Report [Lab 2: Memory Management]

I chose the page size challenge

Part 1: Physical Page Management

Exercise 1

`boot_alloc()`

This function is used to allocate pages before free page list setup.

At the beginning of the function `nextfree` is initialized to point to the address just beyond kernel's BSS segment (above all kernel codes and global variables). So for each allocation I simply return the current `nextfree` and add `n` to it. The point is to remember to keep the pointer aligned to `PGSIZE`

Since `npages` is detected as the number of pages in total, I use this to determine whether there is enough space for current allocation.

```
    // LAB 2: Your code here.
    result = nextfree;
    nextfree = ROUNDUP(nextfree + n, PGSIZE);
    if ((uintptr_t)(nextfree - KERNBASE) / PGSIZE > npages)
    {
        panic("out of memory");
    }
    return result;
```

`mem_init()` (partial)

For now I just have to initialize the `pages` array to store information for each physical page and use `memset` to fill them with `\0`. Using the `boot_alloc` just implemented can simply achieve this.

```
    // Your code goes here:
    pages = (struct PageInfo *) boot_alloc(sizeof(struct PageInfo) * npages);
    memset(pages, 0, sizeof(struct PageInfo) * npages);
```

`page_init()`

For this part of exercise the essential thing is to determine which physical pages are already in use or not allowed to be allocated. Thankfully, the commands give clear hints. The physical page 0 is preserved for future use while IO hole `[IOPHYMEM, EXTPHYSMEM]` must never be allocated.

Then the question comes as which pages are already used by kernel and the data structures we just initialized. By `objdump` we can know that the kernel is loaded at physical address `0x100000`, which is surprisingly, exactly the same as `EXTPHYSMEM`.

And since the `next_free` is initialized as the first free virtual address above kernel's code and global variables and is then allocated sequentially, I inferred that the physical pages used by kernel and page data structure and those between `EXTPHYSMEM` and the highest memory we have allocated by `boot_alloc()`, which is the end of the array `pages`.

```
    size_t i;
    for (i = 1; i < npages; i++) {
        if (i >= IOPHYSMEM / PGSIZE && i < EXTPHYSMEM / PGSIZE)
            continue;
        if (i >= EXTPHYSMEM / PGSIZE && (uintptr_t)KADDR(i * PGSIZE) < (uintptr_t)(pages + npages))
            continue;
        pages[i].pp_ref = 0;
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
```

`page_alloc()`

Since the free page list is setup in `page_init()`, it becomes simply to allocate a free page as just to return the list's head `page_free_list`. Then update it by the result's `pp_link` and set the allocated page's `pp_link` to `NULL` to allow `page_free()` to check the double-free bug.

```
struct PageInfo *
page_alloc(int alloc_flags)
{
    // Fill this function in
    if (page_free_list == NULL)
        return NULL;
    struct PageInfo * result = page_free_list;
    page_free_list = result->pp_link;
    result->pp_link = NULL;
    if (alloc_flags & ALLOC_ZERO)
        memset(page2kva(result), '\0', PGSIZE);
    return result;
}
```

`page_free()`

Check whether the `pp_ref` is non-zero or `pp_link` is not `NULL` by the specification. Then add the freed page to the head of the free page list.

```
void
page_free(struct PageInfo *pp)
{
    // Fill this function in
    // Hint: You may want to panic if pp->pp_ref is nonzero or
    // pp->pp_link is not NULL.
    if (pp->pp_ref)
        panic("nonzero pp->pp_ref in page_free()");
    if (pp->pp_link != NULL)
        panic("double-free!!");
    pp->pp_link = page_free_list;
    page_free_list = pp;
}
```

testing

happy to pass the tests in one time.

Part 2: Virtual Memory

Exercise 2

I have studied concept of multi-level page table in ICS thoroughly so this part is not that hard for me. For 80386 architecture, the memory is managed by a 2-level page table with the first level called page directory, each containing 2^{10} entries, so that the virtual address space of size 2^{32} is managed as the page size is 4K.

The page is protected mainly by two bits U/S and R/W. The U/S bit indicates whether this page can be accessed by user-level procedures while the R/W bit indicates whether a page that is accessible for users is permitted to be written. To justify one's privilege to do some action on corresponding page must pass the check for both levels of page tables.

The segment-based memory management and protection are great ideas which complement some cons. of paging. But it's sad that this will introduce extra complexity. Maybe that's why we just use the so-called "Flat" Architecture[1] in JOS lab 2, which acts **like** segmentation is disabled.

Exercise 3

This exercise is about some useful qemu commands like `info pg`, `xp`, etc. And I think, they're really helpful.

Question 1

The `x` should have type `uintptr_t`.

Because we know that all C pointers indicate virtual addresses, and the `value` of type `char*` is cast to `x`, the content of `x` must also be a virtual address.

So the type of `x` should be `uintptr_t`.

Exercise 4

This exercise asks us to implement some functions about page management as the 2 previous exercises provide us with the knowledge and the tools to debug that we need.

`pgdir_walk()`

This function literally implement the theory "walking through the multilevel page tables using different bits of the virtual address". Thankfully, there are helpful macros at `inc/mmu.h` which extracts the index for the specific virtual address in the page directory and the corresponding page table so it can become really elegant to walk through the page tables regarding pointers as arrays.

```
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
{
    // Fill this function in
    struct PageInfo *pgtablePage;

    int pdx = PDX(va);
    if(~pgdir[pdx] & PTE_P) // relevant page table page doesn't exist yet.
    {
        if (!create)
        {
            return NULL;
        }
        if ((pgtablePage = page_alloc(ALLOC_ZERO)) == NULL)
        {
            return NULL; // allocation fails
        }
        pgtablePage->pp_ref++;

        // insert the new page table into the page directory
        // make the entry the most permissive
        pgdir[pdx] = page2pa(pgtablePage) | PTE_P | PTE_U | PTE_W;
    }
    else
    {
        pgtablePage = pa2page(PTE_ADDR(pgdir[pdx]));
    }

    // translate the page table page to kernel virtual address to be pointed by a C pointer
    pte_t *pgtable = page2kva(pgtablePage);
    int ptx = PTX(va);
    return pgtable + ptx;
}
```

`boot_map_region`

As we have just implemented `pgdir_walk`, this function becomes really simple. Simply use a loop and for every pages we use `pgdir_walk` to find the `pte` and then set up the mapping to the corresponding page frame. Just remember always to check the returning value.

```
static void
boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int
perm)
{
    // Fill this function in
    for (size_t i = 0; i < size; i += PGSIZE)
    {
        pte_t *pte = pgdir_walk(pgdir, (void *)(va + i), true);
        if (pte == NULL)
        {
            panic("out of space");
        }
        tlb_invalidate(pgdir, va + i); // invalid corresponding TLB entry
before modifying one PTE
        *pte = (pa + i) | perm | PTE_P;
    }
}
```

`page_lookup`

This is basically a wrapper function for `pgdir_walk`. Just call `pgdir_walk` to get the corresponding `pte` ; check its presence; then use `pa2page` to return a pointer to the desired `struct PageInfo`

```
struct PageInfo *
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
    // Fill this function in
    pte_t *pte = pgdir_walk(pgdir, va, false);
    if (pte == NULL || ~(*pte) & PTE_P) // no page mapped at va
    {
        return NULL;
    }
    if (pte_store)
    {
        *pte_store = pte;
    }
    return pa2page(PTE_ADDR(*pte));
}
```

`page_remove`

This function is used to unmap the physical page currently at virtual address `va` and this is a somehow delicate job. Use `page_lookup` to check whether there is any physical page at that address and if there is, clear all bits of corresponding `pte` and decrease the ref count of that physical page. Don't forget to invalidate TLB to prevent any out-dated entries there.

```
void
page_remove(pde_t *pgdir, void *va)
{
    // Fill this function in
    pte_t* pte;
```

```
    struct PageInfo * pg = page_lookup(pgdir, va, &pte);
    if (pg == NULL) // no page at va
    {
        return;
    }
    *pte = 0;
    tlb_invalidate(pgdir, va);
    page_decref(pg);
}
```

`page_insert`

This function inserts some physical page at some virtual address `va`. While we have to remove the current physical page at `va` if there is any, it becomes really dirty if the page to insert and the current page are the same. The key is to increase the ref count of `pp` before calling `page_remove` to prevent some page being unintentionally freed.

```
int
page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int perm)
{
    // Fill this function in
    pte_t *pte = pgdir_walk(pgdir, va, 1);
    if (pte == NULL)
    {
        return -E_NO_MEM;
    }
    pp->pp_ref++; // increase ref before removing to address the corner case
    page_remove(pgdir, va);
    *pte = page2pa(pp) | perm | PTE_P;
    return 0;
}
```

Part 3: Kernel Address Space

Exercise 5

This exercise requires us to implement some part of memory layout

1. user's read-only image of `pages`
2. kernel stack
3. kernel memory

All these are realized through `boot_map_region` function implemented at exercise 4. To use this function, I have to be clear about several things

1. the position of the virtual address to map
2. the position of the physical address to be mapped
3. the lengthy of the mapping region
4. the permission to give to such mapping

And these can be inferred by information in hints and graph in `memlayout.h`

```
    boot_map_region(kern_pgdir, UPAGES, PTSIZE, PADDR(pages), PTE_U | PTE_P);
    boot_map_region(kern_pgdir, KSTACKTOP - KSTKSIZE, KSTKSIZE,
PADDR(bootstack), PTE_W);
    boot_map_region(kern_pgdir, KERNBASE, (1ll << 32) - KERNBASE, 0, PTE_W);
```

Question 2

`info pg` instruction of `qemu` turns out to be quite useful to answer this question.

Page Directory Entry	Base Virtual	Points to (logically)
0x3bc	0xef000000	the `PageInfo` structure for each physical pages
0x3bd	0xefff8000	the kernel page directory itself
0x3bf	0xefc00000	the kernel stack
0x3c0~0x3ff	0xf0000000	all physical memory

Question 3

Because 80386 has memory protection mechanism by setting the permission bytes in the page's corresponding page directory entry and page table entry. So user programs are disabled to read or write the kernel's memory since the pages of which kernel virtual address consists have `PTE_U` bit cleared, which prevent the user to read or write them.

Question 4

Since `pages` are mapped at the region above `UPAGES` of size 4MB, there are at most 512K `struct PageInfo` structures because each of them takes up 8B. So the maximum amount of physical memory that this OS supports should be 512K * 4K = 2GB.

Question 5

The overhead could be at most 6MB + 4KB. If all physical pages are mapped, there needs 2GB / 4KB = 512K page table entries to manage them which takes 512K * 4B = 2MB memory. And of course there needs to be a page directory so that is the 4K. And the 4MB memory to hold up `pages` as discussed above.

Question 6

```
    jmp *%eax
```

This instruction makes us transition to running at an EIP above `KERNBASE`. Since we mapped both [0, 4MB) and [KERNBASE, KERNBASE + 4MB) to the physical address [0, 4MB), it's possible for us to run persistently between enabling paging and transitioning. This transition to make kernel run on higher address is necessary to leave the lower address for user programs to be run.

Challenge (page size)

The kernel address space 0xf000000~0xffffffff occupies 64K pages of size 4K so the overhead is roughly 256KB which can be considered large. But if we can enable the page size option which makes page size becomes 4M, there won't be any extra page tables at all, due to the mechanism to translate 4M pages.

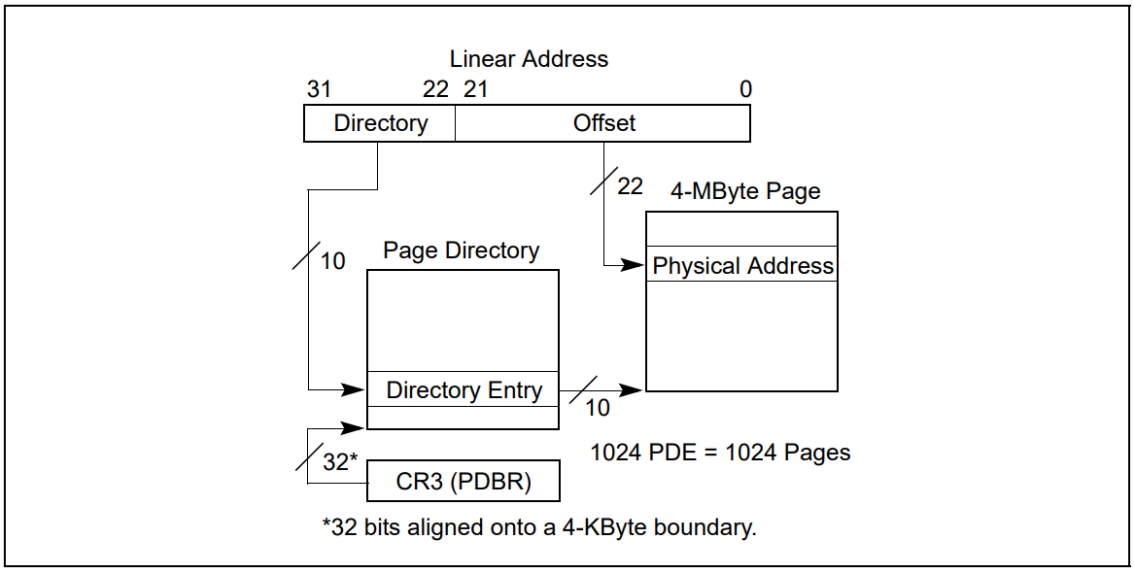

Figure 3-13. Linear Address Translation (4-M Byte Pages) [2]

But since 4M pages are not supported by all 80386 CPUs, we have to use `cpuid` instruction to check whether we can set the bit 4 of CR4 register to enable page size extension (PSE) first. The specification is

> CPUID.01H:EDX.PSE [bit 3] = 1

which means that we set `%eax` to be `0x01` before using `cpuid` then `PSE` is enabled if the bit 3 of `%edx` after the execution of `cpuid` is set.

```
bool SetPSE()
{
    unsigned int edx;
    __asm__ __volatile__ (
        "mov $1, %%eax\n\t"
        "cpuid\n\t"
        "mov %%edx, %0" : "=q" (edx) : : "%eax", "%edx");
    if (~edx & (1 << 3))
    {
        return false;
    }
    __asm__ __volatile__ (
        "mov %%cr4, %%eax\n\t"
        "xor $16, %%eax\n\t"
        "mov %%eax, %%cr4" : : : "%eax");
    tlbflush(); // flush TLB after changing PSE bit of CR4 by specification
    return true;
}
```

And if the check succeed, we use another version of `boot_map_region` called `boot_map_region_4M` to do the job of mapping kernel address.

```
static void
boot_map_region_4M(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa,
int perm)
{
    // check whether address and size are aligned to 4M boundary
    assert(va % (PGSIZE << 10) == 0);
    assert(size % (PGSIZE << 10) == 0);
    for (size_t i = 0; i < size; i += (PGSIZE << 10))
    {
        tlb_invalidate(pgdir, (void *)(va + i));
        pgdir[PDX(va + i)] = (pa + i) | perm | PTE_P | PTE_PS;
    }
}
```

And to pass the check, I have to change the logic of `check_va2pa` to reflect the different translation logic of 4M pages.

```
static physaddr_t
check_va2pa(pde_t *pgdir, uintptr_t va)
{
    pte_t *p;

    pgdir = &pgdir[PDX(va)];
    if (!(*pgdir & PTE_P))
        return ~0;
    if (*pgdir & PTE_PS) // 4M pages
    {
        return PTE_ADDR(*pgdir) + (PTX(va) << 12);
    }
    p = (pte_t*) KADDR(PTE_ADDR(*pgdir));
    if (!(p[PTX(va)] & PTE_P))
        return ~0;
    return PTE_ADDR(p[PTX(va)]);
}
```

This may seem suspicious. Luckily, qemu's `info pg` can help to justify my implementation.

```
(qemu) info pg
VPN range     Entry         Flags        Physical page
[ef000-ef3ff]  PDE[3bc]     -------UWP
  [ef000-ef3ff]  PTE[000-3ff] -------U-P 0011b-0051a
[ef400-ef7ff]  PDE[3bd]     -------U-P
  [ef7bc-ef7bc]  PTE[3bc]     -------UWP 003fd
  [ef7bd-ef7bd]  PTE[3bd]     -------U-P 0011a
  [ef7bf-ef7bf]  PTE[3bf]     -------UWP 003fe
  [ef7c0-ef7df]  PTE[3c0-3df] --SDA---WP 00000 00400 00800 00c00 01000 01400 ..
  [ef7e0-ef7ff]  PTE[3e0-3ff] --S-----WP 08000 08400 08800 08c00 09000 09400 ..
[efc00-effff]  PDE[3bf]     -------UWP
  [efff8-effff]  PTE[3f8-3ff] --------WP 0010e-00115
[f0000-f7fff]  PDE[3c0-3df] --SDA---WP 00000-07fff
[f8000-fffff]  PDE[3e0-3ff] --S-----WP 08000-0ffff
```

Reference

[1] 5.3.1 "Flat" Architecture

[2] Volume 3 of the current Intel manuals