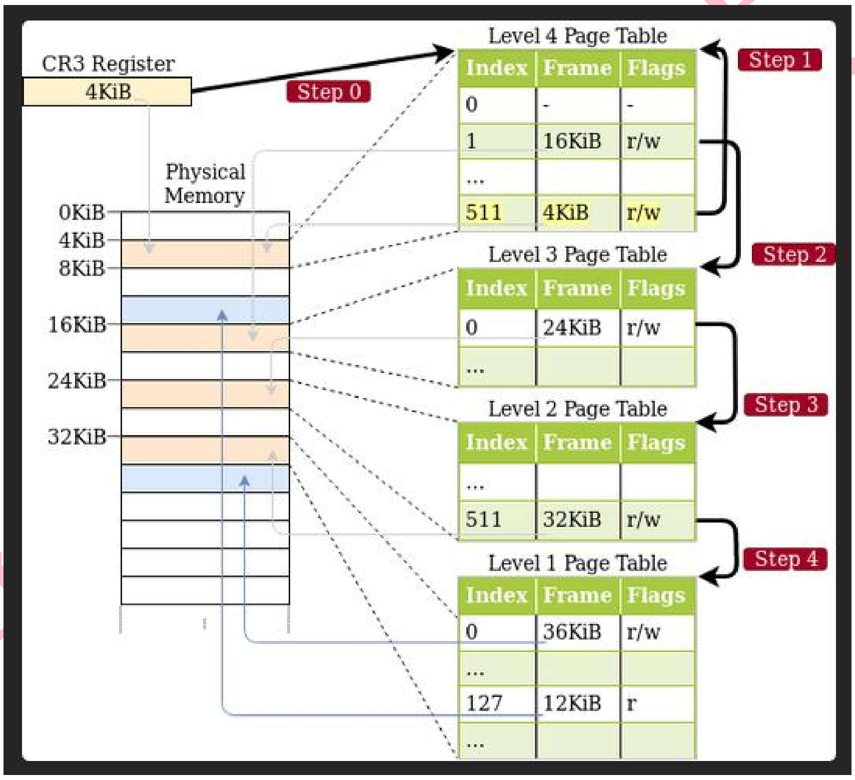
Level 1 Page Table

Index	Frame	Flags
0	36KiB	r/w
...		
127	12KiB	r
...		

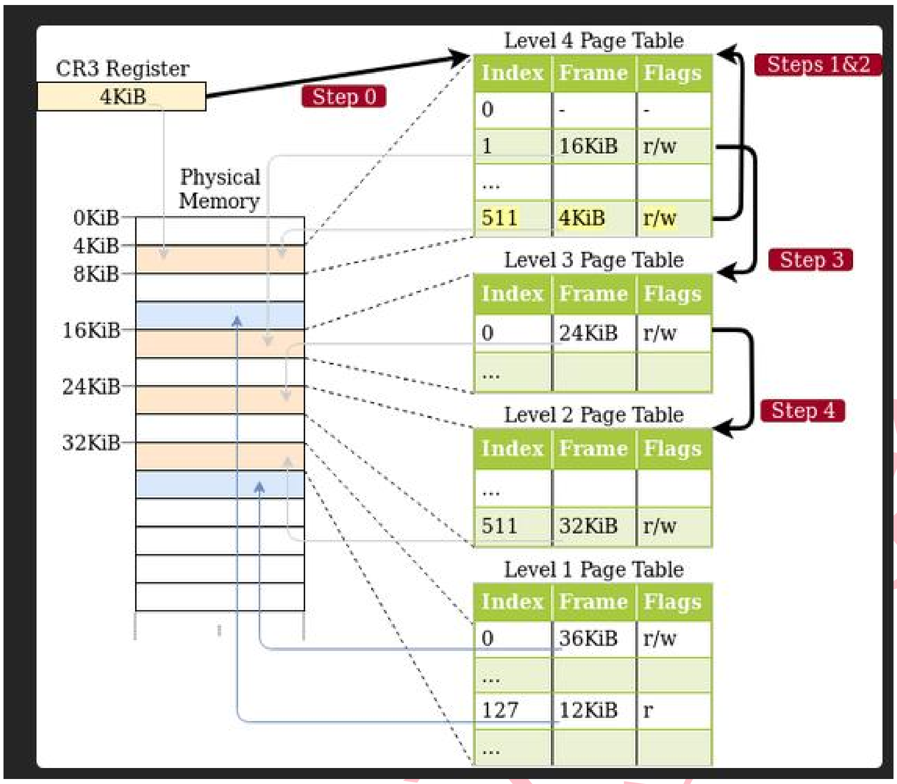
The only difference to the example at the beginning of this post is the additional entry at index 511 in the level 4 table, which is mapped to physical frame 4 KiB, the frame of the level 4 table itself.

By letting the CPU follow this entry on a translation, it doesn't reach a level 3 table but the same level 4 table again. This is similar to a recursive function that calls itself, therefore this table is called a recursive page table. The important thing is that the CPU assumes that every entry in the level 4 table points to a level 3 table, so it now treats the level 4 table as a level 3 table. This works because tables of all levels have the exact same layout on x86_64.

By following the recursive entry one or multiple times before we start the actual translation, we can effectively shorten the number of levels that the CPU traverses. For example, if we follow the recursive entry once and then proceed to the level 3 table, the CPU thinks that the level 3 table is a level 2 table. Going further, it treats the level 2 table as a level 1 table and the level 1 table as the mapped frame. This means that we can now read and write the level 1 page table because the CPU thinks that it is the mapped frame. The graphic below illustrates the five translation steps:

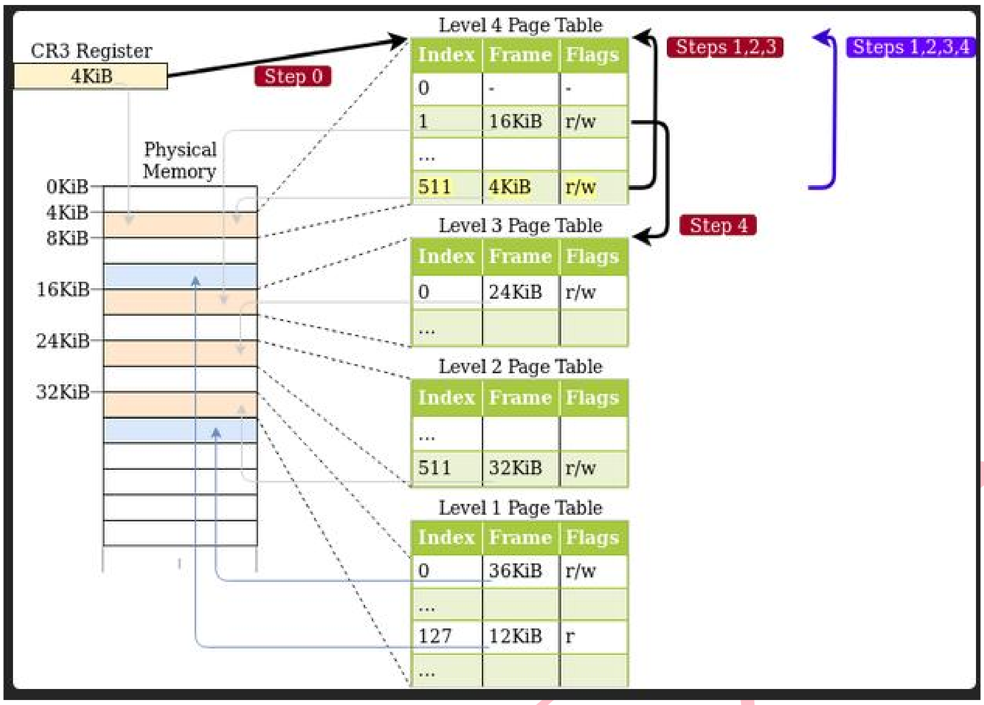


Similarly, we can follow the recursive entry twice before starting the translation to reduce the number of traversed levels to two:



Let's go through it step by step: First, the CPU follows the recursive entry on the level 4 table and thinks that it reaches a level 3 table. Then it follows the recursive entry again and thinks that it reaches a level 2 table. But in reality, it is still on the level 4 table. When the CPU now follows a different entry, it lands on a level 3 table but thinks it is already on a level 1 table. So while the next entry points to a level 2 table, the CPU thinks that it points to the mapped frame, which allows us to read and write the level 2 table.

Accessing the tables of levels 3 and 4 works in the same way. To access the level 3 table, we follow the recursive entry three times, tricking the CPU into thinking it is already on a level 1 table. Then we follow another entry and reach a level 3 table, which the CPU treats as a mapped frame. For accessing the level 4 table itself, we just follow the recursive entry four times until the CPU treats the level 4 table itself as the mapped frame (in blue in the graphic below).



It might take some time to wrap your head around the concept, but it works quite well in practice.

In the section below, we explain how to construct virtual addresses for following the recursive entry one or multiple times. We will not use recursive paging for our implementation, so you don't need to read it to continue with the post. If it interests you, just click on "Address Calculation" to expand it.

Recursive Paging is an interesting technique that shows how powerful a single mapping in a page table can be. It is relatively easy to implement and only requires a minimal amount of setup (just a single recursive entry), so it's a good choice for first experiments with paging.

However, it also has some disadvantages:

It occupies a large amount of virtual memory (512 GiB). This isn't a big problem in the large 48-bit address space, but it might lead to suboptimal cache behavior.

It only allows accessing the currently active address space easily. Accessing other address spaces is still possible by changing the recursive entry, but a temporary mapping is required for switching back. We described how to do this in the (outdated) Remap The Kernel post.

It heavily relies on the page table format of x86 and might not work on other architectures.

2. Bootloader Support

All of these approaches require page table modifications for their setup. For example, mappings for the physical memory need to be created or an entry of the level 4 table needs to be mapped recursively. The problem is that we can’t create these required mappings without an existing way to access the page tables.

This means that we need the help of the bootloader, which creates the page tables that our kernel runs on. The bootloader has access to the page tables, so it can create any mappings that we need. In its current implementation, the bootloader crate has support for two of the above approaches, controlled through cargo features:

The map_physical_memory feature maps the complete physical memory somewhere into the virtual address space. Thus, the kernel has access to all physical memory and can follow the Map the Complete Physical Memory approach.

With the recursive_page_table feature, the bootloader maps an entry of the level 4 page table recursively. This allows the kernel to access the page tables as described in the Recursive Page Tables section.

We choose the first approach for our kernel since it is simple, platform-independent, and more powerful (it also allows access to non-page-table-frames). To enable the required bootloader support, we add the map_physical_memory feature to our bootloader dependency:

```
[dependencies]
bootloader = { version = "0.9.23",features = ["map_physical_memory"]}
```

With this feature enabled, the bootloader maps the complete physical memory to some unused virtual address range. To communicate the virtual address range to our kernel, the bootloader passes a boot information structure.

2.1 Boot Information

The bootloader crate defines a BootInfo struct that contains all the information it passes to our kernel. The struct is still in an early stage, so expect some breakage when updating to future semver-incompatible bootloader versions. With the map_physical_memory feature enabled, it currently has the two fields memory_map and physical_memory_offset:

The memory_map field contains an overview of the available physical memory. This tells our kernel how much physical memory is available in the system and which memory regions are reserved for devices such as the VGA hardware. The memory map can be queried from the BIOS or UEFI firmware, but only very early in the boot process. For this reason, it must be provided by the bootloader because there is no way for the kernel to retrieve it later. We will need the memory map later in this post.

The physical_memory_offset tells us the virtual start address of the physical memory mapping. By adding this offset to a physical address, we get the corresponding virtual address. This allows us to access arbitrary physical memory from our kernel.

This physical memory offset can be customized by adding a [package.metadata.bootloader] table in Cargo.toml and setting the field physical-memory-offset = "0x0000f00000000000" (or any other value). However, note that the bootloader can panic if it runs into physical address values that start to overlap with the the space beyond the offset, i.e., areas it would have previously mapped to some other early physical addresses. So in general, the higher the value (> 1 TiB), the better.

The bootloader passes the BootInfo struct to our kernel in the form of a &'static BootInfo argument to our _start function. We don't have this argument declared in our function yet, so let's add it:

```
// in src/main.rs
use bootloader::BootInfo;
#[no_mangle]
pub extern "C" fn _start(boot_info: &'static BootInfo) -> ! {// new argument
    […]
}
```

It wasn't a problem to leave off this argument before because the x86_64 calling convention passes the first argument in a CPU register. Thus, the argument is simply ignored when it isn't declared. However, it would be a problem if we accidentally used a wrong argument type, since the compiler doesn't know the correct type signature of our entry point function.

2.2 The entry_point Macro

Since our _start function is called externally from the bootloader, no checking of our function signature occurs. This means that we could let it take arbitrary arguments without any compilation errors, but it would fail or cause undefined behavior at runtime.

To make sure that the entry point function always has the correct signature that the bootloader expects, the bootloader crate provides an entry_point macro that provides a type-checked way to define a Rust function as the entry point. Let's rewrite our entry point function to use this macro:

```
// in src/main.rs
use bootloader::{BootInfo, entry_point};
entry_point!(kernel_main);
fn kernel_main(boot_info: &'static BootInfo) -> ! {
    […]
}
```

We no longer need to use extern "C" or no_mangle for our entry point, as the macro defines the real lower level _start entry point for us. The kernel_main function is now a completely normal Rust function, so we can choose an arbitrary name for it. The important thing is that it is type-checked so that a compilation error occurs when we use a wrong function signature, for example by adding an argument or changing the argument type.

Let's perform the same change in our lib.rs:

```
// in src/lib.rs
#[cfg(test)]
use bootloader::{entry_point, BootInfo};
#[cfg(test)]
entry_point!(test_kernel_main);
/// Entry point for `cargo test`
#[cfg(test)]
fn test_kernel_main(_boot_info: &'static BootInfo) -> ! {
    // like before
    init();
    test_main();
    hlt_loop();
}
```

Since the entry point is only used in test mode, we add the #[cfg(test)] attribute to all items. We give our test entry point the distinct name test_kernel_main to avoid confusion with the kernel_main of our main.rs. We don’t use the BootInfo parameter for now, so we prefix the parameter name with a _ to silence the unused variable warning.

3. Implementation

Now that we have access to physical memory, we can finally start to implement our page table code. First, we will take a look at the currently active page tables that our kernel runs on. In the second step, we will create a translation function that returns the physical address that a given virtual address is mapped to. As a last step, we will try to modify the page tables in order to create a new mapping.

Before we begin, we create a new memory module for our code:

```
// in src/lib.rs
pub mod memory;
```

For the module, we create an empty src/memory.rs file.

3.1 Accessing the Page Tables

At the end of the previous post, we tried to take a look at the page tables our kernel runs on, but failed since we couldn't access the physical frame that the CR3 register points to. We're now able to continue from there by creating an active_level_4_table function that returns a reference to the active level 4 page table:

```
// in src/memory.rs
use x86_64::{
        structures::paging::PageTable,
        VirtAddr,
};
/// 返回一个对活动的 4 级表的可变引用。
/// 这个函数是不安全的，因为调用者必须保证完整的物理内存在传递的
/// `physical_memory_offset`处被映射到虚拟内存。另外，这个函数
/// 必须只被调用一次，以避免别名"&mut "引用（这是未定义的行为）。
pub unsafe fn active_level_4_table(physical_memory_offset: VirtAddr)
        -> &'static mut PageTable
{
        use x86_64::registers::control::Cr3;
        let (level_4_table_frame, _) = Cr3::read();
        let phys = level_4_table_frame.start_address();
        let virt = physical_memory_offset + phys.as_u64();
        let page_table_ptr: *mut PageTable = virt.as_mut_ptr();
        &mut *page_table_ptr // unsafe
}
```

First, we read the physical frame of the active level 4 table from the CR3 register. We then take its physical start address, convert it to a u64, and add it to physical_memory_offset to get the virtual address where the page table frame is mapped. Finally, we convert the virtual address to a *mut PageTable raw pointer through the as_mut_ptr method and then unsafely create a &mut PageTable reference from it. We create a &mut reference instead of a & reference because we will mutate the page tables later in this post.

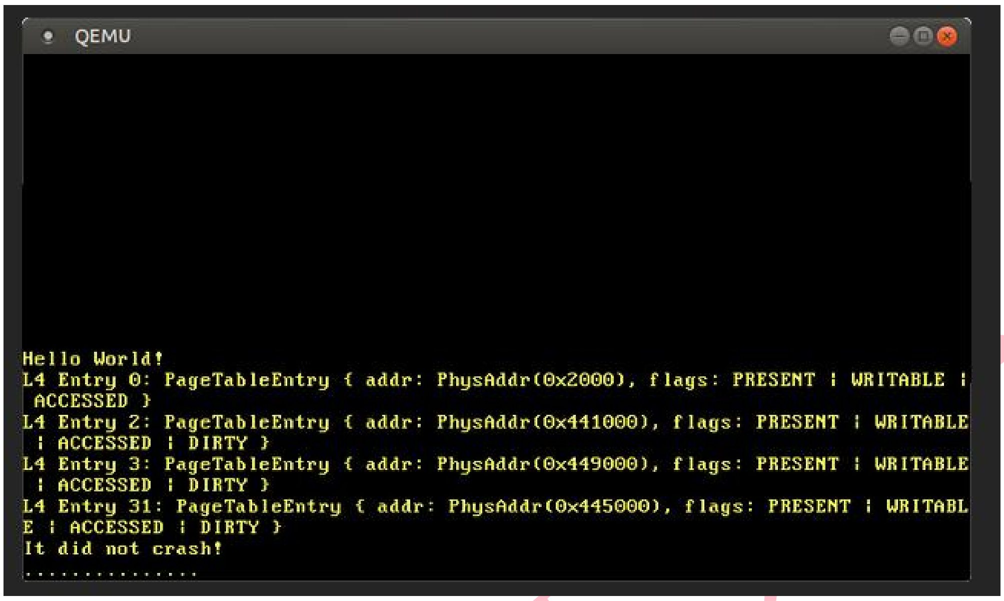
We don't need to use an unsafe block here because Rust treats the complete body of an unsafe fn like a large unsafe block. This makes our code more dangerous since we could accidentally introduce an unsafe operation in previous lines without noticing. It also makes it much more difficult to spot unsafe operations in between safe operations. There is an RFC to change this behavior.

We can now use this function to print the entries of the level 4 table:

```
// in src/main.rs
fn kernel_main(boot_info: &'static BootInfo) -> ! {
        use blog_os::memory::active_level_4_table;
        use x86_64::VirtAddr;
        println!("Hello World{}", "!");
        blog_os::init();
        let phys_mem_offset = VirtAddr::new(boot_info.physical_memory_offset);
        let l4_table = unsafe { active_level_4_table(phys_mem_offset) };
        for (i, entry) in l4_table.iter().enumerate() {
                if !entry.is_unused() {
                        println!("L4 Entry {}: {:?}", i, entry);
                }
        }
        // as before
        #[cfg(test)]
        test_main();
        println!("It did not crash!");
        blog_os::hlt_loop();
}
```

First, we convert the physical_memory_offset of the BootInfo struct to a VirtAddr and pass it to the active_level_4_table function. We then use the iter function to iterate over the page table entries and the enumerate combinator to additionally add an index i to each element. We only print non-empty entries because all 512 entries wouldn't fit on the screen.

When we run it, we see the following output:



```
Hello World!
L4 Entry 0: PageTableEntry { addr: PhysAddr(0x2000), flags: PRESENT | WRITABLE |
 ACCESSED }
L4 Entry 2: PageTableEntry { addr: PhysAddr(0x441000), flags: PRESENT | WRITABLE
 | ACCESSED | DIRTY }
L4 Entry 3: PageTableEntry { addr: PhysAddr(0x449000), flags: PRESENT | WRITABLE
 | ACCESSED | DIRTY }
L4 Entry 31: PageTableEntry { addr: PhysAddr(0x445000), flags: PRESENT | WRITABL
E | ACCESSED | DIRTY }
It did not crash!
...............
```

We see that there are various non-empty entries, which all map to different level 3 tables. There are so many regions because kernel code, kernel stack, physical memory mapping, and boot information all use separate memory areas.

To traverse the page tables further and take a look at a level 3 table, we can take the mapped frame of an entry and convert it to a virtual address again:

```
// in the `for` loop in src/main.rs
use x86_64::structures::paging::PageTable;
if !entry.is_unused() {
    println!("L4 Entry {}: {:?}", i, entry);
    // get the physical address from the entry and convert it
    let phys = entry.frame().unwrap().start_address();
    let virt = phys.as_u64() + boot_info.physical_memory_offset;
    let ptr = VirtAddr::new(virt).as_mut_ptr();
    let l3_table: &PageTable = unsafe { &*ptr };
    // print non-empty entries of the level 3 table
    for (i, entry) in l3_table.iter().enumerate() {
        if !entry.is_unused() {
            println!("    L3 Entry {}: {:?}", i, entry);
        }
    }
}
```

For looking at the level 2 and level 1 tables, we repeat that process for the level 3 and level 2 entries. As you can imagine, this gets very verbose very quickly, so we don’t show the full code here.

Traversing the page tables manually is interesting because it helps to understand how the CPU performs the translation. However, most of the time, we are only interested in the mapped physical address for a given virtual address, so let’s create a function for that.

3.2 Translating Addresses

To translate a virtual to a physical address, we have to traverse the four-level page table until we reach the mapped frame. Let’s create a function that performs this translation:

```
// in src/memory.rs
use x86_64::PhysAddr;
/// 将给定的虚拟地址转换为映射的物理地址，如果地址没有被映射，则为`None'。
/// 这个函数是不安全的，因为调用者必须保证完整的物理内存在
/// 传递的`physical_memory_offset`处被映射到虚拟内存。
pub unsafe fn translate_addr(addr: VirtAddr, physical_memory_offset: VirtAddr)
    -> Option<PhysAddr>
{
    translate_addr_inner(addr, physical_memory_offset)
}
```

We forward the function to a safe translate_addr_inner function to limit the scope of unsafe. As we noted above, Rust treats the complete body of an unsafe fn like a large unsafe block. By calling into a private safe function, we make each unsafe operation explicit again.

The private inner function contains the real implementation:

```
// in src/memory.rs
/// 由 `translate_addr`调用的私有函数。
/// 这个函数是安全的，可以限制`unsafe`的范围，
/// 因为 Rust 将不安全函数的整个主体视为不安全块。
/// 这个函数只能通过`unsafe fn`从这个模块的外部到达。
fn translate_addr_inner(addr: VirtAddr, physical_memory_offset: VirtAddr)
    -> Option<PhysAddr>
{
    use x86_64::structures::paging::page_table::FrameError;
    use x86_64::registers::control::Cr3;
    // 从 CR3 寄存器中读取活动的 4 级 frame
    let (level_4_table_frame, _) = Cr3::read();
    let table_indexes = [
        addr.p4_index(), addr.p3_index(), addr.p2_index(), addr.p1_index()
    ];
    let mut frame = level_4_table_frame;
    // 遍历多级页表
    for &index in &table_indexes {
        // 将该框架转换为页表参考
        let virt = physical_memory_offset + frame.start_address().as_u64();
        let table_ptr: *const PageTable = virt.as_ptr();
        let table = unsafe {&*table_ptr};
        // 读取页表条目并更新`frame`。
        let entry = &table[index];
        frame = match entry.frame() {
            Ok(frame) => frame,
            Err(FrameError::FrameNotPresent) => return None,
            Err(FrameError::HugeFrame) => panic!("huge pages not supported"),
        };
    }
    // 通过添加页面偏移量来计算物理地址
    Some(frame.start_address() + u64::from(addr.page_offset()))
}
```

Instead of reusing our active_level_4_table function, we read the level 4 frame from the CR3 register again. We do this because it simplifies this prototype implementation. Don't worry, we will create a better solution in a moment.

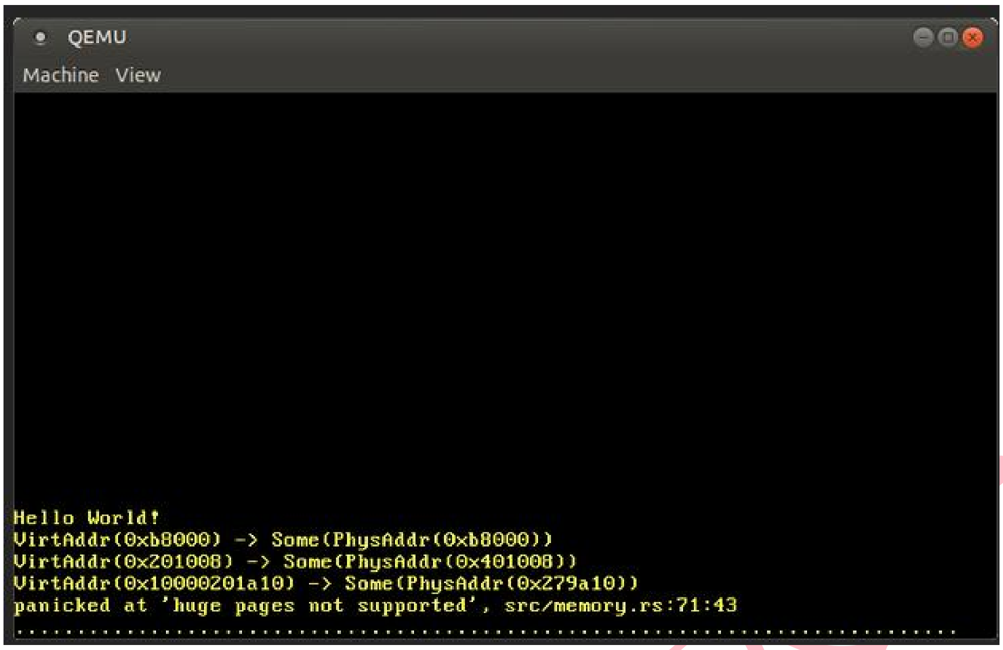
The VirtAddr struct already provides methods to compute the indexes into the page tables of the four levels. We store these indexes in a small array because it allows us to traverse the page tables using a for loop. Outside of the loop, we remember the last visited frame to calculate the physical address later. The frame points to page table frames while iterating and to the mapped frame after the last iteration, i.e., after following the level 1 entry.

Inside the loop, we again use the physical_memory_offset to convert the frame into a page table reference. We then read the entry of the current page table and use the PageTableEntry::frame function to retrieve the mapped frame. If the entry is not mapped to a frame, we return None. If the entry maps a huge 2 MiB or 1 GiB page, we panic for now.

Let’s test our translation function by translating some addresses:

```
// in src/main.rs
fn kernel_main(boot_info: &'static BootInfo) -> ! {
        // new import
        use blog_os::memory::translate_addr;
        [⋯] // hello world and blog_os::init
        let phys_mem_offset = VirtAddr::new(boot_info.physical_memory_offset);
        let addresses = [
                // the identity-mapped vga buffer page
                0xb8000,
                // some code page
                0x201008,
                // some stack page
                0x0100_0020_1a10,
                // virtual address mapped to physical address 0
                boot_info.physical_memory_offset,
        ];
        for &address in &addresses {
                let virt = VirtAddr::new(address);
                let phys = unsafe { translate_addr(virt, phys_mem_offset) };
                println!("{:?} -> {:?}", virt, phys);
        }
        [⋯] // test_main(), "it did not crash" printing, and hlt_loop()
}
```

When we run it, we see the following output:



```
Hello World!
VirtAddr(0xb8000) -> Some(PhysAddr(0xb8000))
VirtAddr(0x201008) -> Some(PhysAddr(0x401008))
VirtAddr(0x10000201a10) -> Some(PhysAddr(0x279a10))
panicked at 'huge pages not supported', src/memory.rs:71:43
```

As expected, the identity-mapped address 0xb8000 translates to the same physical address. The code page and the stack page translate to some arbitrary physical addresses, which depend on how the bootloader created the initial mapping for our kernel. It's worth noting that the last 12 bits always stay the same after translation, which makes sense because these bits are the page offset and not part of the translation.

Since each physical address can be accessed by adding the physical_memory_offset, the translation of the physical_memory_offset address itself should point to physical address 0. However, the translation fails because the mapping uses huge pages for efficiency, which is not supported in our implementation yet.

3.3 Using OffsetPageTable

Translating virtual to physical addresses is a common task in an OS kernel, therefore the x86_64 crate provides an abstraction for it. The implementation already supports huge pages and several other page table functions apart from translate_addr, so we will use it in the following instead of adding huge page support to our own implementation.

At the basis of the abstraction are two traits that define various page table mapping functions:

The Mapper trait is generic over the page size and provides functions that operate on pages. Examples are translate_page, which translates a given page to a frame of the same size, and map_to, which creates a new mapping in the page table.

The Translate trait provides functions that work with multiple page sizes, such as translate_addr or the general translate.

The traits only define the interface, they don't provide any implementation. The x86_64 crate currently provides three types that implement the traits with different requirements. The OffsetPageTable type assumes that the complete physical memory is mapped to the virtual address space at some offset. The MappedPageTable is a bit more flexible: It only requires that each page table frame is mapped to the virtual address space at a calculable address. Finally, the RecursivePageTable type can be used to access page table frames through recursive page tables.

In our case, the bootloader maps the complete physical memory at a virtual address specified by the physical_memory_offset variable, so we can use the OffsetPageTable type. To initialize it, we create a new init function in our memory module:

```
use x86_64::structures::paging::OffsetPageTable;
///  初始化一个新的 OffsetPageTable。
///  这个函数是不安全的，因为调用者必须保证完整的物理内存在
///  传递的`physical_memory_offset`处被映射到虚拟内存。另
///  外，这个函数必须只被调用一次，以避免别名"&mut "引用（这是未定义的行为）。
pub unsafe fn init(physical_memory_offset: VirtAddr) -> OffsetPageTable<'static> {
        let level_4_table = active_level_4_table(physical_memory_offset);
        OffsetPageTable::new(level_4_table, physical_memory_offset)
}
//  私下进行
unsafe fn active_level_4_table(physical_memory_offset: VirtAddr)
        -> &'static mut PageTable
{…}
```

The function takes the physical_memory_offset as an argument and returns a new OffsetPageTable instance with a 'static lifetime. This means that the instance stays valid for the complete runtime of our kernel. In the function body, we first call the active_level_4_table function to retrieve a mutable reference to the level 4 page table. We then invoke the OffsetPageTable::new function with this reference. As the second parameter, the new function expects the virtual address at which the mapping of the physical memory starts, which is given in the physical_memory_offset variable.

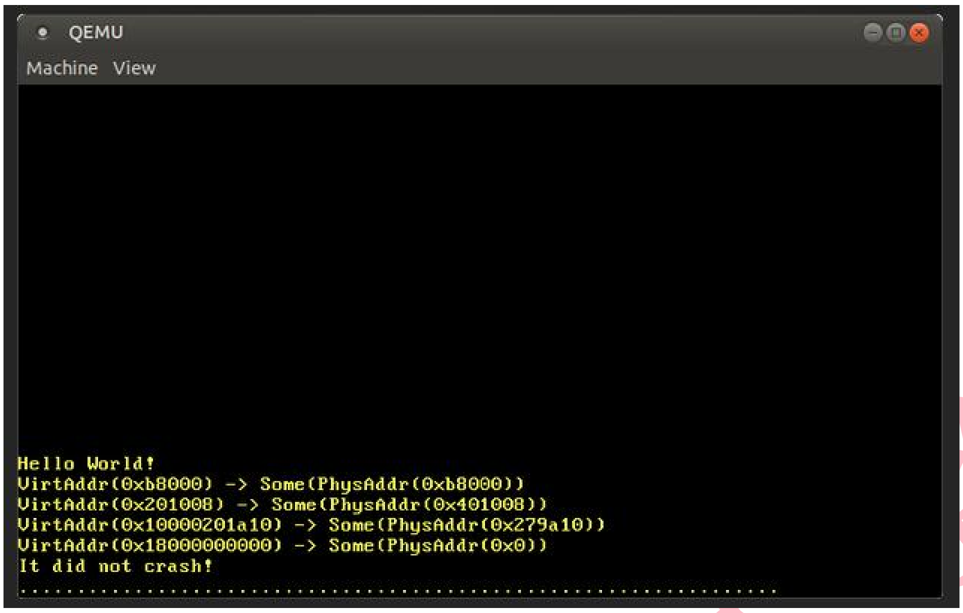
The active_level_4_table function should only be called from the init function from now on because it can easily lead to aliased mutable references when called multiple times, which can cause undefined behavior. For this reason, we make the function private by removing the pub specifier.

We can now use the Translate::translate_addr method instead of our own memory::translate_addr function. We only need to change a few lines in our kernel_main:

```
// in src/main.rs
fn kernel_main(boot_info: &'static BootInfo) -> ! {
        // new: different imports
        use blog_os::memory;
        use x86_64::{structures::paging::Translate, VirtAddr};
        […] // hello world and blog_os::init
        let phys_mem_offset = VirtAddr::new(boot_info.physical_memory_offset);
        // new: initialize a mapper
        let mapper = unsafe { memory::init(phys_mem_offset) };
        let addresses = […]; // same as before
        for &address in &addresses {
                let virt = VirtAddr::new(address);
                // new: use the `mapper.translate_addr` method
                let phys = mapper.translate_addr(virt);
                println!("{:?} -> {:?}", virt, phys);
        }
        […] // test_main(), "it did not crash" printing, and hlt_loop()
}
```

We need to import the Translate trait in order to use the translate_addr method it provides.

When we run it now, we see the same translation results as before, with the difference that the huge page translation now also works:



```
Hello World!
VirtAddr(0xb8000) -> Some(PhysAddr(0xb8000))
VirtAddr(0x201008) -> Some(PhysAddr(0x401008))
VirtAddr(0x10000201a10) -> Some(PhysAddr(0x279a10))
VirtAddr(0x18000000000) -> Some(PhysAddr(0x0))
It did not crash!
..................................................................
```

As expected, the translations of 0xb8000 and the code and stack addresses stay the same as with our own translation function. Additionally, we now see that the virtual address physical_memory_offset is mapped to the physical address 0x0.

By using the translation function of the MappedPageTable type, we can spare ourselves the work of implementing huge page support. We also have access to other page functions, such as map_to, which we will use in the next section.

At this point, we no longer need our memory::translate_addr and memory::translate_addr_inner functions, so we can delete them.

3.4 Creating a new Mapping

Until now, we only looked at the page tables without modifying anything. Let's change that by creating a new mapping for a previously unmapped page.

We will use the map_to function of the Mapper trait for our implementation, so let's take a look at that function first. The documentation tells us that it takes four arguments: the page that we want to map, the frame that the page should be mapped to, a set of flags for the page table entry, and a frame_allocator. The frame allocator is needed because mapping the given page might require creating additional page tables, which need unused frames as backing storage.

3.4.1 A create_example_mapping Function

The first step of our implementation is to create a new create_example_mapping function that maps a given virtual page to 0xb8000, the physical frame of the VGA text buffer. We choose that frame because it allows us to easily test if the mapping was created correctly: We just need to write to the newly mapped page and see whether we see the write appear on the screen.

The create_example_mapping function looks like this:

```
// in src/memory.rs
use x86_64::{
    PhysAddr,
    structures::paging::{Page, PhysFrame, Mapper, Size4KiB, FrameAllocator}
};
/// 为给定的页面创建一个实例映射到框架`0xb8000`。
pub fn create_example_mapping(
    page: Page,
    mapper: &mut OffsetPageTable,
    frame_allocator: &mut impl FrameAllocator<Size4KiB>,
) {
    use x86_64::structures::paging::PageTableFlags as Flags;
    let frame = PhysFrame::containing_address(PhysAddr::new(0xb8000));
    let flags = Flags::PRESENT | Flags::WRITABLE;
    let map_to_result = unsafe {
        // FIXME: 这并不安全，我们这样做只是为了测试。
        mapper.map_to(page, frame, flags, frame_allocator)
    };
    map_to_result.expect("map_to failed").flush();
}
```

In addition to the page that should be mapped, the function expects a mutable reference to an OffsetPageTable instance and a frame_allocator. The frame_allocator parameter uses the impl Trait syntax to be generic over all types that implement the FrameAllocator trait. The trait is generic over the PageSize trait to work with both standard 4 KiB pages and huge 2 MiB/1 GiB pages. We only want to create a 4 KiB mapping, so we set the generic parameter to Size4KiB.

The map_to method is unsafe because the caller must ensure that the frame is not already in use. The reason for this is that mapping the same frame twice could result in undefined behavior, for example when two different &mut references point to the same physical memory location. In our case, we reuse the VGA text buffer frame, which is already mapped, so we break the required condition. However, the create_example_mapping function is only a temporary testing function and will be removed after this post, so it is ok. To remind us of the unsafety, we put a FIXME comment on the line.

In addition to the page and the unused_frame, the map_to method takes a set of flags for the mapping and a reference to the frame_allocator, which will be explained in a moment. For the flags, we set the PRESENT flag because it is required for all valid entries and the WRITABLE flag to make the mapped page writable. For a list of all possible flags, see the Page Table Format section of the previous post.

The map_to function can fail, so it returns a Result. Since this is just some example code that does not need to be robust, we just use expect to panic when an error occurs. On success, the function returns a MapperFlush type that provides an easy way to flush the newly mapped page from the translation lookaside buffer (TLB) with its flush method. Like Result, the type uses the #[must_use] attribute to emit a warning when we accidentally forget to use it.

3.4.2 A dummy FrameAllocator

To be able to call create_example_mapping, we need to create a type that implements the FrameAllocator trait first. As noted above, the trait is responsible for allocating frames for new page tables if they are needed by map_to.

Let’s start with the simple case and assume that we don’t need to create new page tables. For this case, a frame allocator that always returns None suffices. We create such an EmptyFrameAllocator for testing our mapping function:

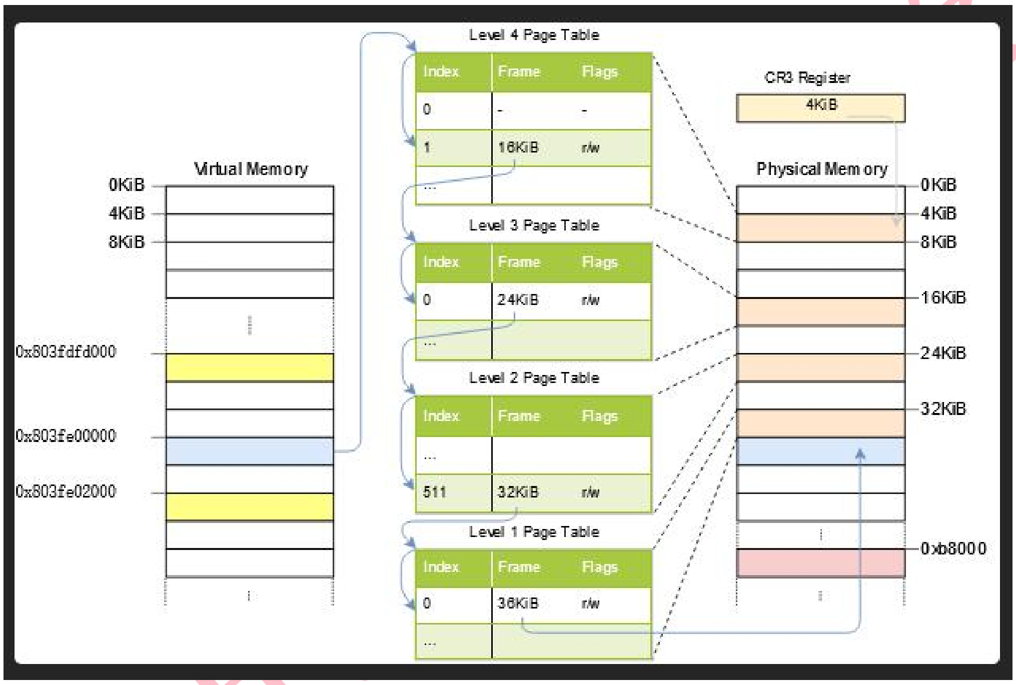
```
// in src/memory.rs
///  一个总是返回`None'的 FrameAllocator。
pub struct EmptyFrameAllocator;
unsafe impl FrameAllocator<Size4KiB> for EmptyFrameAllocator {
        fn allocate_frame(&mut self) -> Option<PhysFrame> {
                None
        }
}
```

Implementing the FrameAllocator is unsafe because the implementer must guarantee that the allocator yields only unused frames. Otherwise, undefined behavior might occur, for example when two virtual pages are mapped to the same physical frame. Our

EmptyFrameAllocator only returns None, so this isn't a problem in this case.

3.4.3 Choosing a Virtual Page

We now have a simple frame allocator that we can pass to our create_example_mapping function. However, the allocator always returns None, so this will only work if no additional page table frames are needed for creating the mapping. To understand when additional page table frames are needed and when not, let's consider an example:



The graphic shows the virtual address space on the left, the physical address space on the right, and the page tables in between. The page tables are stored in physical memory frames, indicated by the dashed lines. The virtual address space contains a single mapped page at address 0x803fe00000, marked in blue. To translate this page to its frame, the CPU walks the 4-level page table until it reaches the frame at address 36 KiB.

Additionally, the graphic shows the physical frame of the VGA text buffer in red. Our goal is to map a previously unmapped virtual page to this frame using our create_example_mapping function. Since our EmptyFrameAllocator always returns None, we want to create the mapping so that no additional frames are needed from the allocator. This depends on the virtual page that we select for the mapping.

The graphic shows two candidate pages in the virtual address space, both marked in yellow. One page is at address 0x803fdfd000, which is 3 pages before the mapped page (in blue). While the level 4 and level 3 page table indices are the same as for the blue page, the level 2 and level 1 indices are different (see the previous post). The different index into the level 2 table means that a different level 1 table is used for this page. Since this level 1 table does not exist yet, we would need to create it if we chose that page for our example mapping, which would require an additional unused physical frame. In contrast, the second candidate page at address 0x803fe02000 does not have this problem because it uses the same level 1 page table as the blue page. Thus, all the required page tables already exist.

In summary, the difficulty of creating a new mapping depends on the virtual page that we want to map. In the easiest case, the level 1 page table for the page already exists and we just need to write a single entry. In the most difficult case, the page is in a memory region for which no level 3 exists yet, so we need to create new level 3, level 2 and level 1 page tables first.

For calling our create_example_mapping function with the EmptyFrameAllocator, we need to choose a page for which all page tables already exist. To find such a page, we can utilize the fact that the bootloader loads itself in the first megabyte of the virtual address space. This means that a valid level 1 table exists for all pages in this region. Thus, we can choose any unused page in this memory region for our example mapping, such as the page at address 0. Normally, this page should stay unused to guarantee that dereferencing a null pointer causes a page fault, so we know that the bootloader leaves it unmapped.

3.4.4 Creating the Mapping

We now have all the required parameters for calling our create_example_mapping function, so let’s modify our kernel_main function to map the page at virtual address 0. Since we map the page to the frame of the VGA text buffer, we should be able to write to the screen through it afterward. The implementation looks like this:

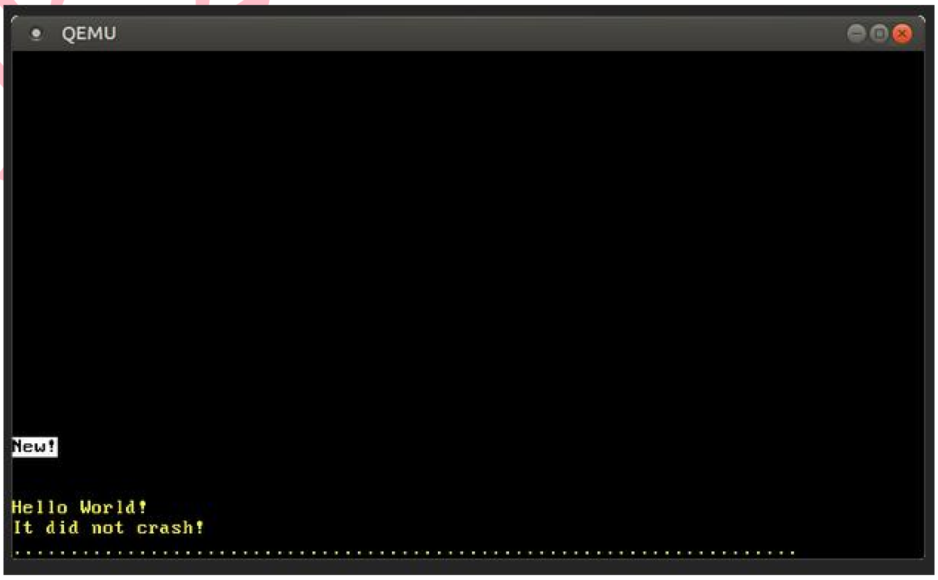
```
// in src/main.rs
fn kernel_main(boot_info: &'static BootInfo) -> ! {
        use blog_os::memory;
        use x86_64::{structures::paging::Page, VirtAddr}; //  新的导入
```

```
    […] // hello world and blog_os::init
    let phys_mem_offset = VirtAddr::new(boot_info.physical_memory_offset);
    let mut mapper = unsafe { memory::init(phys_mem_offset) };
    let mut frame_allocator = memory::EmptyFrameAllocator;
    // 映射未使用的页
    let page = Page::containing_address(VirtAddr::new(0));
    memory::create_example_mapping(page, &mut mapper, &mut frame_allocator);
    // 通过新的映射将字符串 `New!` 写到屏幕上。
    let page_ptr: *mut u64 = page.start_address().as_mut_ptr();
    unsafe { page_ptr.offset(400).write_volatile(0x_f021_f077_f065_f04e)};
    […] // test_main(), "it did not crash" printing, and hlt_loop()
}
```

We first create the mapping for the page at address 0 by calling our create_example_mapping function with a mutable reference to the mapper and the frame_allocator instances. This maps the page to the VGA text buffer frame, so we should see any write to it on the screen.

Then we convert the page to a raw pointer and write a value to offset 400. We don't write to the start of the page because the top line of the VGA buffer is directly shifted off the screen by the next println. We write the value 0x_f021_f077_f065_f04e, which represents the string "New!" on a white background. As we learned in the "VGA Text Mode" post, writes to the VGA buffer should be volatile, so we use the write_volatile method.

When we run it in QEMU, we see the following output:



```
New!


Hello World!
It did not crash!
.....................................................................
```

The “New!” on the screen is caused by our write to page 0, which means that we successfully created a new mapping in the page tables.

Creating that mapping only worked because the level 1 table responsible for the page at address 0 already exists. When we try to map a page for which no level 1 table exists yet, the map_to function fails because it tries to create new page tables by allocating frames with the EmptyFrameAllocator. We can see that happen when we try to map page 0xdeadbeaf000 instead of 0:

```
// in src/main.rs
fn kernel_main(boot_info: &'static BootInfo) -> ! {
        […]
        let page = Page::containing_address(VirtAddr::new(0xdeadbeaf000));
        […]
}
```

When we run it, a panic with the following error message occurs:

```
        panicked at 'map_to failed: FrameAllocationFailed', /…/result.rs:999:5
```

To map pages that don’t have a level 1 page table yet, we need to create a proper FrameAllocator. But how do we know which frames are unused and how much physical memory is available?

3.5 Allocating Frames

In order to create new page tables, we need to create a proper frame allocator. To do that, we use the memory_map that is passed by the bootloader as part of the BootInfo struct:

```
// in src/memory.rs
use bootloader::bootinfo::MemoryMap;
///  一个 FrameAllocator，从 bootloader 的内存地图中返回可用的  frames。
pub struct BootInfoFrameAllocator {
        memory_map: &'static MemoryMap,
        next: usize,
}
impl BootInfoFrameAllocator {
        ///  从传递的内存  map  中创建一个 FrameAllocator。
        ///
        ///  这个函数是不安全的，因为调用者必须保证传递的内存  map  是有效的。
        ///  主要的要求是，所有在其中被标记为 "可用 "的帧都是真正未使用的。
        pub unsafe fn init(memory_map: &'static MemoryMap) -> Self {
                BootInfoFrameAllocator {
                        memory_map,
                        next: 0,
                }
        }
}
```

The struct has two fields: A 'static reference to the memory map passed by the bootloader and a next field that keeps track of the number of the next frame that the allocator should return.

As we explained in the Boot Information section, the memory map is provided by the BIOS/UEFI firmware. It can only be queried very early in the boot process, so the bootloader already calls the respective functions for us. The memory map consists of a list of MemoryRegion structs, which contain the start address, the length, and the type (e.g. unused, reserved, etc.) of each memory region.

The init function initializes a BootInfoFrameAllocator with a given memory map. The next field is initialized with 0 and will be increased for every frame allocation to avoid returning the same frame twice. Since we don’t know if the usable frames of the memory map were already used somewhere else, our init function must be unsafe to require additional guarantees from the caller.

3.5.1 A usable_frames Method

Before we implement the FrameAllocator trait, we add an auxiliary method that converts the memory map into an iterator of usable frames:

```
// in src/memory.rs
use bootloader::bootinfo::MemoryRegionType;
impl BootInfoFrameAllocator {
        /// 返回内存映射中指定的可用框架的迭代器。
        fn usable_frames(&self) -> impl Iterator<Item = PhysFrame> {
                // 从内存 map 中获取可用的区域
                let regions = self.memory_map.iter();
                let usable_regions = regions
                        .filter(|r| r.region_type == MemoryRegionType::Usable);
                // 将每个区域映射到其地址范围
                let addr_ranges = usable_regions
                        .map(|r| r.range.start_addr()..r.range.end_addr());
                // 转化为一个帧起始地址的迭代器
                let frame_addresses = addr_ranges.flat_map(|r| r.step_by(4096));
```

```
            // 从起始地址创建 `PhysFrame` 类型
            frame_addresses.map(|addr| PhysFrame::containing_address(PhysAddr::new(addr)))
        }
}
```

This function uses iterator combinator methods to transform the initial MemoryMap into an iterator of usable physical frames:

First, we call the iter method to convert the memory map to an iterator of MemoryRegions.

Then we use the filter method to skip any reserved or otherwise unavailable regions. The bootloader updates the memory map for all the mappings it creates, so frames that are used by our kernel (code, data, or stack) or to store the boot information are already marked as InUse or similar. Thus, we can be sure that Usable frames are not used somewhere else.

Afterwards, we use the map combinator and Rust's range syntax to transform our iterator of memory regions to an iterator of address ranges.

Next, we use flat_map to transform the address ranges into an iterator of frame start addresses, choosing every 4096th address using step_by. Since 4096 bytes (= 4 KiB) is the page size, we get the start address of each frame. The bootloader page-aligns all usable memory areas so that we don't need any alignment or rounding code here. By using flat_map instead of map, we get an Iterator instead of an Iterator>.

Finally, we convert the start addresses to PhysFrame types to construct an Iterator.

The return type of the function uses the impl Trait feature. This way, we can specify that we return some type that implements the Iterator trait with item type PhysFrame but don't need to name the concrete return type. This is important here because we can't name the concrete type since it depends on unnamable closure types.

3.5.2 Implementing the FrameAllocator Trait

Now we can implement the FrameAllocator trait:

```
// in src/memory.rs
unsafe impl FrameAllocator<Size4KiB> for BootInfoFrameAllocator {
    fn allocate_frame(&mut self) -> Option<PhysFrame> {
        let frame = self.usable_frames().nth(self.next);
        self.next += 1;
        frame
        }
}
```

We first use the usable_frames method to get an iterator of usable frames from the memory map. Then, we use the Iterator::nth function to get the frame with index self.next (thereby skipping (self.next - 1) frames). Before returning that frame, we increase self.next by one so that we return the following frame on the next call.

This implementation is not quite optimal since it recreates the usable_frame allocator on every allocation. It would be better to directly store the iterator as a struct field instead. Then we wouldn't need the nth method and could just call next on every allocation. The problem with this approach is that it's not possible to store an impl Trait type in a struct field currently. It might work someday when named existential types are fully implemented.

3.5.3 Using the BootInfoFrameAllocator

We can now modify our kernel_main function to pass a BootInfoFrameAllocator instance instead of an EmptyFrameAllocator:

```
// in src/main.rs
fn kernel_main(boot_info: &'static BootInfo) -> ! {
        use blog_os::memory::BootInfoFrameAllocator;
        […]
        let mut frame_allocator = unsafe {
                BootInfoFrameAllocator::init(&boot_info.memory_map)
        };
        […]
}
```

With the boot info frame allocator, the mapping succeeds and we see the black-on-white "New!" on the screen again. Behind the scenes, the map_to method creates the missing page tables in the following way:

Use the passed frame_allocator to allocate an unused frame.

Zero the frame to create a new, empty page table.

Map the entry of the higher level table to that frame.

Continue with the next table level.

While our create_example_mapping function is just some example code, we are now able to create new mappings for arbitrary pages. This will be essential for allocating memory or implementing multithreading in future posts.

At this point, we should delete the create_example_mapping function again to avoid accidentally invoking undefined behavior, as explained above.

4. Summary

In this post we learned about different techniques to access the physical frames of page tables, including identity mapping, mapping of the complete physical memory, temporary mapping, and recursive page tables. We chose to map the complete physical memory since it’s simple, portable, and powerful.

We can’t map the physical memory from our kernel without page table access, so we need support from the bootloader. The bootloader crate supports creating the required mapping through optional cargo crate features. It passes the required information to our kernel in the form of a &BootInfo argument to our entry point function.

For our implementation, we first manually traversed the page tables to implement a translation function, and then used the MappedPageTable type of the x86_64 crate. We also learned how to create new mappings in the page table and how to create the necessary FrameAllocator on top of the memory map passed by the bootloader.

外文译文内容：

分页实现

摘　要

这篇文章展示了如何在我们的内核中实现分页支持。 它首先探讨了使内核可以访问物理页表帧的不同技术，并讨论了它们各自的优缺点。 然后它实现地址转换功能和创建新映射的功能。

上一篇文章介绍了分页的概念。通过将分页与分段进行比较来证明分页的优势，解释了分页和页表的工作原理，然后介绍了 x86_64 的 4 级页表设计。 我们发现引导加载程序 bootloader 已经为我们的内核设置了页表层次结构，这意味着我们的内核已经在虚拟地址上运行。 这提高了安全性，因为非法内存访问会导致页面错误异常，而不是修改任意物理内存。

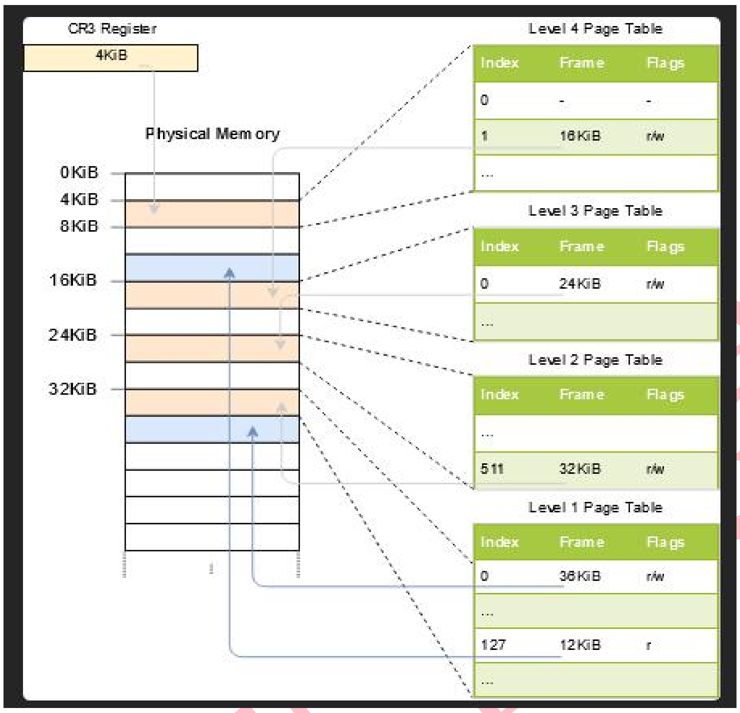
上篇文章最后说我们无法从内核访问页表，因为它们存储在物理内存中并且我们的内核已经在虚拟地址上运行。 这篇文章探讨了使我们的内核可以访问页表帧的不同方法。 我们将讨论每种方法的优点和缺点，最后决定我们的内核采用哪种方法。要实施该方法，我们需要引导加载程序的支持，因此我们将首先对其进行配置。 之后，我们将实现一个遍历页表层次结构的函数，以便将虚拟地址转换为物理地址。 最后，我们学习如何在页表中创建新的映射以及如何找到未使用的内存帧来创建新的页表。

关键词：分页支持；访问页表；地址转换

目　录

1. 访问页表

从内核中访问页表并不像它看起来那么容易。为了理解这个问题，让我们再看一下上一篇文章中的 4 级页表层次结构的例子。



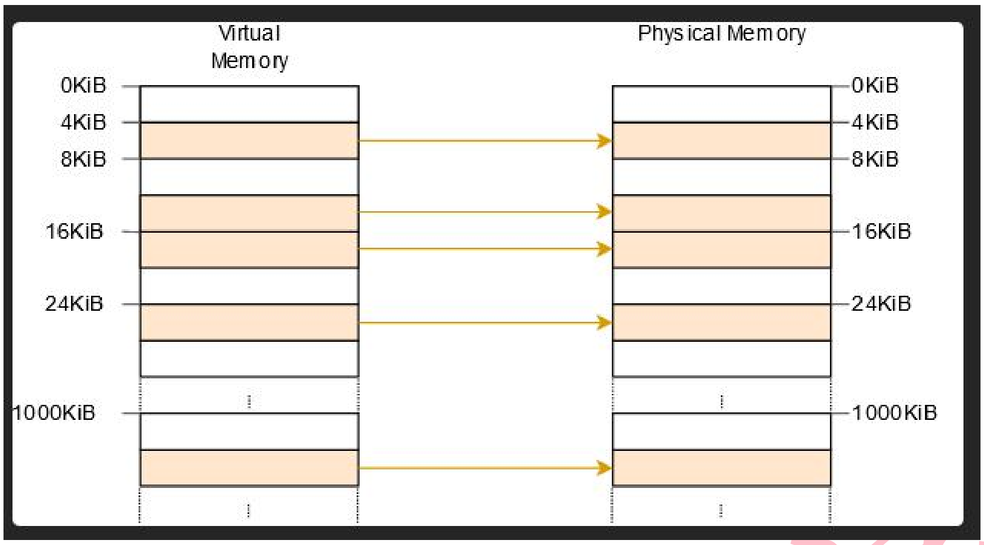
这里重要的是每个页面条目存储下一个表的物理地址。 这就避免了对这些地址也进行翻译，这对性能不利并且很容易导致无休止的翻译循环。

我们的问题是我们不能直接从内核访问物理地址，因为我们的内核也在虚拟地址之上运行。 例如，当我们访问地址 4KiB 时，我们访问的是虚拟地址 4KiB，而不是存储 4 级页表的物理地址 4KiB。 当我们想要访问物理地址 4KiB 时，我们只能通过一些映射到它的虚拟地址来访问。

所以为了访问页表框架，我们需要将一些虚拟页面映射到它们。 有不同的方法来创建这些映射，这些映射都允许我们访问任意页表框架。

一、 1.1 恒等映射

一个简单的解决方案是恒等映射所有页表：



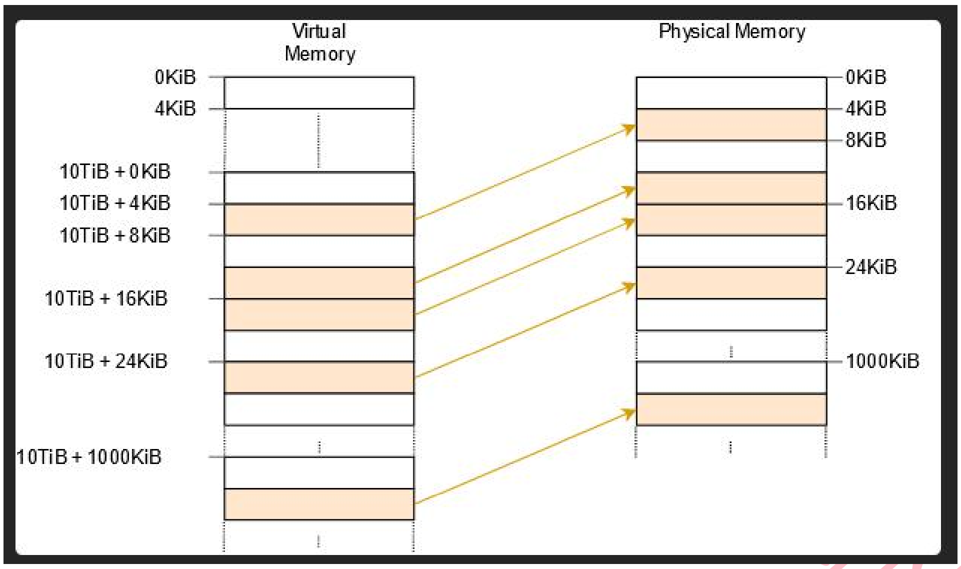
在这个例子中，我们看到了各种恒等映射的页表帧。 这样，页表的物理地址也是有效的虚拟地址，这样我们就可以方便地访问从 CR3 寄存器开始的各级页表。

但是，它会使虚拟地址空间变得混乱，并且更难找到更大尺寸的连续内存区域。 例如，假设我们要在上图中创建一个大小为 1000 KiB 的虚拟内存区域，例如，用于文件的内存映射。 我们不能以 28KiB 开始该区域，因为它会与 1004KiB 的已映射页面发生冲突。 所以我们必须进一步寻找，直到找到足够大的未映射区域，例如 1008KiB。 这是与分段类似的碎片问题。

同样，这使得创建新页表变得更加困难，因为我们需要找到其对应页尚未使用的物理帧。 例如，假设我们为内存映射文件保留了从 1008KiB 开始的 1000KiB 虚拟内存区域。现在我们不能再使用物理地址在 1000KiB 到 2008KiB 之间的任何帧，因为我们不能对它进行恒等映射。

1.2 以固定偏移量映射

为了避免虚拟地址空间混乱的问题，我们可以为页表映射使用一个单独的内存区域。 因此，我们不使用恒等映射页表框架，而是将它们映射到虚拟地址空间中的固定偏移量。 例如，偏移量可能是 10 TiB：

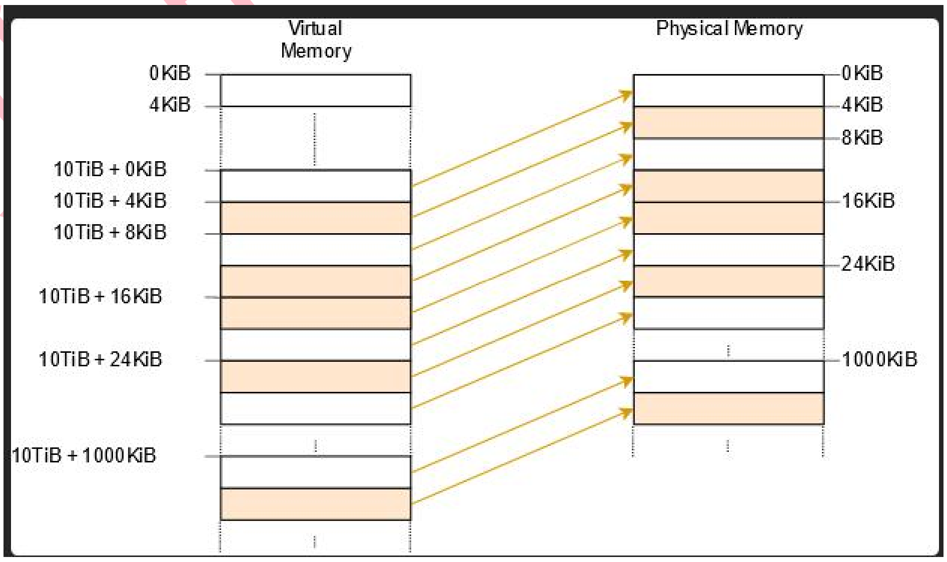


通过将 10 TiB..（10 TiB + 物理内存大小）范围内的虚拟内存专用于页表映射，我们避免了恒等映射的冲突问题。 仅当虚拟地址空间远大于物理内存大小时，才有可能保留如此大的虚拟地址空间区域。 这在 x86_64 上不是问题，因为 48 位地址空间有 256 TiB 大。

这种方法仍然有一个缺点，即我们需要在每次创建新页表时都创建一个新的映射。 此外，它不允许访问其他地址空间的页表，这在创建新进程时很有用。

二、 1.3 映射完整的物理内存

我们可以通过映射完整的物理内存而不是仅仅映射页表框架来解决这些问题：



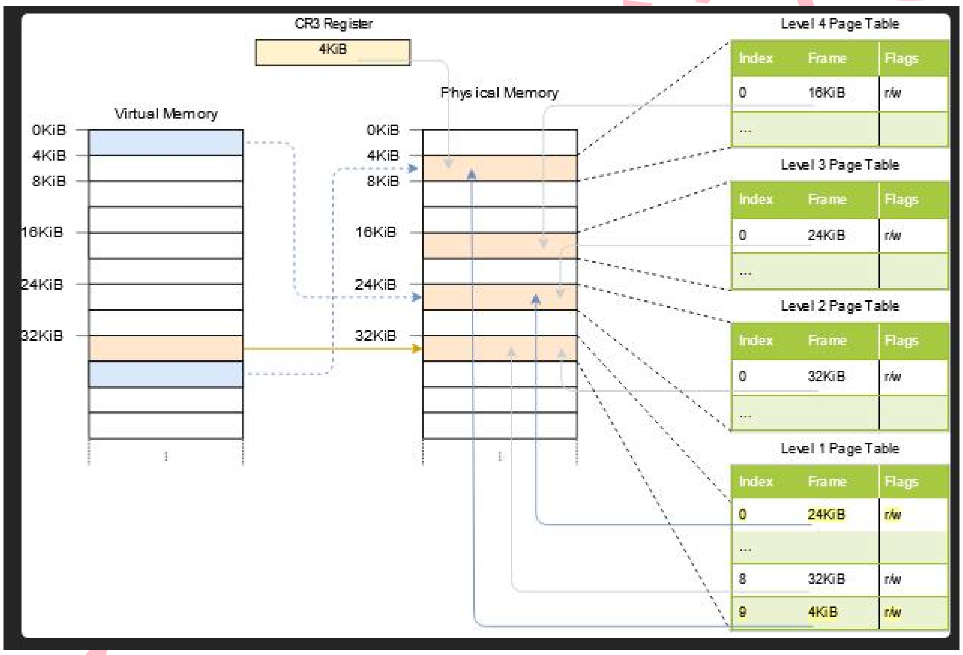
这种方法允许我们的内核访问任意物理内存，包括其他地址空间的页表帧。保留的虚拟内存范围与以前的大小相同，不同之处在于它不再包含未映射的页面。

这种方法的缺点是需要额外的页表来存储物理内存的映射。 这些页表需要存储在某个地方，因此它们会占用一部分物理内存，这在内存量较小的设备上可能会出现问题。

然而，在 x86_64 上，我们可以使用大小为 2 MiB 的大页面进行映射，而不是默认的 4 KiB 页面。 这样，映射 32 GiB 的物理内存只需要 132 KiB 用于页表，因为只需要一个 3 级表和 32 个 2 级表。 大页面的缓存效率也更高，因为它们在转换后备缓冲区 (TLB) 中使用的条目较少。

三、 1.4 临时映射

对于物理内存非常小的设备，我们可以仅在需要访问它们时临时映射页表框架。为了能够创建临时映射，我们只需要一个身份映射级别 1 表：



此图中的 1 级表控制虚拟地址空间的前 2 MiB。 这是因为它可以通过从 CR3 寄存器开始并跟随第 4 级、第 3 级和第 2 级页表中的第 0 个条目来访问。索引为 8 的条目将地址 32KiB 处的虚拟页映射到地址 32KiB 处的物理帧，从而标识映射 1 级表本身。 该图通过 32KiB 处的水平箭头显示了此标识映射。

通过写入身份映射一级表，我们的内核最多可以创建 511 个临时映射（512 减去身份映射所需的条目）。 在上面的例子中，内核创建了两个临时映射：

(1) 通过将 1 级表的第 0 个条目映射到地址为 24KiB 的帧，它创建了 0KiB 处的虚拟页

面到 2 级页表的物理帧的临时映射，如虚线箭头所示。

(2) 通过将第 1 级表的第 9 个条目映射到地址为 4KiB 的帧，它创建了 36KiB 的虚拟页面到第 4 级页表的物理帧的临时映射，如虚线箭头所示。

现在内核可以通过写入页面 0KiB 来访问 2 级页表，通过写入页面 36KiB 来访问 4 级页表。

使用临时映射访问任意页表框架的过程是：

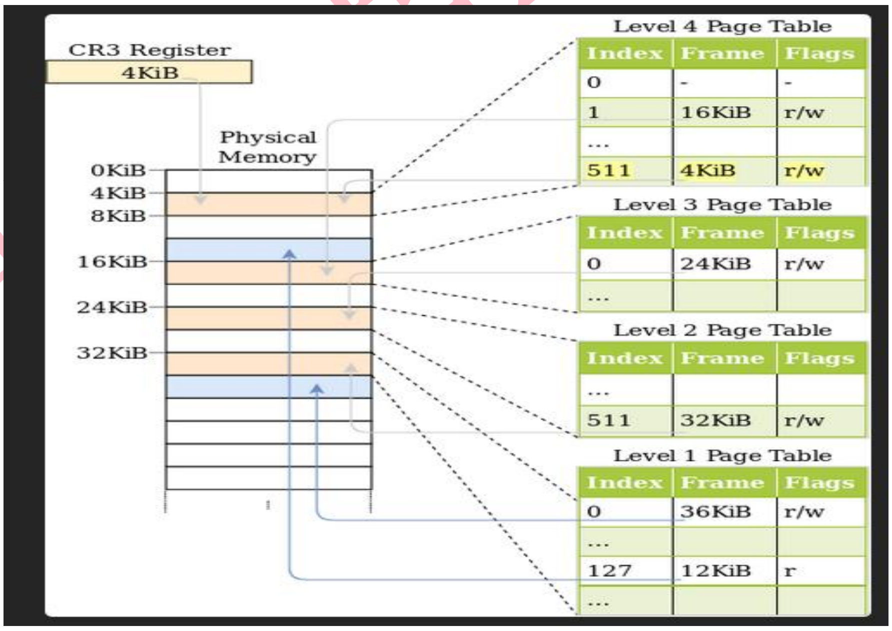
- 在标识映射级别 1 表中搜索一个空闲条目。
- 将该条目映射到我们要访问的页表的物理框架。
- 通过映射到条目的虚拟页面访问目标框架。
- 将条目设置回未使用状态，从而再次删除临时映射。

这种方法重复使用相同的 512 个虚拟页面来创建映射，因此只需要 4 KiB 的物理内存。 缺点是有点麻烦，尤其是一个新的映射可能需要修改多个表级别，这意味着我们需要多次重复上述过程。

四、 1.5 递归页表

另一种有趣的方法是根本不需要额外的页表，即映射页表的递归。这种方法背后思想是将一个条目从第 4 级页面表映射到第 4 级表本身。通过这样做，我们有效地保留了虚拟地址空间的一部分，并将所有当前和未来的页表框架映射到该空间。

让我们通过一个例子来了解这一切是如何进行的。



Level 4 Page Table

Index	Frame	Flags
0	-	-
1	16KiB	r/w
...		
511	4KiB	r/w

Level 3 Page Table

Index	Frame	Flags
0	24KiB	r/w
...		

Level 2 Page Table

Index	Frame	Flags
...		
511	32KiB	r/w

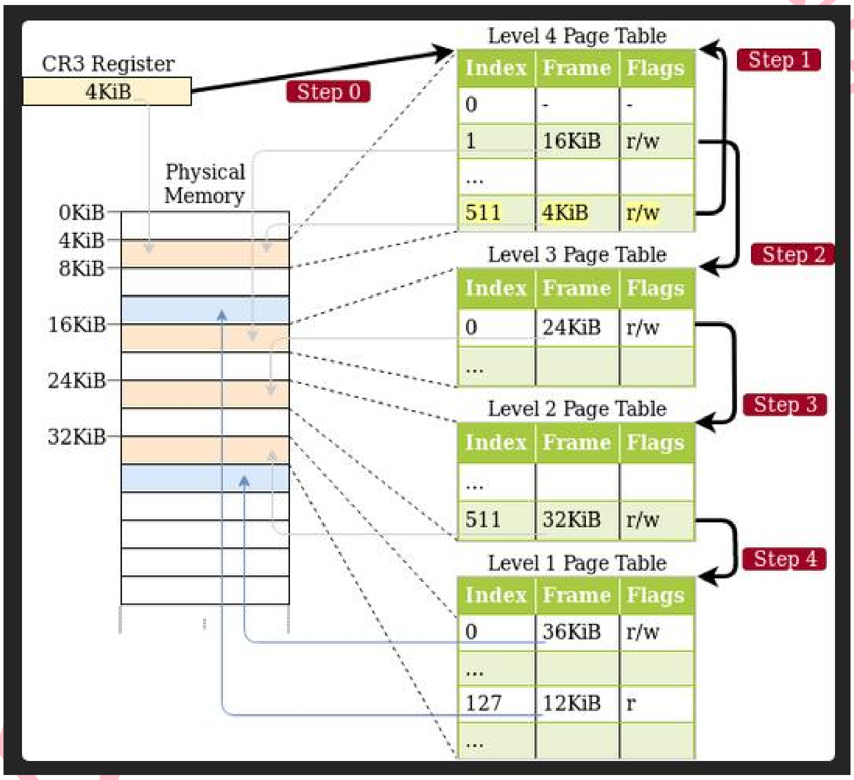
Level 1 Page Table

Index	Frame	Flags
0	36KiB	r/w
...		
127	12KiB	r
...		

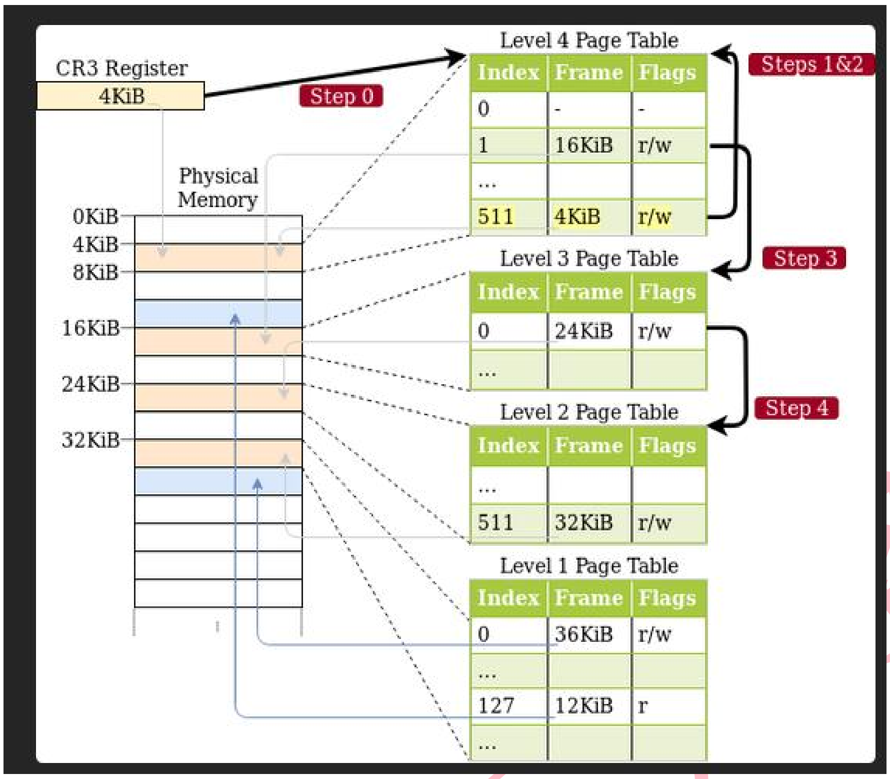
与本文开头的例子的唯一区别是在 4 级表中的索引 511 处增加了一个条目，它被映射到物理帧 4 KiB，即 4 级表本身的帧。

通过让 CPU 跟随这个条目进行翻译，它不会到达 3 级表，而是再次到达同一个 4 级表。这类似于一个调用自身的递归函数，因此这个表被称为 递归页表 。重要的是，CPU 假定 4 级表的每个条目都指向 3 级表，所以它现在把 4 级表当作 3 级表。这是因为所有级别的表在 x86_64 上都有完全相同的布局。

在我们开始实际翻译之前，通过跟随递归条目一次或多次，我们可以有效地缩短 CPU 所穿越的层数。例如，如果我们跟随递归条目一次，然后进入第 3 级表，CPU 会认为第 3 级表是第 2 级表。再往前走，它把第 2 级表当作第 1 级表，把第 1 级表当作映射的框架。这意味着我们现在可以读写第 1 级页表了，因为 CPU 认为它是映射的帧。下面的图形说明了这五个转换步骤。

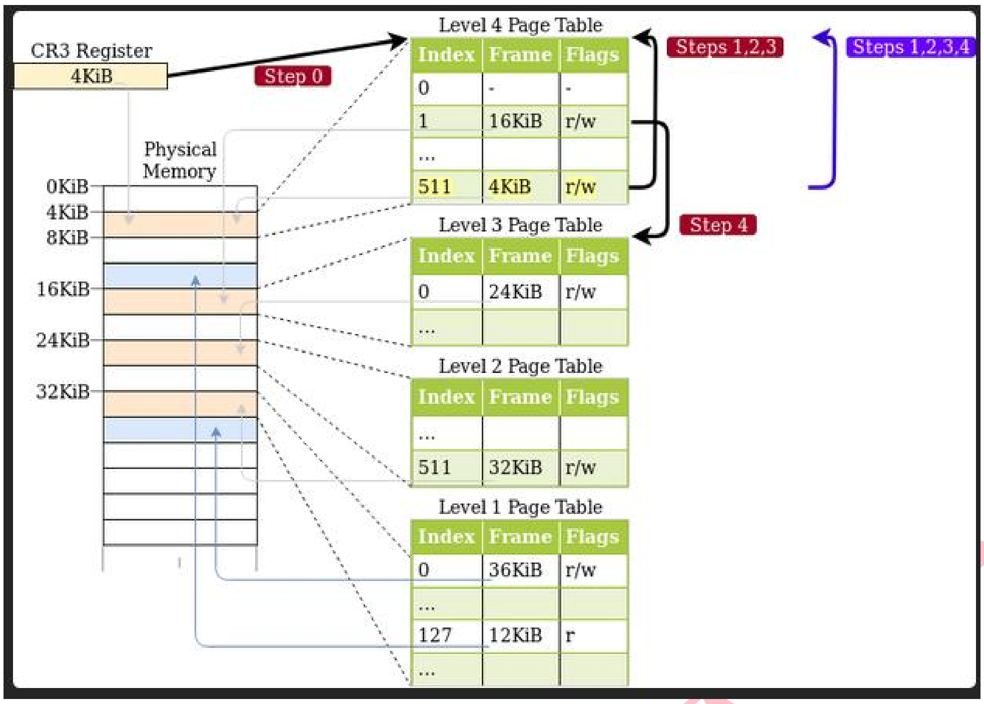


同样地，我们可以在开始翻译之前，先跟随递归条目两次，将遍历的层数减少到两个。



让我们一步一步地看下去。首先，CPU 跟踪 4 级表的递归条目，认为它到达了 3 级表。然后，它再次跟踪递归条目，认为它到达了 2 级表。但实际上，它仍然是在第 4 级表中。当 CPU 现在跟随一个不同的条目时，它到达了一个 3 级表，但认为它已经在 1 级表上。因此，当下一个条目指向第 2 级表时，CPU 认为它指向了映射的框架，这使得我们能够读写第 2 级表。

访问第 3 级和第 4 级表的方法是一样的。为了访问第 3 级表，我们沿着递归条目走了三次，诱使 CPU 认为它已经在第 1 级表上了。然后我们跟随另一个条目，到达第 3 级表，CPU 将其视为一个映射的框架。对于访问第 4 级表本身，我们只需跟随递归条目四次，直到 CPU 将第 4 级表本身视为映射的框架（在下面的图形中为蓝色）。



可能需要一些时间来理解这个概念，但在实践中效果相当好。

在下面的章节中，我们将解释如何构建虚拟地址，用于跟随递归条目一次或多次。在我们的实现中，我们不会使用递归分页，所以你不需要阅读它就可以继续阅读本帖。如果你感兴趣，只需点击 “地址计算” 来展开

递归分页是一种有趣的技术，它显示了页表中的单个映射可以有多么强大。它比较容易实现，而且只需要少量的设置（只是一个单一的递归条目），所以它是第一次实验分页的一个好选择。

然而，它也有一些弊端：

- 它占据了大量的虚拟内存（512 GiB）。在大的 48 位地址空间中，这不是一个大问题，但它可能会导致次优的缓存行为。
- 它只允许轻松访问当前活动的地址空间。通过改变递归条目，访问其他地址空间仍然是可能的，但切换回来时需要一个临时映射。我们在(已过期的)Remap The Kernel 文章“地址空间 “中描述了如何做到这一点。
- 它在很大程度上依赖于 x86 的页表格式，在其他架构上可能无法工作。

2. 支持引导加载程序

所有这些方法都需要为它们的设置修改页表。 例如，需要创建物理内存的映射，或者需要递归映射 4 级表的条目。 问题是，如果没有现有的访问页表的方法，我们就无法创建这些必需的映射。

这意味着我们需要引导加载程序的帮助，它创建我们的内核运行的页表。 引导加载程序可以访问页表，因此它可以创建我们需要的任何映射。 在当前的实现中，bootloader crate 支持上述两种方法，通过 cargo features 控制：

- map_physical_memory 功能将某处完整的物理内存映射到虚拟地址空间。因此，内核可以访问所有的物理内存，并且可以遵循映射完整物理内存的方法。
- 有了 “recursive_page_table “功能，bootloader 会递归地映射 4 级 page table 的一个条目。这允许内核访问页表，如递归页表部分所述。

我们为我们的内核选择了第一种方法，因为它很简单，与平台无关，而且更强大（它还允许访问非页表框架）。为了启用所需的引导程序支持，我们在 “引导程序 “的依赖中加入了 “map_physical_memory “功能。

```
[dependencies]
bootloader = { version = "0.9.23",features = ["map_physical_memory"]}
```

启用这个功能后，bootloader 将整个物理内存映射到一些未使用的虚拟地址范围。为了将虚拟地址范围传达给我们的内核，bootloader 传递了一个 启动信息 结构。

五、 2.1 引导信息

引导加载程序 crate 定义了一个 BootInfo 结构，其中包含它传递给我们内核的所有信息。 该结构仍处于早期阶段，因此在更新到未来与 semver 不兼容的引导加载程序版本时预计会出现一些破损。 启用 map_physical_memory 特性后，它目前有两个字段 memory_map 和 physical_memory_offset：

- memory_map 字段包含了可用物理内存的概览。它告诉我们的内核，系统中有多少物理内存可用，哪些内存区域被保留给设备，如 VGA 硬件。内存图可以从 BIOS 或 UEFI 固件中查询，但只能在启动过程的早期查询。由于这个原因，它必须由引导程序提供，因为内核没有办法在以后检索到它。在这篇文章的后面我们将需要内存图。
- physical_memory_offset`告诉我们物理内存映射的虚拟起始地址。通过把这个偏移量加到物理地址上，我们得到相应的虚拟地址。这使得我们可以从我们的内核中访问任意的物理内存。
- 这个物理内存偏移可以通过在 Cargo.toml 中添加一个[package.metadata.bootloader]表并设置 physical-memory-offset = "0x0000f00000000000"（或任何其他值）来定制。然而，请注意，如果 bootloader 遇到物理地址值开始与偏移量以外的空间重叠，也就是说，它以前会映射到其他早期的物理地址的区域，就会出现恐慌。所以一般来说，这个值越高（>1 TiB）越好。

Bootloader 将 “BootInfo “结构以”&’ static BootInfo “参数的形式传递给我们的内核，并传递给我们的”_start “函数。我们的函数中还没有声明这个参数，所以让我们添加它。

```
// in src/main.rs
use bootloader::BootInfo;
```

```
#[no_mangle]
pub extern "C" fn _start(boot_info: &'static BootInfo) -> ! {// new argument
    […]
}
```

以前省去这个参数并不是什么问题，因为 x86_64 的调用惯例在 CPU 寄存器中传递第一个参数。因此，当这个参数没有被声明时，它被简单地忽略了。然而，如果我们不小心使用了一个错误的参数类型，那将是一个问题，因为编译器不知道我们入口点函数的正确类型签名。

六、 2.2 入口点宏

由于我们的 _start 函数是从引导加载程序外部调用的，因此不会检查我们的函数签名。 这意味着我们可以让它接受任意参数而不会出现任何编译错误，但它会在运行时失败或导致未定义的行为。

为了确保入口点函数总是具有引导程序所期望的正确签名，bootloader 板块提供了一个 entry_point 宏，它提供了一种类型检查的方法来定义一个 Rust 函数作为入口点。让我们重写我们的入口点函数来使用这个宏。

```
// in src/main.rs
use bootloader::{BootInfo, entry_point};
entry_point!(kernel_main);
fn kernel_main(boot_info: &'static BootInfo) -> ! {
    […]
}
```

我们不再需要使用 extern "C"或 no_mangle 作为我们的入口点，因为宏为我们定义了真正的低级_start 入口点。kernel_main 函数现在是一个完全正常的 Rust 函数，所以我们可以为它选择一个任意的名字。重要的是，它是经过类型检查的，所以当我们使用一个错误的函数签名时，例如增加一个参数或改变参数类型，就会发生编译错误。

让我们在我们的 lib.rs 中进行同样的修改。

```
// in src/lib.rs
#[cfg(test)]
use bootloader::{entry_point, BootInfo};
#[cfg(test)]
entry_point!(test_kernel_main);
/// Entry point for `cargo test`
#[cfg(test)]
fn test_kernel_main(_boot_info: &'static BootInfo) -> ! {
    // like before
    init();
    test_main();
    hlt_loop();
}
```

由于这个入口点只在测试模式下使用，我们在所有项目中添加了#[cfg(test)]属性。我们给我们的测试入口点一个独特的名字test_kernel_main，以避免与我们的 main.rs 的 kernel_main 混淆。我们现在不使用 BootInfo 参数，所以我们在参数名前加上_，以消除未使用变量的警告。

3. 实现

现在我们可以访问物理内存了，我们终于可以开始实现我们的页表代码了。 首先，我们将查看我们的内核运行的当前活动页表。 在第二步中，我们将创建一个转换函数，该函数返回给定虚拟地址映射到的物理地址。 作为最后一步，我们将尝试修改页表以创建新映射。

在我们开始之前，我们为我们的代码创建一个新的 memory 模块。

```
// in src/lib.rs
pub mod memory;
```

对于该模块，我们创建一个空的 src/memory.rs 文件。

七、 3.1 访问页表

在上一篇文章的结尾，我们试图查看内核运行的页表，但失败了，因为我们无法访问 CR3 寄存器指向的物理帧。 我们现在可以从那里继续创建一个 active_level_4_table 函数，该函数返回对活动 4 级页表的引用：

```
// in src/memory.rs
use x86_64::{
    structures::paging::PageTable,
    VirtAddr,
};
/// 返回一个对活动的 4 级表的可变引用。
/// 这个函数是不安全的，因为调用者必须保证完整的物理内存在传递的
/// `physical_memory_offset`处被映射到虚拟内存。另外，这个函数
/// 必须只被调用一次，以避免别名"&mut "引用（这是未定义的行为）。
pub unsafe fn active_level_4_table(physical_memory_offset: VirtAddr)
    -> &'static mut PageTable
{
    use x86_64::registers::control::Cr3;
    let (level_4_table_frame, _) = Cr3::read();
    let phys = level_4_table_frame.start_address();
    let virt = physical_memory_offset + phys.as_u64();
    let page_table_ptr: *mut PageTable = virt.as_mut_ptr();
    &mut *page_table_ptr // unsafe
}
```

首先，我们从 CR3 寄存器中读取活动的 4 级表的物理帧。然后我们取其物理起始地址，将其转换为 u64，并将其添加到 physical_memory_offset 中，得到页表框架映射的虚拟地址。最后，我们通过 as_mut_ptr 方法将虚拟地址转换为 *mut PageTable 原始指针，然后不安全地从它创建一个&mut PageTable 引用。

我们创建一个&mut 引用，而不是&引用，因为我们将在本篇文章的后面对页表进行突变。

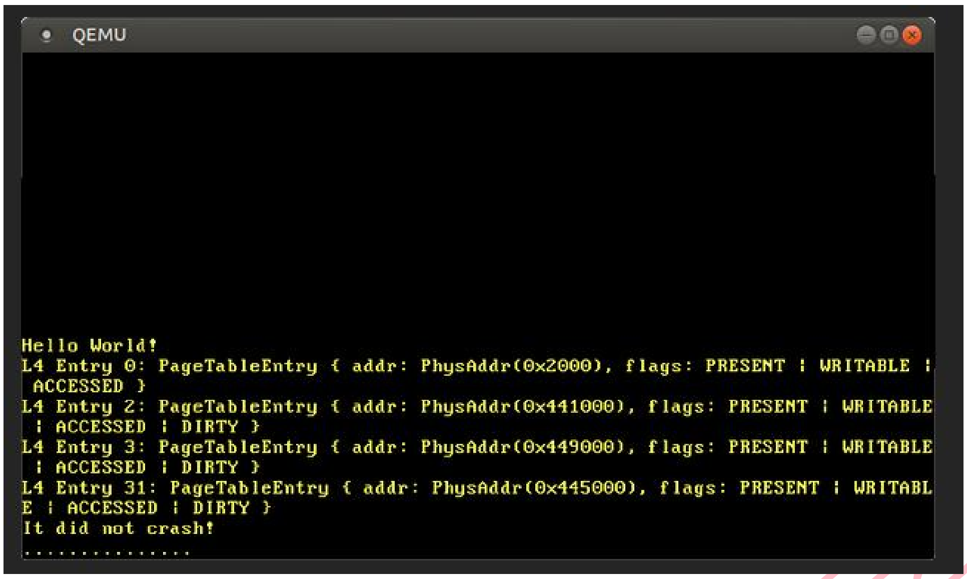
我们不需要在这里使用不安全块，因为 Rust 把一个 “不安全 “fn 的完整主体当作一个大的 “不安全 “块。这使得我们的代码更加危险，因为我们可能会在不知不觉中在前几行引入不安全操作。这也使得在安全操作之间发现不安全操作的难度大大增加。有一个[RFC]
（https://github.com/rust-lang/rfcs/pull/2585）可以改变这种行为。

现在我们可以用这个函数来打印第 4 级表格的条目。

```
// in src/main.rs
fn kernel_main(boot_info: &'static BootInfo) -> ! {
        use blog_os::memory::active_level_4_table;
        use x86_64::VirtAddr;
        println!("Hello World{}", "!");
        blog_os::init();
        let phys_mem_offset = VirtAddr::new(boot_info.physical_memory_offset);
        let l4_table = unsafe { active_level_4_table(phys_mem_offset) };
        for (i, entry) in l4_table.iter().enumerate() {
                if !entry.is_unused() {
                        println!("L4 Entry {}: {:?}", i, entry);
                }
        }
        // as before
        #[cfg(test)]
        test_main();
        println!("It did not crash!");
        blog_os::hlt_loop();
}
```

首先，我们将 “BootInfo” 结构的 “physical_memory_offset “转换为 VirtAddr，并将其传递给 active_level_4_table 函数。然后我们使用 iter 函数来迭代页表条目，并使用 enumerate 组合器为每个元素增加一个索引 i。我们只打印非空的条目，因为所有 512 个条目在屏幕上是放不下的。

当我们运行它时，我们看到以下输出。



```
Hello World!
L4 Entry 0: PageTableEntry { addr: PhysAddr(0x2000), flags: PRESENT | WRITABLE |
 ACCESSED }
L4 Entry 2: PageTableEntry { addr: PhysAddr(0x441000), flags: PRESENT | WRITABLE
 | ACCESSED | DIRTY }
L4 Entry 3: PageTableEntry { addr: PhysAddr(0x449000), flags: PRESENT | WRITABLE
 | ACCESSED | DIRTY }
L4 Entry 31: PageTableEntry { addr: PhysAddr(0x445000), flags: PRESENT | WRITABL
E | ACCESSED | DIRTY }
It did not crash!
...............
```

我们看到有各种非空条目，它们都映射到不同的 3 级表。有这么多区域是因为内核代码、内核堆栈、物理内存映射和启动信息都使用独立的内存区域。

为了进一步遍历页表，看一下三级表，我们可以把一个条目的映射帧再转换为一个虚拟地址。

```
// in the `for` loop in src/main.rs
use x86_64::structures::paging::PageTable;
if !entry.is_unused() {
        println!("L4 Entry {}: {:?}", i, entry);
        // get the physical address from the entry and convert it
        let phys = entry.frame().unwrap().start_address();
        let virt = phys.as_u64() + boot_info.physical_memory_offset;
        let ptr = VirtAddr::new(virt).as_mut_ptr();
        let l3_table: &PageTable = unsafe { &*ptr };
        // print non-empty entries of the level 3 table
        for (i, entry) in l3_table.iter().enumerate() {
                if !entry.is_unused() {
                        println!("    L3 Entry {}: {:?}", i, entry);
                }
        }
}
```

对于查看 2 级和 1 级表，我们对 3 级和 2 级条目重复这一过程。你可以想象，这很快就会变得非常冗长，所以我们不在这里展示完整的代码。

手动遍历页表是很有趣的，因为它有助于了解 CPU 是如何进行转换的。然而，大多数时候，我们只对给定的虚拟地址的映射物理地址感兴趣，所以让我们为它创建一个函数。

八、 3.2 翻译地址

要将虚拟地址转换为物理地址，我们必须遍历四级页表，直到到达映射帧。让我们创建一个执行此翻译的函数：

```
// in src/memory.rs
use x86_64::PhysAddr;
/// 将给定的虚拟地址转换为映射的物理地址，如果地址没有被映射，则为`None'。
/// 这个函数是不安全的，因为调用者必须保证完整的物理内存在
/// 传递的`physical_memory_offset`处被映射到虚拟内存。
pub unsafe fn translate_addr(addr: VirtAddr, physical_memory_offset: VirtAddr)
    -> Option<PhysAddr>
{
    translate_addr_inner(addr, physical_memory_offset)
}
```

我们将该函数转发到一个安全的 `translate_addr_inner` 函数，以限制 `unsafe` 的范围。正如我们在上面指出的，Rust 把一个 `unsafe fn` 的完整主体当作一个大的不安全块。通过调用一个私有的安全函数，我们使每个 `unsafe` 操作再次明确。

私有内部函数包含真正的实现：

```
// in src/memory.rs
/// 由 `translate_addr`调用的私有函数。
/// 这个函数是安全的，可以限制`unsafe`的范围，
/// 因为 Rust 将不安全函数的整个主体视为不安全块。
/// 这个函数只能通过`unsafe fn`从这个模块的外部到达。
fn translate_addr_inner(addr: VirtAddr, physical_memory_offset: VirtAddr)
    -> Option<PhysAddr>
{
    use x86_64::structures::paging::page_table::FrameError;
    use x86_64::registers::control::Cr3;
    // 从 CR3 寄存器中读取活动的 4 级 frame
    let (level_4_table_frame, _) = Cr3::read();
    let table_indexes = [
        addr.p4_index(), addr.p3_index(), addr.p2_index(), addr.p1_index()
    ];
    let mut frame = level_4_table_frame;
    // 遍历多级页表
    for &index in &table_indexes {
        // 将该框架转换为页表参考
        let virt = physical_memory_offset + frame.start_address().as_u64();
        let table_ptr: *const PageTable = virt.as_ptr();
        let table = unsafe {&*table_ptr};
        // 读取页表条目并更新`frame`。
```

```
            let entry = &table[index];
            frame = match entry.frame() {
                Ok(frame) => frame,
                Err(FrameError::FrameNotPresent) => return None,
                Err(FrameError::HugeFrame) => panic!("huge pages not supported"),
            };
        }
        // 通过添加页面偏移量来计算物理地址
        Some(frame.start_address() + u64::from(addr.page_offset()))
}
```

我们没有重复使用 active_level_4_table 函数，而是再次从 CR3 寄存器读取 4 级帧。我们这样做是因为它简化了这个原型的实现。别担心，我们一会儿就会创建一个更好的解决方案。

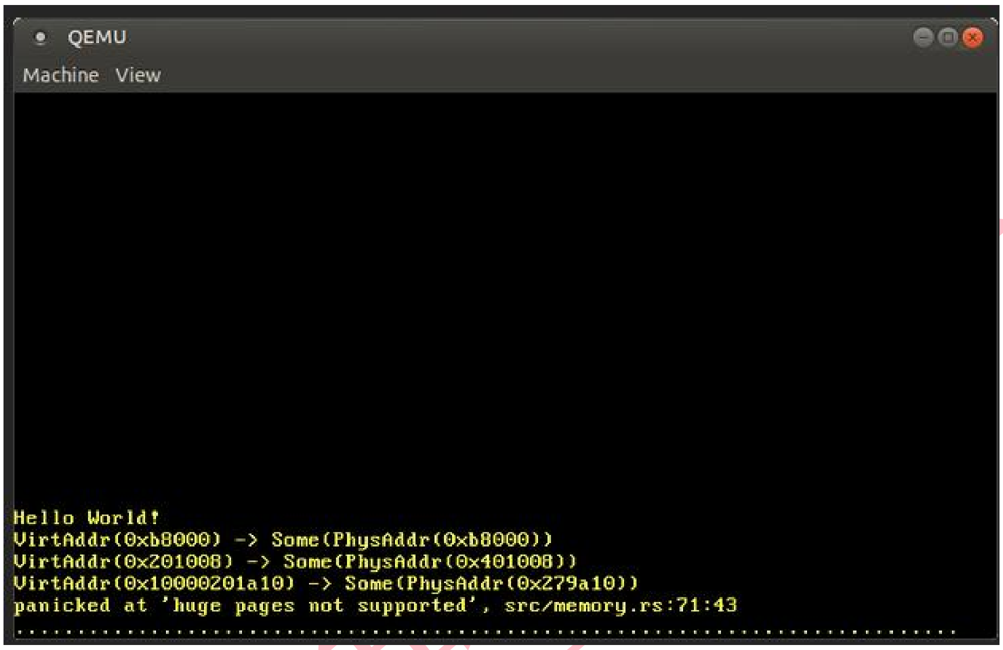
VirtAddr 结构已经提供了计算四级页面表索引的方法。我们将这些索引存储在一个小数组中，因为它允许我们使用 for 循环遍历页表。在循环之外，我们记住了最后访问的 frame'，以便以后计算物理地址。frame`在迭代时指向页表框架，在最后一次迭代后指向映射的框架，也就是在跟随第 1 级条目之后。

在这个循环中，我们再次使用 physical_memory_offset 将帧转换为页表引用。然后我们读取当前页表的条目，并使用 PageTableEntry::frame 函数来检索映射的框架。如果该条目没有映射到一个框架，我们返回`None’ 。如果该条目映射了一个巨大的 2 MiB 或 1 GiB 页面，我们就暂时慌了。

让我们通过翻译一些地址来测试我们的翻译功能。

```
// in src/main.rs
fn kernel_main(boot_info: &'static BootInfo) -> ! {
    // new import
    use blog_os::memory::translate_addr;
    [⋯] // hello world and blog_os::init
    let phys_mem_offset = VirtAddr::new(boot_info.physical_memory_offset);
    let addresses = [
        // the identity-mapped vga buffer page
        0xb8000,
        // some code page
        0x201008,
        // some stack page
        0x0100_0020_1a10,
        // virtual address mapped to physical address 0
        boot_info.physical_memory_offset,
    ];
    for &address in &addresses {
        let virt = VirtAddr::new(address);
        let phys = unsafe { translate_addr(virt, phys_mem_offset) };
        println!("{:?} -> {:?}", virt, phys);
    }
    […] // test_main(), "it did not crash" printing, and hlt_loop()
}
```

当我们运行它时，我们看到以下输出。



```
Hello World!
VirtAddr(0xb8000) -> Some(PhysAddr(0xb8000))
VirtAddr(0x201008) -> Some(PhysAddr(0x401008))
VirtAddr(0x10000201a10) -> Some(PhysAddr(0x279a10))
panicked at 'huge pages not supported', src/memory.rs:71:43
```

正如预期的那样，身份映射的地址 0xb8000 翻译成了相同的物理地址。代码页和堆栈页翻译成了一些任意的物理地址，这取决于引导程序如何为我们的内核创建初始映射。值得注意的是，最后 12 位在翻译后总是保持不变，这是有道理的，因为这些位是 page offset，不是翻译的一部分。

由于每个物理地址都可以通过添加 physical_memory_offset 来访问，physical_memory_offset 地址的翻译本身应该指向物理地址 0。然而，翻译失败了，因为映射使用了巨大的页面来提高效率，而我们的实现还不支持。

九、 3.3 使用偏移页表

将虚拟地址转换为物理地址是操作系统内核中的一项常见任务，因此 x86_64 crate 为其提供了抽象。 除了 translate_addr 之外，该实现已经支持大页面和其他几个页表函数，因此我们将在下面使用它，而不是在我们自己的实现中添加大页面支持。

抽象的基础是定义各种页表映射函数的两个特征：

- Mapper 特质在页面大小上是通用的，并提供对页面进行操作的函数。例如 translate_page，它将一个给定的页面翻译成相同大小的框架，以及 map_to，它在页面表中创建一个新的映射。
- Translate 特性提供了与多个页面大小有关的函数，如 translate_addr 或一般 translate。

特质只定义接口，不提供任何实现。x86_64 板块目前提供了三种类型来实现不同要求的特征。OffsetPageTable 类型假设完整的物理内存被映射到虚拟地址空间的某个偏移处。MappedPageTable 更灵活一些。它只要求每个页表帧在一个可计算的地址处被映射到虚拟地址空间。最后，[递归页表]类型可以用来通过递归页表访问页表框架。

在我们的例子中，bootloader 在 physical_memory_offset 变量指定的虚拟地址上映射完整的物理内存，所以我们可以使用 OffsetPageTable 类型。为了初始化它，我们在 memory 模块中创建一个新的 init 函数。

```
use x86_64::structures::paging::OffsetPageTable;
/// 初始化一个新的 OffsetPageTable。
/// 这个函数是不安全的，因为调用者必须保证完整的物理内存在
/// 传递的`physical_memory_offset`处被映射到虚拟内存。另
/// 外，这个函数必须只被调用一次，以避免别名"&mut "引用（这是未定义的行为）。
pub unsafe fn init(physical_memory_offset: VirtAddr) -> OffsetPageTable<'static> {
    let level_4_table = active_level_4_table(physical_memory_offset);
    OffsetPageTable::new(level_4_table, physical_memory_offset)
}
// 私下进行
unsafe fn active_level_4_table(physical_memory_offset: VirtAddr)
    -> &'static mut PageTable
{…}
```

该函数接受 “physical_memory_offset “作为参数，并返回一个新的“OffsetPageTable “实例，该实例具有 “静态 “寿命。这意味着该实例在我们内核的整个运行时间内保持有效。在函数体中，我们首先调用“active_level_4_table “函数来获取 4 级页表的可变引用。然后我们用这个引用调用 OffsetPageTable::new 函数。作为第二个参数，new’函数希望得到物理内存映射开始的虚拟地址，该地址在 physical_memory_offset’ 变量中给出。

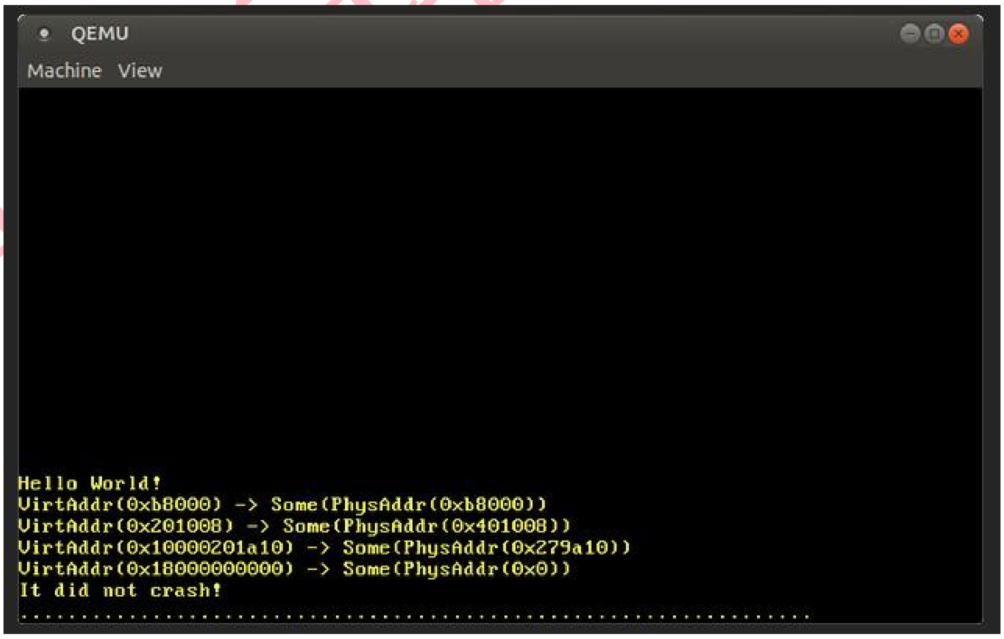
从现在开始，active_level_4_table’函数只能从 init’ 函数中调用，因为它在多次调用时很容易导致别名的可变引用，这可能导致未定义的行为。出于这个原因，我们通过删除 pub 指定符使该函数成为私有的。

我们现在可以使用 Translate::translate_addr 方法而不是我们自己的 memory::translate_addr 函数。我们只需要在 kernel_main 中修改几行。

```
// in src/main.rs
fn kernel_main(boot_info: &'static BootInfo) -> ! {
    // new: different imports
    use blog_os::memory;
    use x86_64::{structures::paging::Translate, VirtAddr};
    [···] // hello world and blog_os::init
    let phys_mem_offset = VirtAddr::new(boot_info.physical_memory_offset);
    // new: initialize a mapper
    let mapper = unsafe { memory::init(phys_mem_offset) };
    let addresses = [···]; // same as before
    for &address in &addresses {
        let virt = VirtAddr::new(address);
        // new: use the `mapper.translate_addr` method
        let phys = mapper.translate_addr(virt);
        println!("{:?} -> {:?}", virt, phys);
    }
    [···] // test_main(), "it did not crash" printing, and hlt_loop()
}
```

我们需要导入 Translate 特性，以便使用它提供的 translate_addr 方法。

当我们现在运行它时，我们看到和以前一样的翻译结果，不同的是，巨大的页面翻译现在也在工作。



```
Hello World!
VirtAddr(0xb8000) -> Some(PhysAddr(0xb8000))
VirtAddr(0x201008) -> Some(PhysAddr(0x401008))
VirtAddr(0x10000201a10) -> Some(PhysAddr(0x279a10))
VirtAddr(0x18000000000) -> Some(PhysAddr(0x0))
It did not crash!
```

正如预期的那样，0xb8000 的翻译以及代码和堆栈地址与我们自己的翻译函数保持一致。此外，我们现在看到，虚拟地址 physical_memory_offset 被映射到物理地址 0x0。

通过使用 MappedPageTable 类型的翻译函数，我们可以免除实现巨大页面支持的工作。我们还可以访问其他的页面函数，如 map_to，我们将在下一节使用。

在这一点上，我们不再需要 memory::translate_addr 和 memory::translate_addr_inner 函数，所以我们可以删除它们。

十、 3.4 创建一个新的映射

到目前为止，我们只查看了页表而没有修改任何内容。 让我们通过为以前未映射的页面创建新映射来改变它。

我们将使用 Mapper 特性的 map_to 函数来实现，所以让我们先看一下这个函数。文档告诉我们，它需要四个参数：我们想要映射的页面，该页面应该被映射到的框架，一组页面表项的标志，以及一个 frame_allocator。之所以需要框架分配器，是因为映射给定的页面可能需要创建额外的页表，而页表需要未使用的框架作为后备存储。

3.4.1 一个 create_example_mapping 函数

我们实现的第一步是创建一个新的 create_example_mapping 函数，它将给定的虚拟页面映射到 0xb8000，即 VGA 文本缓冲区的物理帧。 我们选择该框架是因为它允许我们轻松测试映射是否正确创建：我们只需要写入新映射的页面并查看我们是否看到写入出现在屏幕上。

create_example_mapping 函数看起来像这样：

```
// in src/memory.rs
use x86_64::{
        PhysAddr,
        structures::paging::{Page, PhysFrame, Mapper, Size4KiB, FrameAllocator}
};
/// 为给定的页面创建一个实例映射到框架`0xb8000`。
pub fn create_example_mapping(
        page: Page,
        mapper: &mut OffsetPageTable,
        frame_allocator: &mut impl FrameAllocator<Size4KiB>,
) {
        use x86_64::structures::paging::PageTableFlags as Flags;
        let frame = PhysFrame::containing_address(PhysAddr::new(0xb8000));
        let flags = Flags::PRESENT | Flags::WRITABLE;
        let map_to_result = unsafe {
            // FIXME: 这并不安全，我们这样做只是为了测试。
            mapper.map_to(page, frame, flags, frame_allocator)
        };
        map_to_result.expect("map_to failed").flush();
}
```

除了应映射的页面之外，该函数还需要对 OffsetPageTable 实例和 frame_allocator 的可变引用。 frame_allocator 参数使用 impl Trait 语法对实现 FrameAllocator 特征的所有类型通用。 该特性比 PageSize 特性更通用，可用于标准的 4 KiB 页面和巨大的 2 MiB/1 GiB 页面。 我们只想创建一个 4 KiB 映射，因此我们将通用参数设置为 Size4KiB。

map_to 方法是不安全的，因为调用者必须确保该框架尚未被使用。 这样做的原因是两次映射同一帧可能会导致未定义的行为，例如当两个不同的 &mut 引用指向同一物理内存位置时。 在我们的例子中，我们重用了已经映射的 VGA 文本缓冲区帧，因此我们打破了要求的条件。 不过 create_example_mapping 函数只是一个临时测试函数，发完后会去掉，所以没问题。 为了提醒我们不安全，我们在线上添加了 FIXME 注释。

除了页面和 unused_frame 之外，map_to 方法还采用了一组用于映射的标志和对 frame_allocator 的引用，稍后将对此进行解释。 对于标志，我们设置了 PRESENT 标志，因为它是所有有效条目所必需的，并且设置了 WRITABLE 标志以使映射页面可写。 有关所有可能标志的列表，请参阅上一篇文章的页表格式部分。

map_to 函数可能会失败，因此它会返回一个结果。 由于这只是一些不需要健壮的示例代码，我们只是使用 expect 在发生错误时恐慌。 成功时，该函数返回一个 MapperFlush 类型，该类型提供了一种使用其 flush 方法从转换后备缓冲区 (TLB) 刷新新映射页面的简单方法。 与 Result 一样，该类型使用 #[must_use] 属性在我们不小心忘记使用它时发出警告。

3.4.2 一个虚拟的 FrameAllocator

为了能够调用 create_example_mapping，我们需要先创建一个实现 FrameAllocator 特性的类型。 如上所述，如果 map_to 需要，该特征负责为新页表分配帧。

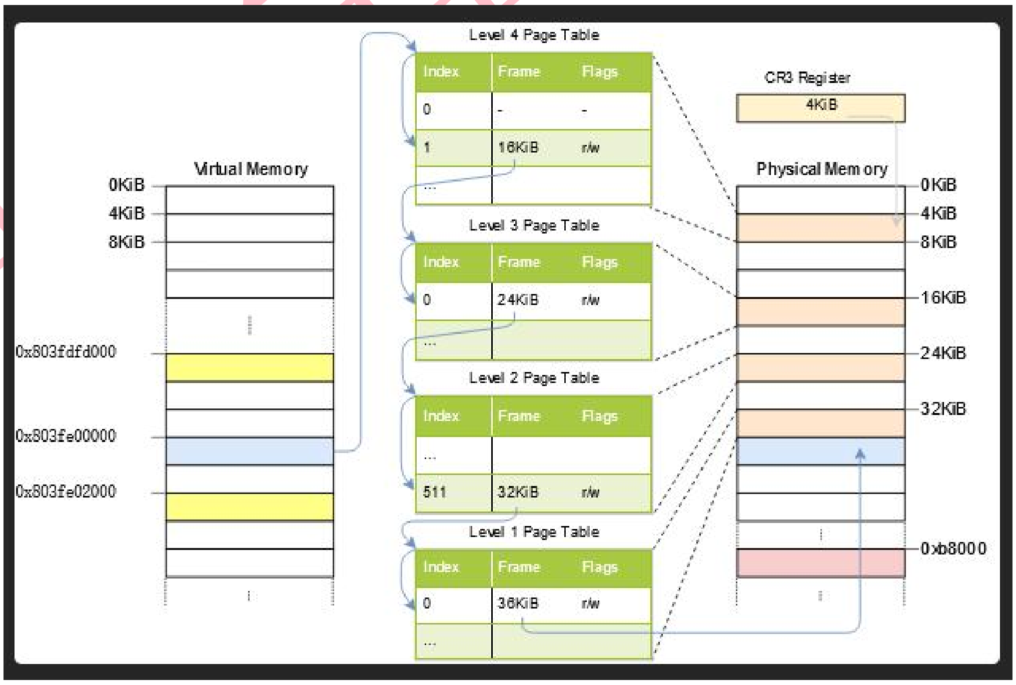
让我们从简单的情况开始，假设我们不需要创建新的页面表。对于这种情况，一个总是返回 “无 “的框架分配器就足够了。我们创建这样一个`空框架分配器’来测试我们的映射函数。

```
// in src/memory.rs
/// 一个总是返回`None'的 FrameAllocator。
pub struct EmptyFrameAllocator;
unsafe impl FrameAllocator<Size4KiB> for EmptyFrameAllocator {
    fn allocate_frame(&mut self) -> Option<PhysFrame> {
        None
    }
}
```

实现 FrameAllocator 是不安全的，因为实现者必须保证分配器只产生未使用的帧。否则，可能会发生未定义的行为，例如，当两个虚拟页被映射到同一个物理帧时。我们的 “空框架分配器 “只返回 “无”，所以在这种情况下，这不是一个问题。

3.4.3 选择虚拟页面

我们现在有一个简单的帧分配器，我们可以将其传递给我们的 create_example_mapping 函数。 但是，分配器总是返回 None，所以这只有在不需要额外的页表框架来创建映射时才有效。 要了解何时需要以及何时不需要额外的页表框架，让我们考虑一个示例：



图中左边是虚拟地址空间，右边是物理地址空间，中间是页表。页表被存储在物理内存框架中，用虚线表示。虚拟地址空间包含一个地址为 0x803fe00000 的单一映射页，用蓝色标记。为了将这个页面转换到它的框架，CPU 在 4 级页表上行走，直到到达地址为 36 KiB 的框架。

此外，该图用红色显示了 VGA 文本缓冲区的物理帧。我们的目标是使用 create_example_mapping 函数将一个先前未映射的虚拟页映射到这个帧。由于我们的 EmptyFrameAllocator’总是返回 None’，我们想创建映射，这样就不需要分配器提供额外的帧。这取决于我们为映射选择的虚拟页。

图中显示了虚拟地址空间中的两个候选页，都用黄色标记。一个页面在地址 0x803fdfd000，比映射的页面（蓝色）早 3 页。虽然 4 级和 3 级页表的索引与蓝色页相同，但 2 级和 1 级的索引不同（见上一篇）。2 级表的不同索引意味着这个页面使用了一个不同的 1 级表。由于这个 1 级表还不存在，如果我们选择该页作为我们的例子映射，我们就需要创建它，这就需要一个额外的未使用的物理帧。相比之下，地址为 0x803fe02000 的第二个候选页就没有这个问题，因为它使用了与蓝色页面相同的 1 级页表。因此，所有需要的页表都已经存在。

总之，创建一个新的映射的难度取决于我们想要映射的虚拟页。在最简单的情况下，该页的 1 级页表已经存在，我们只需要写一个条目。在最困难的情况下，该页是在一个还不存在第三级的内存区域，所以我们需要先创建新的第三级、第二级和第一级页表。

为了用 “EmptyFrameAllocator “调用我们的 “create_example_mapping “函数，我们需要选择一个所有页表都已存在的页面。为了找到这样的页面，我们可以利用 bootloader 在虚拟地址空间的第一兆字节内加载自己的事实。这意味着这个区域的所有页面都存在一个有效的 1 级表。因此，我们可以选择这个内存区域中任何未使用的页面作为我们的例子映射，比如地址为 0 的页面。通常情况下，这个页面应该保持未使用状态，以保证解读空指针会导致页面故障，所以我们知道 bootloader 没有将其映射。

3.4.4 创建映射

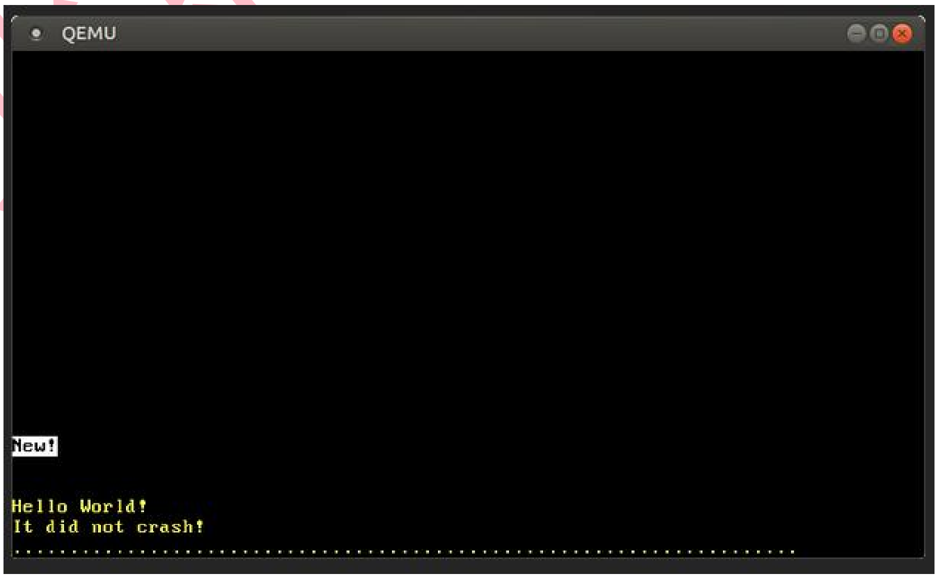
我们现在拥有调用 create_example_mapping 函数所需的所有参数，所以让我们修改 kernel_main 函数以将页面映射到虚拟地址 0。由于我们将页面映射到 VGA 文本缓冲区的帧，我们应该能够写入 之后筛选它。 实现看起来像这样：

```
// in src/main.rs
fn kernel_main(boot_info: &'static BootInfo) -> ! {
    use blog_os::memory;
    use x86_64::{structures::paging::Page, VirtAddr}; // 新的导入
    […] // hello world and blog_os::init
    let phys_mem_offset = VirtAddr::new(boot_info.physical_memory_offset);
    let mut mapper = unsafe { memory::init(phys_mem_offset) };
    let mut frame_allocator = memory::EmptyFrameAllocator;
    // 映射未使用的页
    let page = Page::containing_address(VirtAddr::new(0));
    memory::create_example_mapping(page, &mut mapper, &mut frame_allocator);
    // 通过新的映射将字符串 `New!` 写到屏幕上。
    let page_ptr: *mut u64 = page.start_address().as_mut_ptr();
    unsafe { page_ptr.offset(400).write_volatile(0x_f021_f077_f065_f04e)};
    […] // test_main(), "it did not crash" printing, and hlt_loop()
}
```

我们首先通过调用 “create_example_mapping “函数为地址为 0 的页面创建映射，并为 “mapper “和 “frame_allocator “实例提供一个可变的引用。这将页面映射到 VGA 文本缓冲区框架，所以我们应该在屏幕上看到对它的任何写入。

然后我们将页面转换为原始指针，并写一个值到偏移量 400。我们不写到页面的开始，因为 VGA 缓冲区的顶行被下一个 println 直接移出了屏幕。我们写值 0x_f021_f077_f065_f04e，表示白色背景上的字符串 “New!” 。正如我们在 _“VGA 文本模式”_帖子中所学到的，对 VGA 缓冲区的写入应该是不稳定的，所以我们使用 write_volatile 方法。

当我们在 QEMU 中运行它时，我们看到以下输出。



```
New!


Hello World!
It did not crash!
..........................................................................
```

屏幕上的 “New!” 是由我们写到页 0 引起的，这意味着我们成功地在页表中创建了一个新的映射。

创建该映射只是因为负责地址为 0 的页面的 1 级表已经存在。当我们试图映射一个还不存在一级表的页面时，map_to 函数失败了，因为它试图通过用 EmptyFrameAllocator 分配帧来创建新的页表。当我们试图映射 0xdeadbeaf000 而不是 0 页面时，我们可以看到这种情况发生。

```
// in src/main.rs
fn kernel_main(boot_info: &'static BootInfo) -> ! {
    […]
    let page = Page::containing_address(VirtAddr::new(0xdeadbeaf000));
    […]
}
```

当我们运行它时，出现了恐慌，并有以下错误信息。

```
    panicked at 'map_to failed: FrameAllocationFailed', /…/result.rs:999:5
```

为了映射那些还没有一级页表的页面，我们需要创建一个合适的 FrameAllocator。但是我们如何知道哪些帧是未使用的，以及有多少物理内存是可用的？

十一、 3.5 分配帧

为了创建新的页表，我们需要创建一个合适的帧分配器。 为此，我们使用引导加载程序传递的 memory_map 作为 BootInfo 结构的一部分：

```
// in src/memory.rs
use bootloader::bootinfo::MemoryMap;
/// 一个 FrameAllocator，从 bootloader 的内存地图中返回可用的 frames。
pub struct BootInfoFrameAllocator {
    memory_map: &'static MemoryMap,
    next: usize,
}
impl BootInfoFrameAllocator {
    /// 从传递的内存 map 中创建一个 FrameAllocator。
    ///
    /// 这个函数是不安全的，因为调用者必须保证传递的内存 map 是有效的。
    /// 主要的要求是，所有在其中被标记为 "可用 "的帧都是真正未使用的。
    pub unsafe fn init(memory_map: &'static MemoryMap) -> Self {
        BootInfoFrameAllocator {
            memory_map,
            next: 0,
        }
    }
}
```

该结构有两个字段。一个是对 bootloader 传递的内存 map 的 'static 引用，一个是跟踪分配器应该返回的下一帧的 next 字段。

正如我们在启动信息部分所解释的，内存图是由 BIOS/UEFI 固件提供的。它只能在启动过程的早期被查询，所以引导程序已经为我们调用了相应的函数。内存地图由 MemoryRegion 结构列表组成，其中包含每个内存区域的起始地址、长度和类型（如未使用、保留等）。

init 函数用一个给定的内存映射初始化一个 BootInfoFrameAllocator。next 字段被初始化为 0，并将在每次分配帧时增加，以避免两次返回相同的帧。由于我们不知道内存映射的可用帧是否已经在其他地方被使用，我们的 init 函数必须是不安全的，以要求调用者提供额外的保证。

3.5.1 一个 usable_frames 方法

在我们实现 FrameAllocator 特性之前，我们添加一个辅助方法，将内存映射转换为可用帧的迭代器：

```
// in src/memory.rs
use bootloader::bootinfo::MemoryRegionType;
impl BootInfoFrameAllocator {
    /// 返回内存映射中指定的可用框架的迭代器。
    fn usable_frames(&self) -> impl Iterator<Item = PhysFrame> {
        // 从内存 map 中获取可用的区域
        let regions = self.memory_map.iter();
        let usable_regions = regions
            .filter(|r| r.region_type == MemoryRegionType::Usable);
        // 将每个区域映射到其地址范围
        let addr_ranges = usable_regions
            .map(|r| r.range.start_addr()..r.range.end_addr());
        // 转化为一个帧起始地址的迭代器
        let frame_addresses = addr_ranges.flat_map(|r| r.step_by(4096));
        // 从起始地址创建 `PhysFrame` 类型
        frame_addresses.map(|addr| PhysFrame::containing_address(PhysAddr::new(addr)))
    }
}
```

这个函数使用迭代器组合方法将初始的 MemoryMap 转化为可用的物理帧的迭代器。

- 首先，我们调用 iter 方法，将内存映射转换为 MemoryRegions 的迭代器。
- 然后我们使用 filter 方法跳过任何保留或其他不可用的区域。Bootloader 为它创建的所有映射更新了内存地图，所以被我们的内核使用的帧（代码、数据或堆栈）或存储启动信息的帧已经被标记为 InUse 或类似的。因此，我们可以确定“可使用”的

帧没有在其他地方使用。

- 之后，我们使用 map 组合器和 Rust 的 range 语法将我们的内存区域迭代器转化为地址范围的迭代器。
- 接下来，我们使用 flat_map 将地址范围转化为帧起始地址的迭代器，使用 step_by 选择每 4096 个地址。由于 4096 字节（=4 KiB）是页面大小，我们得到了每个帧的起始地址。Bootloader 对所有可用的内存区域进行页对齐，所以我们在这里不需要任何对齐或舍入代码。通过使用 flat_map 而不是 map，我们得到一个 Iterator 而不是 Iterator。
- 最后，我们将起始地址转换为 PhysFrame 类型，以构建一个 Iterator。

该函数的返回类型使用了 impl Trait 特性。这样，我们可以指定返回某个实现 Iterator 特质的类型，项目类型为 PhysFrame，但不需要命名具体的返回类型。这在这里很重要，因为我们不能命名具体的类型，因为它依赖于不可命名的闭包类型。

3.5.2 实现 FrameAllocator 特性

现在我们可以实现 FrameAllocator 特性：

```
// in src/memory.rs
unsafe impl FrameAllocator<Size4KiB> for BootInfoFrameAllocator {
        fn allocate_frame(&mut self) -> Option<PhysFrame> {
                let frame = self.usable_frames().nth(self.next);
                self.next += 1;
                frame
        }
}
```

我们首先使用 usable_frames 方法，从内存 map 中获得一个可用帧的迭代器。然后，我们使用 Iterator::nth 函数来获取索引为 self.next 的帧（从而跳过(self.next - 1)帧）。在返回该帧之前，我们将 self.next 增加 1，以便在下次调用时返回下一帧。

这个实现不是很理想，因为它在每次分配时都会重新创建 usable_frame 分配器。最好的办法是直接将迭代器存储为一个结构域。这样我们就不需要 nth 方法了，可以在每次分配时直接调用 next。这种方法的问题是，目前不可能将“impl Trait “类型存储在一个结构字段中。当[name existential types]完全实现时，它可能会在某一天发挥作用。

3.5.3 使用 BootInfoFrameAllocator

我们现在可以修改我们的 kernel_main 函数来传递一个 BootInfoFrameAllocator 实例而不是 EmptyFrameAllocator：

```
// in src/main.rs
fn kernel_main(boot_info: &'static BootInfo) -> ! {
        use blog_os::memory::BootInfoFrameAllocator;
        […]
        let mut frame_allocator = unsafe {
            BootInfoFrameAllocator::init(&boot_info.memory_map)
        };
        […]
}
```

通过启动信息框架分配器，映射成功了，我们又在屏幕上看到了白底黑字的"New!" 。在幕后，map_to 方法以如下方式创建了丢失的页表。

- 使用传递的 frame_allocator 来分配一个未使用的框架。
- 将框架归零，创建一个新的、空的页表。
- 将上一级表的条目映射到该框架。
- 继续下一级的表。

虽然我们的 create_example_mapping 函数只是一些示例代码，但我们现在能够为任意的页面创建新的映射。这对于分配内存或在未来的文章中实现多线程是至关重要的。

此时，我们应该再次删除 create_example_mapping 函数，以避免意外地调用未定义的行为，正如[上面](#create_example_mapping 函数)所解释的那样。

总结

在这篇文章中，我们了解了访问页表物理帧的不同技术，包括恒等映射、完整物理内存的映射、临时映射和递归页表。 我们选择映射完整的物理内存，因为它简单、便携且功能强大。

如果没有页表访问，我们无法从内核映射物理内存，因此我们需要引导加载程序的支持。 bootloader crate 支持通过可选的 cargo crate 功能创建所需的映射。 它以 &BootInfo 参数的形式将所需信息传递给我们的内核，并将其传递给我们的入口点函数。

对于我们的实现，我们首先手动遍历页表以实现翻译功能，然后使用 x86_64 crate 的 MappedPageTable 类型。 我们还学习了如何在页表中创建新映射，以及如何在引导加载程序传递的内存映射之上创建必要的 FrameAllocator。