Lab2-report

蒲彦丞-2200012956

Attention: I completed some some challenges, you can see details in challenge part.

Part 1

Exercise 1:

boot_alloc()

Return directly when `n=0`; otherwise, store the value of `nextfree` before updating as the return value, then increase `nextfree` by the size of `alloc`. Make sure the memory is aligned. Panic when memory overflow occurs.

mem_init()

Call `boot_alloc()` to allocate space and initialize it to zero.

page_init() & page_free()

Set according to the original code as required.

page_alloc()

Take a free physical page from `page_free_list`. If `alloc_flags` contains `ALLOC_ZERO`, the page needs to be zeroed out. Finally, return a pointer to that page.

Part 2

Exercise 2:

Omitted.

Exercise 3:

Q1

Since `value` is a pointer type, `x` would have type `uintptr_t`.

Exercise 4:

pgdir_walk()

Note the case where the page directory entry does not have a page table (`!(*pde & PTE_P)`) and the condition where creating a new page table is not allowed (`!create`). Finally, return a pointer to the page table entry.

boot_map_region()

Simply map the virtual address space `[va, va+size)` to the physical space `[pa, pa+size)` in the page table rooted at `pgdir`.

page_lookup()

If there is no page table entry, or the page table entry is invalid, return `NULL`. If `pte_store` is not `NULL`, store the PTE address into `pte_store`.

page_remove()

If there is a physical page mapped at `va`, unmap the physical page; otherwise, do nothing.

page_insert()

Map the physical page `pp` to the virtual address `va`, remove any already mapped pages, increment the physical page reference count, and update the page table entry.

Note an extreme case: the same `pp` is re-inserted at the same virtual address in the same `pgdir`. In this case, the `page_decref()` will free `pp` if there are no more references. Therefore, we should move `pp->pp_ref++;` to before `page_remove(pgdir, va);`.

(Interestingly, when I asked GPT for improvement suggestions on each section of the code after completing the task, it insisted on moving `pp->pp_ref++;` after.)

Part 3

Exercise 5

mem_init()

Omitted.

Q2

Entry	Base Virtual Address	Points to (logically):
1023	0xffc00000	Page table for top [252,256) MB of phys memory
1022	0xff800000	Page table for top [248,252) MB of phys memory
...	...	...
960	0xf0000000	Page table for top [0,4) MB of phys memory
959	0xefc00000	First 8 PDEs are page table for `bootstack`
958	0xef800000	ULIM
957	0xef400000	Page directory
956	0xef000000	`pages` data structure
955	0xeec00000	Unmapped
...	...	Unmapped
2	0x00800000	Unmapped
1	0x00400000	Unmapped
0	0x00000000	[see next question]

Q3

The page table can set permission bits. If the PTE_U is not set to 0, then the user does not have read/write access.

Q4

Before setting virtual memory, `pages` is part of the kernel memory located at `KERNBASE`. Therefore, only the physical memory mapped to virtual addresses above `KERNBASE = 0xF0000000` can be used. Thus, the maximum supported physical memory is `256MB`.

Q5

PD: 1024 * 4 B = 4 KB

PT: 1024 * 1024 * 4 B = 4 MB

The total overhead is 4MB + 4KB = 4100 KB.

Q6

The statement `jmp *%eax` jumps to the address stored in `eax`, completing the jump here.

Since in `kern/entrypgdir.c`, the virtual addresses 0~4MB and KERNBASE ~ KERNBASE + 4MB are both mapped to the physical address 0~4MB, the code can be executed regardless of whether the EIP is in the high or low range.

This must be done. Because if only the high address is mapped, the system would not be compatible after enabling paging.

Challenge

Challenge! Extend the JOS kernel monitor with commands ...

This looked a bit simple at first, so I did this one first. However, while doing it, I encountered many small issues. Below are my specific solutions for a few problems.

Display in a useful and easy-to-read format all of the physical page mappings (or lack thereof) that apply to a particular range of virtual/linear addresses in the currently active address space. For example, you might enter `'showmappings 0x3000 0x5000'` to display the physical page mappings and corresponding permission bits that apply to the pages at virtual addresses 0x3000, 0x4000, and 0x5000.

This is essentially done by iterating over the page table as required.

I used `kern/monitor.c/check` to verify if the input addresses are valid; `kern/monitor.c/trans` translates the input string addresses into `uint32_t` type; `kern/monitor.c/mon_showmappings` aligns the addresses and then returns the physical page mappings and permission bits if the input addresses are valid. If no corresponding physical page exists, it will also report this. `check` and `trans` could have better implementations, but I ran out of time, so I didn't think too much about it.

Explicitly set, clear, or change the permissions of any mapping in the current address space.

Just check if the input is valid and modify the permissions as needed. See `kern/monitor.c/mon_showmappings` for details.

Dump the contents of a range of memory given either a virtual or physical address range. Be sure the dump code behaves correctly when the range extends across page boundaries!

Notice that we can directly access the corresponding physical address by accessing the virtual address above KERNBASE. Pay attention to the differences in pa and va.See `kern/monitor.c/mon_dumpmem` for details.