Brandon Withington

CS444 Lab 2 Writeup :

4 / 23 / 2021

Exercise 1.)

```
Exercise 1. In the file kern/pmap.c, you must implement code for the following
functions (probably in the order given).

boot_alloc()
mem_init() (only up to the call to check_page_free_list(1))
page_init()
page_alloc()
page_free()

check_page_free_list() and check_page_alloc() test your physical page allocator. You
should boot JOS and see whether check_page_alloc() reports success. Fix your code so
that it passes. You may find it helpful to add your own assert()s to verify that your
assumptions are correct
```

Code :

```
boot_alloc :
```

```
    //print boot memory number
    //ROUNDUP would be used here as it expands x up to the nearest n
    //Within the print statement I should use the passed in n into this argument
    cprintf("boot_alloc memory at %x\n", nextfree);
    cprintf("next memory value is at %x\n", ROUNDUP((char *) (nextfree + n), PGSIZE));

    // If n>0, allocates enough pages of contiguous physical memory to hold 'n'
    // bytes.  Doesn't initialize the memory.  Returns a kernel virtual address.
    //

    if(n == 0){
        return nextfree;
    }

    result = nextfree;
    nextfree = ROUNDUP(nextfree + n, PGSIZE);
    if (PADDR(nextfree) > npages * PGSIZE){
        panic("bootalloc ran out of memory!");
    }
  return result;
}
```

```
Page_init :
```

```
size_t i;
    for (i = 1; i < npages_basemem; i++) {
        pages[i].pp_ref = 0;
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
    char* kernel_;
    extern char end[];
    //int test_memory = (int) ROUNDUP(((char*)pages) + (sizeof(struct PageInfo) * npages) - 0xf0000000, PG
SIZE)/PGSIZE;
    //cprintf("mem_allocated : %d\n", test_memory);
    //int test_memory = (int) ROUNDUP(((char*)pages) + (sizeof(struct PageInfo) * npages) - 0xf0000000, PG
SIZE)/PGSIZE;
    //cprintf("mem_allocated : %d\n", test_memory);
    //I think this works!
    //cprintf("test memory value : %d\n", test_memory);

    cprintf("size of PageInfo : %d\n", sizeof(struct PageInfo));
    cprintf("%x\n", ((char*)pages) + sizeof(struct PageInfo) * npages);

    kernel_ = ROUNDUP((ROUNDUP((char *) PADDR(end), PGSIZE) + PGSIZE) + sizeof(struct PageInfo) * npages,
PGSIZE);

    for (i = (uint32_t)kernel_/PGSIZE; i < npages; i++) {
        pages[i].pp_ref = 0;
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
```

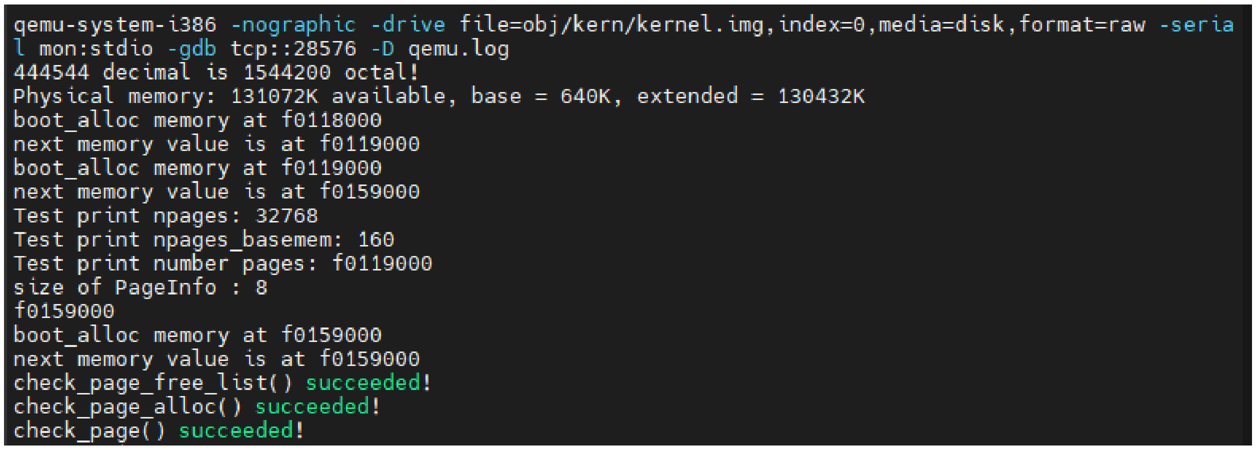
```
Page_alloc :
```

```
struct PageInfo *page = page_free_list;
    if(page){
        if (alloc_flags & ALLOC_ZERO){
            memset(page2kva(page), 0, PGSIZE);
        }
        page_free_list = page_free_list->pp_link;
        page->pp_link = NULL;
        return page;
    }
    return NULL;
```

```
Page_free :
```

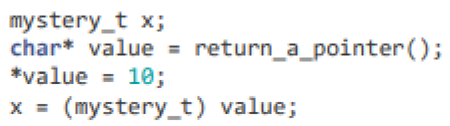
```
page_free(struct PageInfo *pp)
{
    // Fill this function in
    // Hint: You may want to panic if pp->pp_ref is nonzero or
    // pp->pp_link is not NULL.
//  if(pp->pp_link != NULL || pp->pp_ref != 0){
//      panic("panic message!!");
//  }
    //cprintf("pp->pp_link is : %x\n", pp->pp_link);
    if(pp->pp_link || pp->pp_ref){
        panic("panic message!! pp->pp_link is not null or is non zero\n");
    }

    pp->pp_link = page_free_list;
    page_free_list = pp;
}
```

Sample output :



```
qemu-system-i386 -nographic -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -seria
l mon:stdio -gdb tcp::28576 -D qemu.log
444544 decimal is 1544200 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
boot_alloc memory at f0118000
next memory value is at f0119000
boot_alloc memory at f0119000
next memory value is at f0159000
Test print npages: 32768
Test print npages_basemem: 160
Test print number pages: f0119000
size of PageInfo : 8
f0159000
boot_alloc memory at f0159000
next memory value is at f0159000
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
```

Question 1.) Assuming that the following JOS kernel code is correct, what type should variable x have, uintptr_t or physaddr_t?



```
mystery_t x;
char* value = return_a_pointer();
*value = 10;
x = (mystery_t) value;
```

My answer : The type of the X variable should be a uintptr_t, because of the char* that is being returned from the return_a_pointer function will be coming from a VA. Type casting it into a uintptr_t would make sense here.

Exercise 4.)

```
In the file kern/pmap.c, you must implement code for the following functions.

        pgdir_walk()
        boot_map_region()
        page_lookup()
        page_remove()
        page_insert()

check_page(), called from mem_init(), tests your page table management routines. You
should make sure it reports success before proceeding.
```

Code :

```
    uintptr_t page_address;
    struct PageInfo *page;
    pte_t *pointr;
    pte_t *ppointr;
    pde_t page_direc = pgdir[PDX(va)];
    if(page_direc == 0){
        /*pseudocode
        if no creation happens / is invoked
            return NULL

        else create the page
        page = page_alloc(1)
        if(page not made)
            return NULL

        else page->pp_ref += 1
        */
        if(!create){
            return NULL;
        }
        page = page_alloc(1);
        if(!page){
            return NULL;
        }
        page->pp_ref += 1;
        page_address = (uintptr_t)page2pa(page);
        pgdir[PDX(va)] = page_address | PTE_W | PTE_U | PTE_P;
        pointr  = (pte_t *)KADDR(page_address);
        ppointr = pointr + PTX(va);
    } else {
        pointr  = (pte_t *)KADDR(PTE_ADDR(page_direc));
        ppointr = pointr + PTX(va);
    }
    return ppointr;

}
```

Boot_map_region :

```
    // Fill this function in
    //for i less than SIZE / PGSIZE; i+=1; va += SIZE (or page size?); pa += PAGESIZE{
    //handle creation
    //Hint: the TA solution uses pgdir_walk
    //pte_t *PTE_VAL = pgdir_walk(pgdir, va, 1)
    //need some kind of way to handle when/if memory runs out during boot-map-region
    //}
    int i;
    uintptr_t n = va + size;
    //cprintf("V-Memory Address : %x  ||  mapped at P-Memory Address : %x\n", va, pa);
    for ( ; va < n; ++i, va += PGSIZE){
        pte_t *PTE_VAL = pgdir_walk(pgdir, (void *) va, 1);
        if (!PTE_VAL){
            panic("boot_map_region function is sending out a panic message, ran out of memory");
        }
        *PTE_VAL = pa | perm;
        pa += PGSIZE;
    }
    cprintf("New mapping : \n");
    cprintf("V-Memory Address : %x  ||  mapped at P-Memory Address : %x\n", va, pa);

}
```

Page lookup :

```
    pte_t *pointer_pgdirwalk = pgdir_walk(pgdir, va, 0);
    if(!pointer_pgdirwalk){
        return NULL;
    }
    if(!(*pointer_pgdirwalk & PTE_P)){
        return NULL;
    }
    if(pte_store){
        *pte_store = (pte_t *)pointer_pgdirwalk;
    }

    //return pa2page(*pte_store)??
    //return pa2page(pointer_pgdirwalk)??
    physaddr_t pag = PTE_ADDR(*pointer_pgdirwalk);
    return pa2page(pag);
```

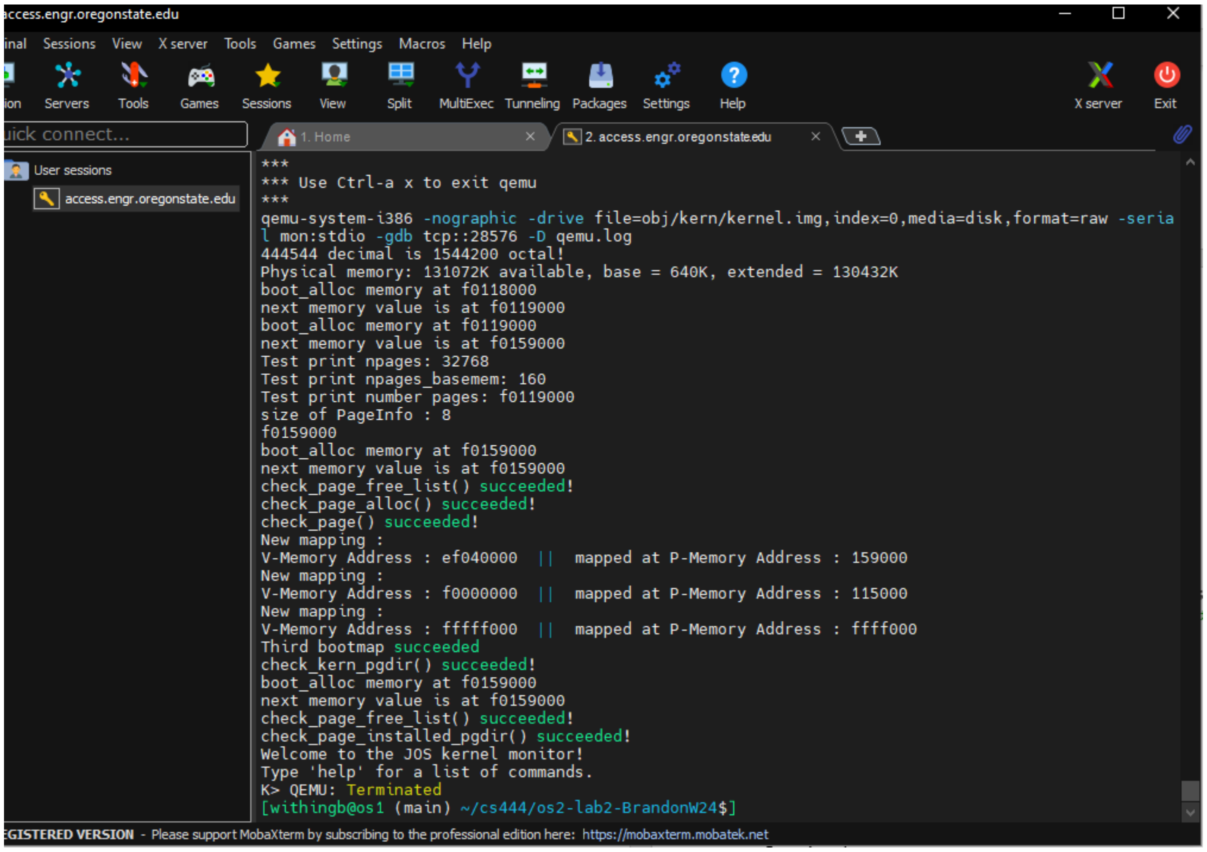
page_remove :

```
    // Fill this function in
    pte_t *pointer_t;
    struct PageInfo *page = page_lookup(pgdir, va, &pointer_t);
    // If there is no physical page at that address, silently does nothing.
    // Hint: The TA solution is implemented using page_lookup,
    //  tlb_invalidate, and page_decref.
    if(page){
        page_decref(page);
        *pointer_t = 0;
        //  hint use tlb_invalidate
        tlb_invalidate(pgdir, va);
    }
```

Page_insert :

```
    // Fill this function in
    pte_t *page = pgdir_walk(pgdir, va, 1);
    physaddr_t phys_page = page2pa(pp);
    //Covers the situation where a page table does not get allocated for whatever reason
    if(!page){
        return -E_NO_MEM;
    }
    // Hint: The TA solution is implemented using pgdir_walk, page_remove,
    // and page2pa.
    pp->pp_ref++;
    if (*page){
        page_remove(pgdir, va);
    }

    *page = phys_page | perm | PTE_P;
    //   - The TLB must be invalidated if a page was formerly present at 'va'.
    //tlb_invalidate(pgdir, va);
    return 0;
```

Sample output :


ccess.engr.oregonstate.edu
inal Sessions View X server Tools Games Settings Macros Help
ion Servers Tools Games Sessions View Split MultiExec Tunneling Packages Settings Help X server Exit
uick connect...
1. Home
2. access.engr.oregonstate.edu
User sessions
access.engr.oregonstate.edu
EGISTERED VERSION - Please support MobaXterm by subscribing to the professional edition here: https://mobaxterm.mobatek.net

```
***
*** Use Ctrl-a x to exit qemu
***
qemu-system-i386 -nographic -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -seria
l mon:stdio -gdb tcp::28576 -D qemu.log
444544 decimal is 1544200 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
boot_alloc memory at f0118000
next memory value is at f0119000
boot_alloc memory at f0119000
next memory value is at f0159000
Test print npages: 32768
Test print npages_basemem: 160
Test print number pages: f0119000
size of PageInfo : 8
f0159000
boot_alloc memory at f0159000
next memory value is at f0159000
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
New mapping :
V-Memory Address : ef040000  ||  mapped at P-Memory Address : 159000
New mapping :
V-Memory Address : f0000000  ||  mapped at P-Memory Address : 115000
New mapping :
V-Memory Address : fffff000  ||  mapped at P-Memory Address : ffff000
Third bootmap succeeded
check_kern_pgdir() succeeded!
boot_alloc memory at f0159000
next memory value is at f0159000
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> QEMU: Terminated
[withingb@os1 (main) ~/cs444/os2-lab2-BrandonW24$]
```

Question 2.)

Entry	**Base Virtual Address**	**Points to (logically)**
1023	0xffc00000	Where the page table for the top 4MB of physical memory is
1022	0xff800000	Where the page table for the second top portion 4MB of physical memory is
-------------------------------------	-------------------------------------	-------------------------------------
960	0xf0000000	Where the page table for first 4MB of physical memory is
959	0xefc00000	Where the page table for bootstack NOTE: the second half has not mapped here
958	0xef800000	Where the page table for memory-mapped for input and output is
957	0xef400000	Where the page table for second part of physical address of pages are
956	0xef000000	Where the page table for physical address of pages is located.
-------------------------------------	-------------------------------------	-------------------------------------
2	0x00800000	Not mapped
1	0x00400000	Not mapped
0	0x00000000	

Question 3.) We have placed the kernel and user environment in the same address space. Why will user programs not be able to read or write the kernel's memory? What specific mechanisms protect the kernel memory?

The program by the user cannot read into kernel memory as it requires the permission bit located within the page table entry. Virtual memory is also divided into segments ULIM and UTOP. The kernel part of the VA has the values PTE_P and PTE_W which allows the kernel to read or write. This is inplace to firstly protect the kernel's memory as well as to ensure it is not being tampered with. The specific mechanisms in place here are to protect and oversee which programs (or users) can write to virtual memory addresses. This in turn protects kernel memory.

Question 4.) What is the maximum amount of physical memory that this operating system can support? Why?

The maximum amount of physical memory that this OS can support is 256MB. This is because we are limited to 256MB because we map the kernel between 0xF0000000 and 0xFFFFFFFF (which is 256MB long).

Question 5.) How much space overhead is there for managing memory, if we actually had the maximum amount of physical memory? How is this overhead broken down?

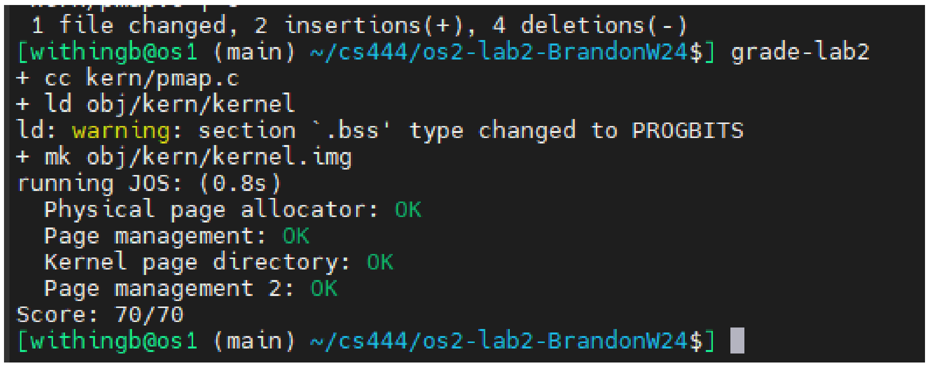
There is an up to 6MB + 4KB space overhead with our page directory requiring:

2^10 x 4 bytes = 4KB per PTE, this leads to it needing 4KB physical memory which per PT means 2^10 x 4KB = 4MB physical memory which then leads to 2GB / 4 MB which gives us about 512 page tables that takes up 512 x 4KB = 2MB leaving our pageinfo structure 4MB.

Question 6.) Revisit the page table setup in kern/entry.S and kern/entrypgdir.c. Immediately after we turn on paging, EIP is still a low number (a little over 1MB). At what point do we 12 transition to running at an EIP above KERNBASE? What makes it possible for us to continue executing at a low EIP between when we enable paging and when we begin running at an EIP above KERNBASE? Why is this transition necessary?

After the jump instruction found in entry.S we transition to running at VA kernel addresses. The lower 4MB of VA is mapped to the lower 4MB of physical address space, which allows user programs to utilize lower address spaces. This transition is necessary because if we had mapped the KERNBASE to KERNBASE+4MB virtual address then the program would crash trying to access memory that firstly was not allocated for that usage and indicate a triple fault.

RESULT :



```
 1 file changed, 2 insertions(+), 4 deletions(-)
[withingb@os1 (main) ~/cs444/os2-lab2-BrandonW24$] grade-lab2
+ cc kern/pmap.c
+ ld obj/kern/kernel
ld: warning: section `.bss' type changed to PROGBITS
+ mk obj/kern/kernel.img
running JOS: (0.8s)
  Physical page allocator: OK
  Page management: OK
  Kernel page directory: OK
  Page management 2: OK
Score: 70/70
[withingb@os1 (main) ~/cs444/os2-lab2-BrandonW24$] 
```