OSLab 2 Report

2200013213

October 18, 2024

1 Overview

OSLab 2 focuses on implementing memory management code for JOS. The primary tasks involved are:

1. Developing a physical memory allocator.
2. Creating page table management code that supports virtual memory.

2 Normal Task

2.1 Exercise 1

We need to implement five functions in `kern/pmap.c`: `boot_alloc`, `mem_init`, `page_init`, `page_alloc`, and `page_free`. I implemented them in execution order.

After computing the usable memory, `boot_alloc` is called to allocate memory for the kernel, returning the address for the page directory. Then, `mem_init` initializes the pages information table. `page_init` initializes the pages information table and marks the pages that are not usable.

Most of the code follows the comments provided in the source code.

In this way, we complete the first exercise.

2.2 Exercises 2 & 3

These two exercises provide basic information about the page table and virtual memory.

For the question in Exercise 3, the type of `x` is `uintptr_t`. The type of `value` is a pointer, which should be a virtual address. Therefore, the type of `x` should be `uintptr_t`.

2.3 Exercise 4

We need to implement `pgdir_walk`, `boot_map_region`, `page_lookup`, `page_remove`, and `page_insert`.

`pgdir_walk` is used to get the page table entry for a virtual address. We first obtain the page directory entry for the virtual address. For the second-level page table, we need to use `KADDR` to retrieve the corresponding address. If the page table entry is not present, we may need to allocate a new page table.

`boot_map_region` is used to map a region of physical memory to a region of virtual memory. We use `pgdir_walk` to get the page table entry for the virtual address, then set the page table entry to the corresponding physical address.

`page_lookup` is used to obtain the page mapped to a virtual address. We utilize `pgdir_walk` to get the page table entry for the virtual address.

When we use `pa2page` to retrieve the page, we must check if the page is present.

`page_remove` is used to remove the mapping for a virtual address. This implementation follows the comments provided in the code.

`page_insert` is used to insert a mapping from a physical address to a virtual address. This implementation also follows the comments in the code.

In this way, we complete the fourth exercise.

2.4 Exercise 5

2.4.1 Question 1

We completed the code in `mem_init`. This is done by repeatedly calling `boot_map_region` to map physical memory to virtual memory, as indicated by the comments in the code.

2.4.2 Question 2

The base virtual addresses for entries `1022` and `1023` are `0xff800000` and `0xffc00000`, respectively. Entry `0` points to the page table for the kernel, which is mapped to the first 4MB of physical memory.

2.4.3 Question 3

This is due to the difference in `PTE_U`, which will be utilized by the MMU.

2.4.4 Question 4

We need to map all physical memory to virtual memory below `KERNBASE`, which is `0xf0000000`. So the maximum amount of physical memory that this kernel can manage is `256MB`.

However, if we remove this restriction, since a two-level page table is used, the maximum amount of physical memory that this kernel can manage is `4GB`. But in this case we need to modify the code such as $KADDR$.

2.4.5 Question 5

The memory required for the virtual page table is $(1024 + 1) \times 4$ KB ≈ 4 MB. And the space of $Page\_info$ is $8B$ for each $4KB$ page. So we need

$$256MB/4KB \times 8B = 512KB$$

Or

$$4GB/4KB \times 8B = 8MB$$

for the $Page\_info$.

Therefore, the total memory required is approximately

$$4 \text{ MB} + 512 \text{ KB} = 4.5 \text{ MB}$$

or

$$4 \text{ MB} + 8 \text{ MB} = 12 \text{ MB}$$

. However,in most cases, only a few virtual page tables are present in physical memory. So the actual memory required is much less than the calculated value.

2.4.6 Question 6

We jump above `KERNBASE` in the following code:

```
mov $relocated, %eax
jmp *%eax
```

This transition is possible because we have already mapped the interval $[KERNBASE, KERNBASE+ 4MB)$ to $[0, 4MB)$. By doing this, we can access the memory using the virtual address.

3 Challenge Task

In this section, I will introduce the approach to implementing the challenge task.

3.1 Support for 4MB Pages

According to references and guidance available online, we need to modify `boot_map_region` to support 4MB pages. The main idea is to set the `PTE_PS` bit in the page table entry. Additionally, we need to modify the test code, as it currently cannot test 4MB pages. However, this implementation seems relatively straightforward, so I wonder if I might be overlooking some complexities.

To better control the use of bigpage. I use $\#def\ BIGPAGE\_FLAG$ to control the support of bigpage.

3.2 Extending the JOS Kernel Monitor

For the `showmap` command, we need to display the mapping for a range of virtual addresses. We can use `pgdir_walk` to obtain the page table entry for each virtual address within the specified range. Then, we print the virtual address, physical address, and permissions for each virtual address.

For the `setperm` command, we need to set the permission for a specific page. This process is similar to `showmap`; however, we set the permission bit after retrieving the page table entry.

The `dump` command presents the most challenges. For the physical memory that is mapped, we simply access those virtual addresses and print their content. For the virtual memory, we first use `pgdir_walk` to obtain the page table entry and then print the content of the corresponding page.

3.3 Mapping Kernel

We need to establish a kernel page directory and a user page directory. To support system calls, it is essential to map a common area accessible to both the kernel and the user. This shared area is utilized for passing parameters and returning results.

3.4 Full User Virtual Memory

We may require a `kern_move` function to relocate the kernel when the previous address is needed by the user. When entering kernel mode, we must move the kernel back to its original location. Upon returning to user mode, the kernel needs to be relocated once again.

To utilize the user address, we have two options: either save the address in the kernel (which consumes too much memory) or return to user mode and then use the address (which is too slow).

In my opinion, full user virtual memory is not necessary. For a typical user, there is no need to be aware of kernel addresses. For computer science professionals, it is preferable to understand and adhere to traditional memory management practices. In special cases where memory is extremely limited, the methods discussed above may still incur excessive memory usage or time costs. Consequently, I have yet to find a compelling reason to implement this until a better design is proposed.

3.5 Continuous Memory Allocation

The task requires us to support more efficient continuous memory allocation. An extremely simple approach is to traverse the entire `page_free_list` to find the first contiguous memory block. This method allows us to allocate continuous memory for any size.

However, this approach is too slow. A lab in the ICS course requires us to manage memory at the byte level. Therefore, we can use the buddy system to manage memory more efficiently. However, for JOS, implementing the buddy system is too complex.