MIT 6.828: Operating System Engineering

Lab Report
Lab 2: Memory management

CSNLP
csnlp16@126.com
https://csnlp.github.io/

Contents

1 Introduction

Two components of memory management:

1. Physical memory allocator: allocates memory and then free memory.
2. Virtual memory: maps the virtual addresses to physical addresses.

2 Part 1: Physical Page Management

The OS must keep track of which parts of physical RAM are free and which are currently in use. In JOS, it keep tracks of which pages are free using a linked list **struct PageInfo** objects, each of which corresponds to a physical page.

2.1 Exercise 1

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

1. ***boot_alloc()**,*
2. ***mem_init()** (only up to the call to check_page_free_list(1)),*
3. ***page_init()**,*
4. ***page_alloc()**,*
5. ***page_free()**,*

*After your implementation, **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.*

2.1.1 The Analysis of mem_init() Function

Firstly, let's see the **mem_init()** function. The **mem_init()** is used for detect how many memory can be used.

```
// Set up a two-level page table:
//    kern_pgdir is its linear (virtual) address of the root
//
// This function only sets up the kernel part of the address space
// (ie. addresses >= UTOP).  The user part of the address space
// will be set up later.
//
// From UTOP to ULIM, the user is allowed to read but not write.
// Above ULIM the user cannot read or write.
void
mem_init(void)
{
  uint32_t cr0;
  size_t n;

```

```
	// Find out how much memory the machine has (npages & npages_basemem).
	i386_detect_memory();

	// Remove this line when you're ready to test this function.
	panic("mem_init: This function is not finished\n");

	//////////////////////////////////////////////////////////////////////
	// create initial page directory.
	kern_pgdir = (pde_t *) boot_alloc(PGSIZE);
	memset(kern_pgdir, 0, PGSIZE);

	//////////////////////////////////////////////////////////////////////
	// Recursively insert PD in itself as a page table, to form
	// a virtual page table at virtual address UVPT.
	// (For now, you don't have understand the greater purpose of the
	// following line.)

	// Permissions: kernel R, user R
	kern_pgdir[PDX(UVPT)] = PADDR(kern_pgdir) | PTE_U | PTE_P;

	//////////////////////////////////////////////////////////////////////
	// Allocate an array of npages 'struct PageInfo's and store it in 'pages'.
	// The kernel uses this array to keep track of physical pages: for
	// each physical page, there is a corresponding struct PageInfo in this
	// array.  'npages' is the number of physical pages in memory.  Use memset
	// to initialize all fields of each struct PageInfo to 0.
	// Your code goes here:


	//////////////////////////////////////////////////////////////////////
	// Now that we've allocated the initial kernel data structures, we set
	// up the list of free physical pages. Once we've done so, all further
	// memory management will go through the page_* functions. In
	// particular, we can now map memory using boot_map_region
	// or page_insert
	page_init();

	check_page_free_list(1);
	check_page_alloc();
	check_page();

	//////////////////////////////////////////////////////////////////////
	// Now we set up virtual memory

	//////////////////////////////////////////////////////////////////////
	// Map 'pages' read-only by the user at linear address UPAGES
	// Permissions:
	//    - the new image at UPAGES -- kernel R, user R
	//      (ie. perm = PTE_U | PTE_P)
	//    - pages itself -- kernel RW, user NONE
	// Your code goes here:

	//////////////////////////////////////////////////////////////////////
	// Use the physical memory that 'bootstack' refers to as the kernel
	// stack.  The kernel stack grows down from virtual address KSTACKTOP.
	// We consider the entire range from [KSTACKTOP-PTSIZE, KSTACKTOP)
	// to be the kernel stack, but break this into two pieces:
	//     * [KSTACKTOP-KSTKSIZE, KSTACKTOP) -- backed by physical memory
```

```
  //     * [KSTACKTOP-PTSIZE, KSTACKTOP-KSTKSIZE) -- not backed; so if
  //       the kernel overflows its stack, it will fault rather than
  //       overwrite memory.  Known as a "guard page".
  //     Permissions: kernel RW, user NONE
  // Your code goes here:

  //////////////////////////////////////////////////////////////////////
  // Map all of physical memory at KERNBASE.
  // Ie.  the VA range [KERNBASE, 2^32) should map to
  //      the PA range [0, 2^32 - KERNBASE)
  // We might not have 2^32 - KERNBASE bytes of physical memory, but
  // we just set up the mapping anyway.
  // Permissions: kernel RW, user NONE
  // Your code goes here:

  // Check that the initial page directory has been set up correctly.
  check_kern_pgdir();

  // Switch from the minimal entry page directory to the full kern_pgdir
  // page table we just created.  Our instruction pointer should be
  // somewhere between KERNBASE and KERNBASE+4MB right now, which is
  // mapped the same way by both page tables.
  //
  // If the machine reboots at this point, you've probably set up your
  // kern_pgdir wrong.
  lcr3(PADDR(kern_pgdir));

  check_page_free_list(0);

  // entry.S set the really important flags in cr0 (including enabling
  // paging).  Here we configure the rest of the flags that we care about.
  cr0 = rcr0();
  cr0 |= CR0_PE|CR0_PG|CR0_AM|CR0_WP|CR0_NE|CR0_MP;
  cr0 &= ~(CR0_TS|CR0_EM);
  lcr0(cr0);

  // Some more checks, only possible after kern_pgdir is installed.
  check_page_installed_pgdir();
}
```

Let's detail this process:

1. We can see that it calls the **i386_detect_memory()** function. Let's check this detect function's code:

```
static void
i386_detect_memory(void)
{
  size_t basemem, extmem, ext16mem, totalmem;

  // Use CMOS calls to measure available base & extended memory.
  // (CMOS calls return results in kilobytes.)
  basemem = nvram_read(NVRAM_BASELO);
  extmem = nvram_read(NVRAM_EXTLO);
  ext16mem = nvram_read(NVRAM_EXT16LO) * 64;

  // Calculate the number of physical pages available in both base
  // and extended memory.
  if (ext16mem)
```

```c
    totalmem = 16 * 1024 + ext16mem;
  else if (extmem)
    totalmem = 1 * 1024 + extmem;
  else
    totalmem = basemem;

  npages = totalmem / (PGSIZE / 1024);
  npages_basemem = basemem / (PGSIZE / 1024);

  cprintf("Physical memory: %uK available, base = %uK, extended = %uK\n",
    totalmem, basemem, totalmem - basemem);
}
```

Listing 1: i386_detect_memory() code

2. After counting the total memory and base memory, we come to the command as

```c
  //////////////////////////////////////////////////////////////////////
  // create initial page directory.
  kern_pgdir = (pde_t *) boot_alloc(PGSIZE);
  memset(kern_pgdir, 0, PGSIZE);
```

Listing 2: Create initial page directory.

3. We can see that we use boot_alloc() function. Let's check this function:

```c
// This simple physical memory allocator is used only while JOS is setting
// up its virtual memory system.  page_alloc() is the real allocator.
//
// 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, returns the address of the next free page without allocating
// anything.
//
// If we're out of memory, boot_alloc should panic.
// This function may ONLY be used during initialization,
// before the page_free_list list has been set up.
static void *
boot_alloc(uint32_t n)
{
        static char *nextfree;  // virtual address of next byte of free memory
        char *result;

        // Initialize nextfree if this is the first time.
        // 'end' is a magic symbol automatically generated by the linker,
        // which points to the end of the kernel's bss segment:
        // the first virtual address that the linker did *not* assign
        // to any kernel code or global variables.
        if (!nextfree) {
                extern char end[];
                nextfree = ROUNDUP((char *) end, PGSIZE);
        }

        // Allocate a chunk large enough to hold 'n' bytes, then update
        // nextfree.  Make sure nextfree is kept aligned
        // to a multiple of PGSIZE.
```

```c
	//
	// LAB 2: Your code here.
	result = nextfree;
	nextfree = ROUNDUP(next_free + n, PGSIZE);

	// decide if there is enough nextfree. npages * PGSIZE is
	// is all available space.
	if ((uint32_t)nextfree - KERNBASE > (npages * PGSIZE))
	    panic("NO ENOUGH MEMORY!!!\n");
	return result;
}
```

Listing 3: boot_alloc function

It involves the **ROUNDUP**.

```c
// Round up to the nearest multiple of n
#define ROUNDUP(a, n)                                           \
({                                                              \
	uint32_t __n = (uint32_t) (n);                          \
	(typeof(a)) (ROUNDDOWN((uint32_t) (a) + __n - 1, __n)); \
})
```

We can see that the **boot_alloc** allocate a page of size n which just follows the kernal.

Please notice that **memset()** is defined at lib/string.c.

```c
#if ASM
void *
memset(void *v, int c, size_t n)
{
	char *p;

	if (n == 0)
		return v;
	if ((int)v%4 == 0 && n%4 == 0) {
		c &= 0xFF;
		c = (c<<24)|(c<<16)|(c<<8)|c;
		asm volatile("cld; rep stosl\n"
			:: "D" (v), "a" (c), "c" (n/4)
			: "cc", "memory");
	} else
		asm volatile("cld; rep stosb\n"
			:: "D" (v), "a" (c), "c" (n)
			: "cc", "memory");
	return v;
}
```

```c
	//////////////////////////////////////////////////////////////////////
	// Recursively insert PD in itself as a page table, to form
	// a virtual page table at virtual address UVPT.
	// (For now, you don't have understand the greater purpose of the
	// following line.)

	// Permissions: kernel R, user R
	kern_pgdir[PDX(UVPT)] = PADDR(kern_pgdir) | PTE_U | PTE_P;
```

Listing 4: UVPT

I DO NOT REALLY FIGURE OUT THE MEANING OF THIS CODE!!

5. Set up the list of physical memory pages:

```
        //////////////////////////////////////////////////////////////////////
        // Allocate an array of npages 'struct PageInfo's and store it in 'pages'.
        // The kernel uses this array to keep track of physical pages: for
        // each physical page, there is a corresponding struct PageInfo in this
        // array.  'npages' is the number of physical pages in memory.  Use memset
        // to initialize all fields of each struct PageInfo to 0.
        // Your code goes here:
                // ***
                //My Code follows
                // ***
        pages = (struct PageInfo *) boot_alloc(npages * sizeof(struct PageInfo));
        memset(pages, 0, npages * sizeof(struct PageInfo));
```

Listing 5: My code about the set up of list of physical memory pages

6. The page_init() function: see in Section ?? .

2.1.2 The Analysis of page_init() Function

```
//
// Initialize page structure and memory free list.
// After this is done, NEVER use boot_alloc again.  ONLY use the page
// allocator functions below to allocate and deallocate physical
// memory via the page_free_list.
//
void
page_init(void)
{
        // The example code here marks all physical pages as free.
        // However this is not truly the case.  What memory is free?
        //  1) Mark physical page 0 as in use.
        //     This way we preserve the real-mode IDT and BIOS structures
        //     in case we ever need them.  (Currently we don't, but...)
        //  2) The rest of base memory, [PGSIZE, npages_basemem * PGSIZE)
        //     is free.
        //  3) Then comes the IO hole [IOPHYSMEM, EXTPHYSMEM), which must
        //     never be allocated.
        //  4) Then extended memory [EXTPHYSMEM, ...).
        //     Some of it is in use, some is free. Where is the kernel
        //     in physical memory?  Which pages are already in use for
        //     page tables and other data structures?
        //
        // Change the code to reflect this.
        // NB: DO NOT actually touch the physical memory corresponding to
        // free pages!
        size_t i;
        for (i = 0; i < npages; i++) {
                pages[i].pp_ref = 0;
                pages[i].pp_link = page_free_list;
```

```
                page_free_list = &pages[i];
        }
}
```

Listing 6: original code of page_init() function

We can have our finished page_init() function:

```
//
// Initialize page structure and memory free list.
// After this is done, NEVER use boot_alloc again.  ONLY use the page
// allocator functions below to allocate and deallocate physical
// memory via the page_free_list.
//
void
page_init(void)
{
  // The example code here marks all physical pages as free.
  // However this is not truly the case.  What memory is free?
  //  1) Mark physical page 0 as in use.
  //     This way we preserve the real-mode IDT and BIOS structures
  //     in case we ever need them.  (Currently we don't, but...)
  //  2) The rest of base memory, [PGSIZE, npages_basemem * PGSIZE)
  //     is free.
  //  3) Then comes the IO hole [IOPHYSMEM, EXTPHYSMEM), which must
  //     never be allocated.
  //  4) Then extended memory [EXTPHYSMEM, ...).
  //     Some of it is in use, some is free. Where is the kernel
  //     in physical memory?  Which pages are already in use for
  //     page tables and other data structures?
  //
  // Change the code to reflect this.
  // NB: DO NOT actually touch the physical memory corresponding to
  // free pages!
  // Let's begin our implementation:

  // (1). Mark physical page 0 as in use
  pages[0].pp_ref = 1;

  // (2). The rest of base memory, i.e., the memory between [PGSIZE, npages_basemem* PGSIZE)
  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];
  }

  // (3). IO hole [IOPHYSMEM, EXTPHYSMEM) must never be allocated.

  for(i=npages_basemem; i < EXTPHYSMEM/PGSIZE; i++) {
    pages[i].pp_ref = 1;
  }

  // (4). Extended memory: [EXTPHYSMEM, ...]
  // The followings are memory that being used in extended memory.
  physaddr_t first_free_addr = PADDR(boot_alloc(0));
  size_t first_free_page = first_free_addr/PGSIZE;
  for(i=EXTPHYSMEM/PGSIZE; i < first_free_page; i++) {
```

```
    pages[i].pp_ref = 1;
  }
  // The followings are memory that are free in extended memory.
  for(i=first_free_page; i < npages; i++) {
    pages[i].pp_ref = 0;
    pages[i].pp_link = page_free_list;
    page_free_list = &pages[i];
  }
```

2.1.3 Continue of mem_init() Function

After the page_init() function, the mem_init() function continues:

```
        check_page_free_list(1);
        check_page_alloc();
        check_page();
```

1. check_page_free_list() is to check that the pages on page_free_list are reasonable.

2. check_page_alloc() is to check the physical page allocator: page_alloc(), page_free(), and page_init(). Since we have implemented page_init(), let's implement the rest two functions.

2.1.4 The Analysis of page_alloc() Function

Let's firstly check its description about page_alloc()

```
//
// Allocates a physical page.  If (alloc_flags & ALLOC_ZERO), fills the entire
// returned physical page with '\0' bytes.  Does NOT increment the reference
// count of the page - the caller must do these if necessary (either explicitly
// or via page_insert).
//
// Be sure to set the pp_link field of the allocated page to NULL so
// page_free can check for double-free bugs.
//
// Returns NULL if out of free memory.
//
// Hint: use page2kva and memset
struct PageInfo *
page_alloc(int alloc_flags)
{
        // Fill this function in
        return 0;
}
```

We notice the hint the page2kva, memset, defined in kern/pmap.h may be useful: let see this two function:

```
static inline void*
page2kva(struct PageInfo *pp)
{
        return KADDR(page2pa(pp));
}
```

Listing 7: page2kva inline

.

- KADDR is about:

```
/* This macro takes a physical address and returns the corresponding kernel
 * virtual address.  It panics if you pass an invalid physical address. */
#define KADDR(pa) _kaddr(__FILE__, __LINE__, pa)

static inline void*
_kaddr(const char *file, int line, physaddr_t pa)
{
	if (PGNUM(pa) >= npages)
		_panic(file, line, "KADDR called with invalid pa %08lx", pa);
	return (void *)(pa + KERNBASE);
}
```

- page2pa is for return the physical address given PageInfo *.

```
static inline physaddr_t
page2pa(struct PageInfo *pp)
{
	return (pp - pages) << PGSHIFT;
}
```

Let's see the final implementation:

```
struct PageInfo *
page_alloc(int alloc_flags)
{
	// Fill this function in
	struct PageInfo *result;

	// Out of free memory
	if(page_free_list == NULL) {
		return NULL;
	}
	// if have free memory, return the Page that page_free_list points at.
	// move the page_free_list to next item in the linked list.
	result = page_free_list;
	page_free_list = result->pp_link;
	result->pp_link = NULL;

	// alloc_flags & ALLOC_ZERO
	if (alloc_flags & ALLOC_ZERO) {
		memset(page2kva(result), 0, PGSIZE);
	}
	return result;
	//return 0;
}
```

2.1.5 The Analysis of page_free Function

Finally, it's the page_free function: Let's see the original code and its hints:

```c
//
// Return a page to the free list.
// (This function should only be called when pp->pp_ref reaches 0.)
//
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.
}
```

We have two steps to do:

1. double check pp $\rightarrow$ pp_link and pp $\rightarrow$ pp_ref.
2. insert the *freed* page to free_page_list.

Now, we have our finished code

```c
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_link || pp->pp_ref) {
                panic("pp->pp_ref is nonzero or pp->pp_link is not NULL");
        }
        pp->pp_link = page_free_list;
        page_free_list = pp;
}
```

Then, we finally finish this part.

```
           Selector  +--------------+         +-----------+
          ---------->|              |         |           |
                     | Segmentation |         |  Paging   |
Software             |              |-------->|           |---------->  RAM
            Offset   |  Mechanism   |         | Mechanism |
          ---------->|              |         |           |
                     +--------------+         +-----------+
            Virtual                   Linear                Physical
```

Figure 1: The relationships of virtual address, linear address, and physical address.

3 Part 2: Virtual Memory

3.1 x86's Protected-mode Memory Management Architecture

3.1.1 Segmentation

3.1.2 Page Translation

3.2 Exercise 2

Look at chapters 5 and 6 of the Intel 80386 Reference Manual, if you haven't done so already. Read the sections about page translation and page-based protection closely (5.2 and 6.4). We recommend that you also skim the sections about segmentation; while JOS uses the paging hardware for virtual memory and protection, segment translation and segment-based protection cannot be disabled on the x86, so you will need a basic understanding of it.

3.3 The Relationships of Virtual Address, Linear Address, and Physical Address

In x86 terminology, a virtual address consists of a segment selector and an offset within the segment. A linear address is what you get after segment translation but before page translation. A physical address is what you finally get after both segment and page translation and what ultimately goes out on the hardware bus to your RAM.

3.4 Exercise 3

3.5 Virtual Address, Physical Address, and Their Interconversion

C type	Address type
uintptr_t	Virtual
physaddr_t	Physical

For code executing on the CPU, once we're in the protected mode, there's no way to directly use a linear address or physical address. **All memory reference are interpreted as virtual address and translated by the MMU, which means all pointer in C are virtual addresses.**

Let's see the interconversion:

```c
/* This macro takes a kernel virtual address -- an address that points above
 * KERNBASE, where the machine's maximum 256MB of physical memory is mapped --
 * and returns the corresponding physical address.  It panics if you pass it a
 * non-kernel virtual address.
 */
#define PADDR(kva) _paddr(__FILE__, __LINE__, kva)

static inline physaddr_t
_paddr(const char *file, int line, void *kva)
{
        if ((uint32_t)kva < KERNBASE)
                _panic(file, line, "PADDR called with invalid kva %08lx", kva);
        return (physaddr_t)kva - KERNBASE;
}

/* This macro takes a physical address and returns the corresponding kernel
 * virtual address.  It panics if you pass an invalid physical address. */
#define KADDR(pa) _kaddr(__FILE__, __LINE__, pa)

static inline void*
_kaddr(const char *file, int line, physaddr_t pa)
{
        if (PGNUM(pa) >= npages)
                _panic(file, line, "KADDR called with invalid pa %08lx", pa);
        return (void *)(pa + KERNBASE);
}
```

Listing 8: Interconversion of physical address and virtual address.

3.6 Reference Counting

A same physical page can be mapped at multiple virtual addresses simultaneously. The number of references to each physical page is kept in the **pp_ref** field of the **struct PageInfo** corresponding to the physical page. When **pp_ref** goes to zero for a physical page, the page can be freed since it is no longer being used.

3.7 Page Table Management

Exercise 4 is bout a set of routines to manage page tables: to insert and remove linear-to-physical mappings, to create page table pages when needed and so on.

3.7.1 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. Your should make sure it reports success before proceeding.*

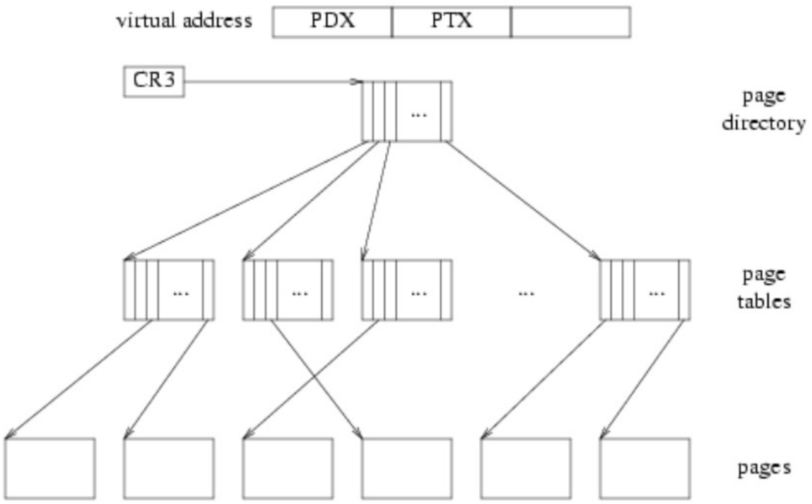
3.7.2 Some Useful Function in inc/mmu.h

```c
/*
 * This file contains definitions for the x86 memory management unit (MMU),
 * including paging- and segmentation-related data structures and constants,
 * the %cr0, %cr4, and %eflags registers, and traps.
 */

/*
 *
 *  Part 1.  Paging data structures and constants.
 *
 */

// A linear address 'la' has a three-part structure as follows:
//
// +--------10------+-------10-------+---------12----------+
// | Page Directory |   Page Table   | Offset within Page  |
// |      Index     |      Index     |                     |
// +----------------+----------------+---------------------+
//  \--- PDX(la) --/ \--- PTX(la) --/ \---- PGOFF(la) ----/
//  \---------- PGNUM(la) ----------/
//
// The PDX, PTX, PGOFF, and PGNUM macros decompose linear addresses as shown.
// To construct a linear address la from PDX(la), PTX(la), and PGOFF(la),
// use PGADDR(PDX(la), PTX(la), PGOFF(la)).

// page number field of address
#define PGNUM(la) (((uintptr_t) (la)) >> PTXSHIFT)

// page directory index
#define PDX(la)   ((((uintptr_t) (la)) >> PDXSHIFT) & 0x3FF)

// page table index
#define PTX(la)   ((((uintptr_t) (la)) >> PTXSHIFT) & 0x3FF)

// offset in page
#define PGOFF(la) (((uintptr_t) (la)) & 0xFFF)

// construct linear address from indexes and offset
#define PGADDR(d, t, o) ((void*) ((d) << PDXSHIFT | (t) << PTXSHIFT | (o)))
```

Let's get more information about **PDX** and **PTX**.

1. CR3 points at the page directory.
2. The PDX part of the address indexes into the page directory to give you a page table.
3. The PTX part indexed into the page table to give you a page.

However, the processor has no concept of page directories, page tables and pages. The processor just follows pointers:




```
pd = lcr3();
pt = *(pd + 4*PDX);
page = *(pt + 4*PTX);
```

3.7.3 pgdir_walk() Function

Let's firstly see its hints:

```
// Given 'pgdir', a pointer to a page directory, pgdir_walk returns
// a pointer to the page table entry (PTE) for linear address 'va'.
// This requires walking the two-level page table structure.
//
// The relevant page table page might not exist yet.
// If this is true, and create == false, then pgdir_walk returns NULL.
// Otherwise, pgdir_walk allocates a new page table page with page_alloc.
//    - If the allocation fails, pgdir_walk returns NULL.
//    - Otherwise, the new page's reference count is incremented,
//      the page is cleared,
//      and pgdir_walk returns a pointer into the new page table page.
//
// Hint 1: you can turn a PageInfo * into the physical address of the
// page it refers to with page2pa() from kern/pmap.h.
//
// Hint 2: the x86 MMU checks permission bits in both the page directory
// and the page table, so it's safe to leave permissions in the page
// directory more permissive than strictly necessary.
//
// Hint 3: look at inc/mmu.h for useful macros that manipulate page
// table and page directory entries.
//
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
{
        // Fill this function in
        return NULL;
}
```

See the implementation:

```c
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
{
	// Fill this function in
	assert(pgdir != NULL);
	unsigned int dir_offset = PDX(va);
	unsigned int page_offset = PTX(va);
	pte_t *page_table = NULL;
	pde_t *dir_page_entry = pgdir + dir_offset;

	struct PageInfo *new_page = NULL;

	bool pg_dir_entry_exists_flag = (*dir_page_entry) & PTE_P;
	if (!pg_dir_entry_exists_flag){
		if (!create)
			return NULL;
		// doesn't exist but want to create
		else {
			new_page = page_alloc(ALLOC_ZERO);
			if (new_page == NULL)
				return NULL;
			new_page->pp_ref++;
			*dir_page_entry = (page2pa(new_page)|PTE_P|PTE_W|PTE_U);
		}
	}
	// if exists, just return.
	page_table = KADDR(PTE_ADDR(*dir_page_entry));
	return &page_table[page_offset];
}
```

3.7.4 boot_map_region() Function

```c
//
// Map [va, va+size) of virtual address space to physical [pa, pa+size)
// in the page table rooted at pgdir.  Size is a multiple of PGSIZE, and
// va and pa are both page-aligned.
// Use permission bits perm|PTE_P for the entries.
//
// This function is only intended to set up the ``static'' mappings
// above UTOP. As such, it should *not* change the pp_ref field on the
// mapped pages.
//
// Hint: the TA solution uses pgdir_walk
static void
boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int perm)
{
	// Fill this function in

}
```

This function is only intended to set up the static mappings above UTOP.

```c
static void
boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int perm)
{
	// Fill this function in
```

```
        pte_t *entry = NULL;
        for (int i=0; i < size; i+=PGSIZE) {
                // get the entry corresponding to va.
                entry = pgdir_walk(pgdir, (void*)va, 1);
                *entry = (pa | perm | PTE_P);
                pa += PGSIZE;
                va += PGSIZE;
        }
}
```

3.7.5 page_insert() Function

```
//
// Map the physical page 'pp' at virtual address 'va'.
// The permissions (the low 12 bits) of the page table entry
// should be set to 'perm|PTE_P'.
//
// Requirements
//   - If there is already a page mapped at 'va', it should be page_remove()d.
//   - If necessary, on demand, a page table should be allocated and inserted
//     into 'pgdir'.
//   - pp->pp_ref should be incremented if the insertion succeeds.
//   - The TLB must be invalidated if a page was formerly present at 'va'.
//
// Corner-case hint: Make sure to consider what happens when the same
// pp is re-inserted at the same virtual address in the same pgdir.
// However, try not to distinguish this case in your code, as this
// frequently leads to subtle bugs; there's an elegant way to handle
// everything in one code path.
//
// RETURNS:
//   0 on success
//   -E_NO_MEM, if page table couldn't be allocated
//
// Hint: The TA solution is implemented using pgdir_walk, page_remove,
// and page2pa.
//
int
page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int perm)
{
  // Fill this function in
  return 0;
}
```

3.7.6 page_lookup() Function

The origin code of page_lookup() function

```
//
// Return the page mapped at virtual address 'va'.
// If pte_store is not zero, then we store in it the address
// of the pte for this page.  This is used by page_remove and
// can be used to verify page permissions for syscall arguments,
// but should not be used by most callers.
//
// Return NULL if there is no page mapped at va.
//
// Hint: the TA solution uses pgdir_walk and pa2page.
//
```

```
struct PageInfo *
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
        // Fill this function in
        return NULL;
}
```

```
struct PageInfo *
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
        // Fill this function in
        pte_t *pte_ptr = NULL;
        struct PageInfo *ret = NULL;

        // find the va's corresponding page entry.
        pte_ptr = pgdir_walk(pgdir, va, 0);

        // if there is no page mapped at va
        if (pte_ptr == NULL)
                return NULL;
        if (!(*entry & PTE_P))
                return NULL;

        ret = pa2page(PTE_ADDR(*pte_ptr));
        if (pte_store != 0) {
                *pte_store = pte_ptr;
        }
        return ret;
}
```

3.7.7 page_remove Function

```
//
// Unmaps the physical page at virtual address 'va'.
// If there is no physical page at that address, silently does nothing.
//
// Details:
//   - The ref count on the physical page should decrement.
//   - The physical page should be freed if the refcount reaches 0.
//   - The pg table entry corresponding to 'va' should be set to 0.
//     (if such a PTE exists)
//   - The TLB must be invalidated if you remove an entry from
//     the page table.
//
// Hint: The TA solution is implemented using page_lookup,
//      tlb_invalidate, and page_decref.
//
void
page_remove(pde_t *pgdir, void *va)
{
        // Fill this function in
}
```

Now, we have finished part 1 and part 2 in Lab 2.

4 Part 3: Kernal Address Space

JOS divides the processor's 32-bit linear address space into two parts:

1. The lower part: usually controlled by the user environment.
2. The upper part: usually controlled by the kernal.

The dividing line is defined somewhat arbitrarily by the symbol *ULIM* in **inc/memlayout.h**

4.1 Permissions and Fault Isolation

From low to top, there are mainly three parts:

1. **The address below *UTOP***: for the user environment to use. The user environment will set the permissions for accessing this memory.
2. **The address range [*UTOP, ULIM*)**: both the kernal and the user environment have the same permission: they can read but not write this address range.
3. **The address above *ULIM***: the user environment have no permission to any of this range of memory, while the kernal can be able to read and write this memory.

The user environment will have no permission to any of the memory above *ULIM*, while the kernal have the right to read and write this memory.

4.2 Initializing the Kernel Address Space

4.2.1 Exercise 5

*Fill in the missing code in **mem_init()** after the call to **check_page()**. Your code should now pass the **check_kern_pgdir()** and **check_page_installed_pgdir()** checks.*

Firstly see the hints:

```
        //////////////////////////////////////////////////////////////////////
        // Now we set up virtual memory

        //////////////////////////////////////////////////////////////////////
        // Map 'pages' read-only by the user at linear address UPAGES
        // Permissions:
        //    - the new image at UPAGES -- kernel R, user R
        //      (ie. perm = PTE_U | PTE_P)
        //    - pages itself -- kernel RW, user NONE
        // Your code goes here:

        //////////////////////////////////////////////////////////////////////
        // Use the physical memory that 'bootstack' refers to as the kernel
        // stack.  The kernel stack grows down from virtual address KSTACKTOP.
        // We consider the entire range from [KSTACKTOP-PTSIZE, KSTACKTOP)
        // to be the kernel stack, but break this into two pieces:
        //     * [KSTACKTOP-KSTKSIZE, KSTACKTOP) -- backed by physical memory
        //     * [KSTACKTOP-PTSIZE, KSTACKTOP-KSTKSIZE) -- not backed; so if
        //       the kernel overflows its stack, it will fault rather than
```

```
	//	overwrite memory.  Known as a "guard page".
	//     Permissions: kernel RW, user NONE
	// Your code goes here:

	//////////////////////////////////////////////////////////////////////
	// Map all of physical memory at KERNBASE.
	// Ie.  the VA range [KERNBASE, 2^32) should map to
	//      the PA range [0, 2^32 - KERNBASE)
	// We might not have 2^32 - KERNBASE bytes of physical memory, but
	// we just set up the mapping anyway.
	// Permissions: kernel RW, user NONE
	// Your code goes here:
```

We can see that we have three parts two finish:

1. Map 'pages' read-only by the user at linear address UPAGES
2. Use the physical memory that 'bootstack' refers to as the kernel stack.
3.

5 Reference

1. bysui's github and blog
   - github: https://github.com/bysui/mit6.828
   - blog: https://blog.csdn.net/bysui
2. SmallPond's github and blog
   - github: https://github.com/SmallPond/MIT6.828_OS
   - blog: https://me.csdn.net/Small_Pond
3. SimpCosm's github
   - github: https://github.com/SimpCosm/6.828
4. fatsheep9146's blog
   - blog: https://www.cnblogs.com/fatsheep9146/category/769143.html