JOS Lab 2: Memory Management

邓文 17307130171

Part 1 Physical Page Manageme：

The operating system must keep track of which memory areas are free and which are occupied. The JOS kernel manages memory with page as the smallest granularity, using MMU maps to protect each allocated chunk of memory.Here you will write a physical page allocator. It USES a linked list of structure PageInfo to record which pages are free, and each node in the linked list corresponds to a physical page.

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.

The operating system must keep track of which physical RAM is free and which is in use. This exercise focuses on writing the physical page allocator. It USES a linked list of PageInfo structures, one for each physical page, to record which pages are free. Because page table implementations need to allocate physical memory to store page tables, we need to write physical page allocators before virtual memory implementations.

1.1First, let's look at the function boot_alloc() :

```
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.
        //
        // LAB 2: Your code here.

        return NULL;
}
```

To write this function, you need to know what n is, what end is, what ROUNDUP macros are, and what nextfree is:
First of all, n is to allocate n bytes, which means we're now allocating n bytes, taking n bytes.
Second, end is a pointer to the end of the BSS segment of the kernel.
Go to the $ROOT/obj/kern directory, type objdump -h kernel, and look at the file structure to see that BSS is already at the end of the kernel. So, end is a pointer to the end of the BSS segment of the kernel, which is the first unused virtual memory address.
Third, the ROUNDUP macros.
Go to types.h in the $ROOT/inc directory to see the original definition of ROUNDUP.

```
// 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)); \
})
```

Defined as: to aggregate the nearest a Byte with multiple pages of size n. (Round up to the nearest multiple of n)
Finally, nextfree is the virtual address for the nextfree address.
The function begins with:

```
if (!nextfree) {
        extern char end[];
        nextfree = ROUNDUP((char *) end, PGSIZE);
}
```

If nextfree is used for the first time, initialize the variable, and if not, find the virtual address for the nextfree address. The way to find the virtual address for the next free address is to use PGSIZE pages to aggregate the content at the end address into bytes.
PGSIZE is the size of a physical page 4KB=4096B, defined in inc/mmu.h, and an important later constant PTSIZE, which corresponds to the actual physical memory size of a page table 1024*4KB=4MB
So what happens next is that nextfree isn't the first time it's been used:

```
        // LAB 2: Your code here.
        if (n == 0) return nextfree;
        result = nextfree;
        nextfree = KADDR(PADDR(ROUNDUP(nextfree + n, PGSIZE)));
        return result;
```

1) if the allocated n is 0, it means that there is no need to allocate, just output the pointer to nextfree.
2) if n allocated is not 0, then it means that there is allocated content, and n bytes are aggregated with pgsize-sized pages. Then the nextfree page (nextfree) is the page number from the current one plus the assigned page.
1.2 mem_init () function

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

        //////////////////////////////////////////////////////////////////////
        // 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:
        boot_map_region(kern_pgdir, UPAGES, PTSIZE, PADDR(pages), PTE_U);

        //////////////////////////////////////////////////////////////////////
        // 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:
        boot_map_region(kern_pgdir, KSTACKTOP - KSTKSIZE, KSTKSIZE,
PADDR(bootstack), PTE_W);
```

```
        //////////////////////////////////////////////////////////////////////
        // 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:
        boot_map_region_large(kern_pgdir, KERNBASE, -KERNBASE, 0, PTE_W);
        lcr4(rcr4() | CR4_PSE);

        // 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();
}
```

First of all, this function uses the data structure PageInfo just mentioned, so let's see what the specific definition of PageInfo is:

1.2.1 PageInfo structure used in the function

Enter $ROOT/inc/memlayout.h to see the definition of PageInfo:

```
struct PageInfo {
        // Next page on the free list.
        struct PageInfo *pp_link;

        // pp_ref is the count of pointers (usually in page table entries)
        // to this page, for pages allocated using page_alloc.
        // Pages allocated at boot time using pmap.c's
        // boot_alloc do not have valid reference count fields.

        uint16_t pp_ref;
};
```

This PageInfo contains two variables:
pp_link: points to the next free page.
pp_ref: this is the number of Pointers to the page. At startup time, Pages does not have a valid pointer, which is NULL.

1.2.2 initialize the home page

```
kern_pgdir = (pde_t *) boot_alloc(PGSIZE);
memset(kern_pgdir, 0, PGSIZE);
```

The header says that kern_pgdir is the header of the virtual address, so kern_pgdir is the pointer to the header of the page table for the first page assignment.
1.2.3 assign an array of npages sizes

The sizeof the allocated array is npages, and the data unit is PageInfo, so the space occupied is: npages*sizeof(struct PageInfo)
After the allocated data pointer is put into the pages variable, the following code can be written by imitating the steps of initializing the allocation of the first page in 1.2.2:

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

Only up to the call to check_page_free_list(1), so I won't go down first.

1.3 page_init() function

```
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];
        }
}
```

Initializes the page table structure and a linked list of free pages.
The code snippet already written in this function assumes that all physical pages are empty, which of course is impossible in practice.
So to initialize a list of free pages you need to know what memory is free.
The steps are as follows:
1) mark page 0 as being in use.
2) the remaining base memory is free.
3) IO holes cannot be allocated.
4) some of the expanded memory is used, some is free.

1.3.1 mark the initial page as in use

1.3.2 the remaining base memory is free

What is called basic memory, probably means that there is a part of the head empty. So just copy the code below and make a few changes.

1.3.3 IO holes cannot be assigned
What is the meaning of IO hole?
The notes say that the space between IOPHYSMEM and EXTPHYSMEM is the IO hole.

1.3.4 IO hole after some used, some idle

First, if you need a physical address to go to a virtual address, you can use KADDR(physical_address) or PADDR(virtual_address).

Secondly, comb the above a few, the first page to page is being used, not to be used, since the end of the second page to page Base can be used, then IO hole inside page shall not be used, then again, that is the extended memory mentioned in the article, some can be used, some idle, it is depends on where the critical point of use and shall not be used.

So we figured out that the location of the first page that could be used could be looked up using the boot_alloc() function, and if we wanted to find the location of the first page we could just write the parameter 0. Here boot_alloc() gets the first available virtual address, so you need to convert it to a physical address, using the PADDR macro.

So, this means that EXTPHYSMEM to PADDR(boot_alloc(0)) is not available and can be used later.

```
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;
        page_free_list = NULL;
        for (i = 0; i < 1; i++) {
                pages[i].pp_ref = 1;
                pages[i].pp_link = NULL;
        }
```

```
        for (; i < npages_basemem; i++) {
                pages[i].pp_ref = 0;
                pages[i].pp_link = page_free_list;
                page_free_list = &pages[i];
        }
        for (i = PTX(IOPHYSMEM); i < PTX(EXTPHYSMEM); i++) {
                pages[i].pp_ref = 1;
                pages[i].pp_link = NULL;
        }
        for (; i < PTX(PADDR(boot_alloc(0))); i++) {
                pages[i].pp_ref = 1;
                pages[i].pp_link = NULL;
        }
        for (; i < npages; i++) {
                pages[i].pp_ref = 0;
                pages[i].pp_link = page_free_list;
                page_free_list = &pages[i];
        }
}
```

1.4 page_alloc() function

Look at the comments first. The function is to assign the physical page.

1) this function takes alloc_flags as an argument, and populates all returned physical pages with '\0' if alloc_flags & ALLOC_ZERO.
2)

2) the function of page2kva is to get the virtual address of its kernel through the physical page, and the pp_link pointer should be set to NULL for the allocated page.

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

1.5 page_free() function

```
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("page_free: page's link is not NULL or ref count is
nonzero\n");

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

The release of the page
If pp_ref is not 0, pp_ref is not 0, then the page is not empty. If pp_ref is not NULL, then the list is empty.

If you exclude the above, then you can be free. The free step is to change the pp_link from NULL to the next available page, which is now page_free_list.

1.6 summarize

There are four functions written in this exercise. Boot_alloc (), mem_init (), page_init(), page_alloc(), page_free().

Let's look at these five functions vertically, but by name there are three types: boot, memory, and page.

Boot, boot level, boot_alloc() does the function of allocating N bytes of space at boot. That gives us some space.

Memory, memory level, mem_init() introduces the data structure Page_Info, which splits the incoming space into page levels in memory. I just allocated n bytes of space, so in this case it also corresponds to npage*sizeof(Page_Info) bytes, which is npage pages.

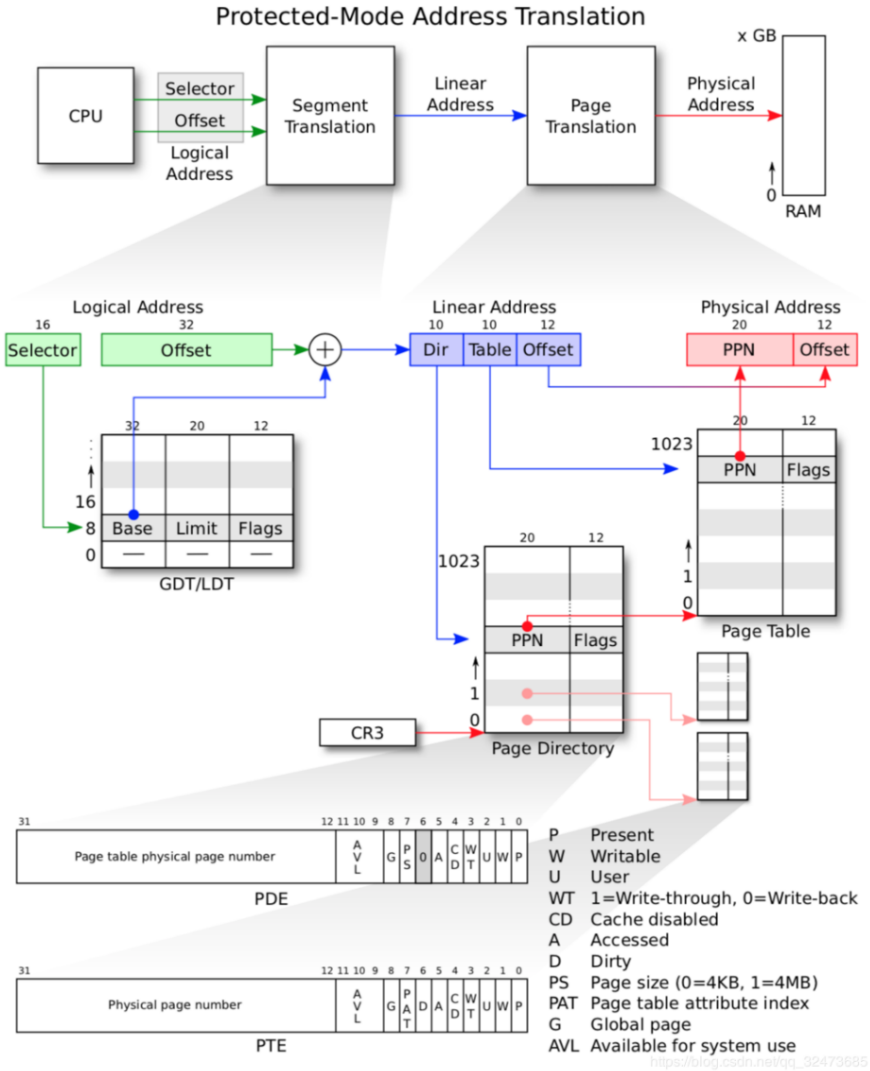
Page, page level, page_init() is the initialization of the page, which is to mark what is available and what is not, and to link what is available. Page_alloc () allocates the available pages, and page_free () frees the pages.

So, in fact, this experiment basically completes the memory-level initialization of the page and the basic operations of the page (including deletion and allocation).

Return $ROOT. Now make grade should be 20. The first exercise is complete.

Part 2 virtual memory：

2.1 conversion of physical memory and virtual memory



This diagram probably helps me review the conversion between physical memory and virtual memory:

The C pointer in our code is offset in the Virtual Address, which is converted to a linear Address (Virtual Address) by the segmentation mechanism through the descriptor table and the segment Selector, because the segment base set in JOS is 0, the linear Address is equal to offset. Before pagination is started, linear addresses are physical addresses. After we started paging, the linear address was converted into the physical address by the page conversion of the CPU's MMU unit.

After paging is started, when the processor encounters a linear address, its MMU component divides the address into three parts: the Directory, the Table, and the Offset. These three parts divide the original 32-bit linear address into three segments of 10+10+12. The size of each page table is 4KB (because the page is offset to 12 bits).

Example: now you want to convert the linear address 0xf011294c to a physical address. First, take the high 10 bits (page catalog item offset), which is 960(0x3c0), the middle 10 bits (page table item offset), which is 274(0x112), and the offset address is 1942(0x796).First, the processor retrieves the page directory through CR3 and retrieves item 960 of itRecord the entry, get the high 20 bit address of the page catalog entry,

thus get the first address of the corresponding page table physical page, get the 274 item page table entry in the page table again, and then get the first address of the page table entry, plus the low 12 bit offset address 1942 of the linear address, thus get the physical address.
As you can see from the above, each page table of contents has 1024 page items, each page item takes up 4 bytes, and a page table of contents takes up 4KB of memory. Each page entry points to a page table with 1024 page table entries, each of which also takes up 4 bytes, so the page directory and page table in JOS take up 1025 * 4KB = 4100KB about 4MB of memory. In general, we say that the virtual address space of each user process is 4GB, which means that each process has a page directory table, and the page directory address is loaded into the CR3 register when the process runs, so that each process can use up to 4GB of memory. In JOS, for simplicity, only one page table of contents is used, and the entire system's linear address space of 4GB is Shared by the kernel and all other user programs.
In paging management, page directories and page tables are stored in memory, and due to the CPU and memory speed mismatch, address translation is bound to reduce the efficiency of the system. To speed up address translation, x86 processors introduced the address translation cache TLB (bypass translation buffer) to cache recently translated addresses. Of course, caching then introduces the problem of inconsistent cache and in-memory page table contents, either by overloading CR3 to invalidate the entire TLB content or by using the invlpg directive.

2.2 uintptr_t and physaddr_t

uintptr_t represents the opaque virtual address, physaddr_t represents the physical address.
If you want to reference a virtual address, you need to first convert the type from an integer to a pointer type.

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;
```

Obviously, this should be uintptr_t, because data manipulation in the kernel is done at kernel virtual addresses.

2.3 the Reference Counting

The virtual memory space is mentioned here, including addresses such as UTOP. Firstly, the layout of virtual memory can be found in inc/memlayout.h.

```
* Virtual memory map:                                Permissions
*                                                   kernel/user
*
*    4 Gig --------->  +------------------------------+
*                      |                              | RW/--
*                      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
*                      :              .               :
*                      :              .               :
*                      :              .               :
*                      |~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~| RW/--
*                      |                              | RW/--
*                      |   Remapped Physical Memory   | RW/--
*                      |                              | RW/--
*    KERNBASE, ---->   +------------------------------+ 0xf0000000      --+
*    KSTACKTOP         |     CPU0's Kernel Stack      | RW/--  KSTKSIZE   |
*                      | - - - - - - - - - - - - - - -|                   |
*                      |      Invalid Memory (*)      | --/--  KSTKGAP    |
*                      +------------------------------+                   |
*                      |     CPU1's Kernel Stack      | RW/--  KSTKSIZE   |
*                      | - - - - - - - - - - - - - - -|                 PTSIZE
*                      |      Invalid Memory (*)      | --/--  KSTKGAP    |
*                      +------------------------------+                   |
*                      :              .               :                   |
*                      :              .               :                   |
*    MMIOLIM ------>   +------------------------------+ 0xefc00000      --+
*                      |       Memory-mapped I/O      | RW/--  PTSIZE
* ULIM, MMIOBASE -->   +------------------------------+ 0xef800000
*                      |  Cur. Page Table (User R-)   | R-/R-  PTSIZE
*    UVPT      ---->   +------------------------------+ 0xef400000
*                      |          RO PAGES            | R-/R-  PTSIZE
*    UPAGES    ---->   +------------------------------+ 0xef000000
*                      |           RO ENVS            | R-/R-  PTSIZE
* UTOP,UENVS ------>   +------------------------------+ 0xeec00000
* UXSTACKTOP -/        |     User Exception Stack     | RW/RW  PGSIZE
*                      +------------------------------+ 0xeebff000
*                      |       Empty Memory (*)       | --/--  PGSIZE
*    USTACKTOP  --->   +------------------------------+ 0xeebfe000
*                      |      Normal User Stack       | RW/RW  PGSIZE
*                      +------------------------------+ 0xeebfd000
*                      |                              |
*                      |                              |
*                      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
*                      .                              .
*                      .                              .
*                      .                              .
*                      |~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~|
*                      |     Program Data & Heap      |
*    UTEXT -------->   +------------------------------+ 0x00800000
*    PFTEMP ------->   |       Empty Memory (*)       |        PTSIZE
*                      |                              |
*    UTEMP -------->   +------------------------------+ 0x00400000      --+
*                      |       Empty Memory (*)       |                   |
*                      | - - - - - - - - - - - - - - -|                   |
*                      |  User STAB Data (optional)   |                 PTSIZE
*    USTABDATA ---->   +------------------------------+ 0x00200000        |
*                      |       Empty Memory (*)       |                   |
*    0 ------------>   +------------------------------+                   --+
*
* (*) Note: The kernel ensures that "Invalid Memory" is *never* mapped.
*     "Empty Memory" is normally unmapped, but user programs may map pages
*     there if desired.  JOS user programs map pages temporarily at UTEMP.
*/
```

You can find out from the page table structure how the virtual address corresponds to the physical address:

First, query in virtual memory space according to virtual address vaddr = UVPT[31:22] PDX|PTX|00, according to the translation mechanism of secondary page table:

1) page contents. The system first takes out the first 10 bits of vaddr, that is, UVPT[31:22], goes to the page directory to query, and obtains the page header

2) page table entry and offset address. Take the middle 10 bits of vaddr, PDX, and the offset after vaddr, and combine them to find the final page physical address in the page table.

As can be seen from the text, addresses from [0, ULIM] are allocated to the user and the rest to the operating system.

2.4 page table management

Exercise 4. In the file `kern/pmap.c`, you must implement code for the following functions.

```
pgdir_walk()
boot_map_region()
boot_map_region_large() //  Map all phy-mem at KERNBASE as large pages
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.

The code to write now in kern/ map.c is all about virtual and physical address translation.
Each physical Page corresponds to a Page structure and a physical Page Number PPN (Pysical Page Number) and a physical first address.
There are several definitions in map.h:

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

static inline struct PageInfo*
pa2page(physaddr_t pa)
{
        if (PGNUM(pa) >= npages)
                panic("pa2page called with invalid pa");
        return &pages[PGNUM(pa)];
}

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

A Page physical address is page2pa(*PageInfo),
The Page corresponding to a physical address is pa2page(physaddr_t),
Finally, the function that matches a Page to a virtual kernel address is page2kva(*PageInfo).
All three of the above functions can be used in the following functions.

2.4.1 pgdir_walk() function

```
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
{
        // Fill this function in
        pde_t *dir = &pgdir[PDX(va)];
        // Present bit = 0
        if (!(*dir & PTE_P)) {
                if (!create) return NULL;
                struct PageInfo *pinfo = page_alloc(ALLOC_ZERO);
                if (!pinfo) return NULL;
                pinfo->pp_ref += 1;
                physaddr_t paddr = page2pa(pinfo);
                *dir = paddr | PTE_P | PTE_W | PTE_U;
        }
        pte_t *pageTab  = (pte_t *)KADDR(PTE_ADDR(*dir));
        return &pageTab[PTX(va)];
}
```

This function is used to translate virtual addresses to physical addresses.
In tip 3 it says that inc/mmu.h has many macros that can be invoked, so you can start by looking at the contents of this file:

```
// 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) ----------/
```

This comment notes the use of the following macros:
PDX: a page directory index of a virtual address, also the first 10 bits of the address.
PTX: a page table index of a virtual address, also the middle 10 bits of the address.
PGOFF: the page offset of a virtual address, also the last 12 bits of the address.
Use methods such as comments can be known.

```
// Page table/directory entry flags.
#define PTE_P           0x001   // Present
#define PTE_W           0x002   // Writeable
#define PTE_U           0x004   // User
#define PTE_PWT         0x008   // Write-Through
#define PTE_PCD         0x010   // Cache-Disable
#define PTE_A           0x020   // Accessed
#define PTE_D           0x040   // Dirty
#define PTE_PS          0x080   // Page Size
#define PTE_G           0x100   // Global
```

This section represents the various states of the page directory and page table indexes. (exists, writable, user, no cache allowed, reachable, dirty read, page size, whether global, etc.)
According to the notes, the steps of the whole program are as follows:
When the page directory index does not exist within the va corresponding table items, namely virtual address without the corresponding physical address, need to create more assigns to judge whether a physical pages used as a secondary page table, this need to set the permissions, due to the level of page tables and secondary page table have access control, so the general approach is to relax level page table, mainly by the

secondary page table to control permissions, wrote in 2, attention should be paid to remove the page directory index (symbol) and a page table index (PTX) permission to ensure safe operation.
When there is a table entry corresponding to va in the page directory index, that is, the virtual address has a corresponding physical address, the number of Pointers to the page should be increased by one, the page should be emptied, and a pointer to a new page table page should be returned.

At the beginning, the function is the role of virtual address into a physical address process, and whether KADDR or page2kva are clearly a physical address or a physical page structure conversion virtual address function, why return will be a virtual address? This is because the conversion is now only specific to a page, if expressed in terms of address, then only accurate to the "page directory + page table." The address stored in the first-level page table is the physical address, while the return must be the virtual address and the permission bit must be removed. Recall the whole process:
1) determine the value corresponding to the address of the "page directory" of the virtual address, pgdir[page directory] is the process of entering the directory, if the page does not exist, but it can be created, then create a new empty page, the number of its pointer plus one;
2) when you cancel the permission, you cancel the permission limit in the physical page, which must be done in the physical address, so you use page2pa or PADDR (page2kva) to convert the newly created page to the physical address;
3) when returning the address, a combination of "page directory + page table" is returned. When looking for the address, the address storage of the page table is in the virtual memory, so the corresponding physical address in the page table should be found in the virtual memory. He actually found the location of a second level page table, the last level page table.

2.4.2 boot_map_region() function

```
..
// 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
        for (size_t s = 0; s < size; ) {
                pte_t *entry = pgdir_walk(pgdir, (const void *)(va + s), 1);
                if (!entry) {
                        cprintf("boot_map_region: page table not exists!\n");
                        return;
                }
                *entry = (pa + s) | perm | PTE_P;

                s += PGSIZE;
        }
}
```

The function pgdir_walk() is used to find the physical address of the page corresponding to the virtual address, so the function boot_map_region() is used to map the actual virtual address to the physical address page after finding the physical address. The mapping scope starts from va and ends with va+size. The size is n times of PGSIZE, so after finding the page corresponding to the virtual address, write the physical address into the page table address.

2.4.3 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
        pte_t *entry = pgdir_walk(pgdir, va, 0);
        if (pte_store) {
                *pte_store = entry;
        }
        if (!entry)
                return NULL;
        if (!(*entry & PTE_P)) {
                return NULL;
        }
        struct PageInfo *pp = pa2page(PTE_ADDR(*entry));
        return pp;
}
```

As understood from the comments, this function returns the page of the physical address mapped to the virtual address at va. If the third parameter, pte_store, is not 0, put the address it represents in the page.

2.4.4 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)
{
        pte_t *pte_store;
        struct PageInfo *pp = page_lookup(pgdir, va, &pte_store);
        // no physical page
        if (!pp)
                return;
        page_decref(pp);
        tlb_invalidate(pgdir, va);
        *pte_store = 0;
```

The purpose of this function is to cancel the virtual address of the physical address map. If there is no corresponding virtual address you do nothing.
Specific measures are as follows:
1) reduce the number of pages mapped by physical addresses. (use page_lookup to find the physical address corresponding to the va virtual address before proceeding)
2) the physical page should be released. (implemented with page_decref)
3) the entry of the page table corresponding to the va virtual address should be set to 0.
4) TLB translation cache must become unavailable if page table entry is removed. (implemented with tlb_invalidate)

2.4.5 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
        pte_t *entry = pgdir_walk(pgdir, va, 1);
        if (!entry)
                return -E_NO_MEM;

        pp->pp_ref++;

        if (*entry & PTE_P) {
                page_remove(pgdir, va);
        }
        *entry = page2pa(pp) | perm | PTE_P;

        return 0;
}
```

The purpose of this function is to map the situation of the virtual address to the physical page information pp.
Steps as notes:
1) if the virtual address va already has a mapped physical page, it should first remove

the map and implement it with page_remove.
2) page tables are allocated and inserted into pgdir if necessary.
3) if the insertion is successful, the number of references in the structure should increase.
4) if the address of the page already exists is va, then TLB must be invalidated, with tlb_invalidate.

2.4.6 boot_map_region_large() function

```
// 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 *PTSIZE*.
// Use permission bits perm|PTE_P|PTE_PS 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_large(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa,
int perm)
{
        // Fill this function in
        for (size_t s = 0; s < size; s += PTSIZE) {
                pde_t *entry = &pgdir[PDX(va + s)];
                //  Page directory entry points to a page table
                if ((*entry & PTE_P) && !(*entry & PTE_PS)) {
                        cprintf("boot_map_region_large: Page table is
overlapped.\n");
                }
                *entry = (pa + s) | perm | PTE_P | PTE_PS;
        }
}
```

We consume a lot of physical pages to record the mapping to KERNBASE. Use the PTE_PS ("Page Size") bit for a more spatially efficient mapping of the top 256MB pd.
reference document
Implement the boot_map_region_large() function and enable PSE(page size extension) before loading cr3. Then call boot_map_region_large() to map KERNBASE.
You can see in chapter 3.9 of the reference documentation that it becomes a partition [31..22][21..0], a page becomes a 4MB page, and see PSE flag (bit 4) in control register CR4 and the PS flag in PDE -- Set to 1 to enable the page size extension for 4-mbyte pages.
So our page-walking table is going to be just cr3-> pd-> PAGE,
That's three steps
Boot_map_region_large where there is only one layer of PD on top which is PDE pointing directly to the page
Modify the PSE
Replace the original KERNBASE boot_map_region function

Open the $ROOT/kern/entry. S

2.5 summarize

This exercise focuses on the translation of virtual and physical addresses and the mapping between page tables. It contains six functions.
pgdir_walk () is the most basic function, which mainly finds the physical address of the corresponding page table based on the virtual address.
boot_map_region () is used to map a contiguous virtual address to its corresponding physical address. For example, [va, va+size] is mapped to [pa, pa+size].
boot_map_region_large () is a function that appears to be related to the previous function
page_lookup () returns the page structure of the physical address that has been mapped to the virtual address at the va;
page_remove () is used to unbind the mapping relation between physical address and virtual address.
page_insert () binds a virtual address at the va to a physical address, and if a relationship exists, removes it before binding.
As can be seen from the above operations, these functions are actually completed a series of physical address and virtual address correspondence.

Part 3 Kernel address space

3.1 the actual correspondence between virtual address and physical address and the authorization granted

Here, a function is used to map the virtual address to the physical address:

Boot_map_region () is used to map a contiguous virtual address to its corresponding physical address. For example, [va, va+size] is mapped to [pa, pa+size].

```
static void
boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int
perm)
```

Take a look at its parameters. The first is the directory of the page, the second is the virtual address, the third is the size of the mapping range, the fourth is the corresponding physical address, and the fifth is the permissions granted.

So let's look at the first part of the mapping:

The mapping for this section starts with UPAGES, size is PTSIZE, physical address is PADDR(pages), and permission is PTE_U|PTE_P, not PTE_P because the function itself has given this permission by default.

```
//////////////////////////////////////////////////////////////////////
// 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:
boot_map_region(kern_pgdir, UPAGES, PTSIZE, PADDR(pages), PTE_U);
```

The mapping of the second part is as follows:

The mapping in this section starts with KSTACKTOP-KSTKSIZE, the size is KSTKSIZE, the physical address is PADDR, and the permission is PTE_W, which is not written here because the function itself has given this permission by default.

```
        //////////////////////////////////////////////////////////////////////
        // 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:
        boot_map_region(kern_pgdir, KSTACKTOP - KSTKSIZE, KSTKSIZE,
PADDR(bootstack), PTE_W);
```

The mapping of the third part is as follows:

This part of the map, the starting point is KERNBASE, size is 2^32 - KERNBASE, physical address is 0, permission is PTE_W, PTE_P is not written here because the function itself has given this permission by default.

```
        //////////////////////////////////////////////////////////////////////
        // 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:
        boot_map_region_large(kern_pgdir, KERNBASE, -KERNBASE, 0, PTE_W);
        lcr4(rcr4() | CR4_PSE);
```

Run make grade

```
boot block is 390 bytes (max 510)
+ mk obj/kern/kernel.img
make[1]: Leaving directory '/home/jos/17307130171/jos01/jos-2019-spring'
running JOS: (0.7s)
  Physical page allocator: OK
  Page management: OK
  Kernel page directory: OK
  Page management 2: OK
  Large kernel page size: OK
Score: 80/80
```

3.2 some questions about mapping

Question

2. What entries (rows) in the page directory have been filled in at this point? What addresses do they map and where do they point? In other words, fill out this table as much as possible:

Entry	Base Virtual Address	Points to (logically):
1023	?	Page table for top 4MB of phys memory
1022	?	?
.	?	?
.	?	?
.	?	?
2	0x00800000	?
1	0x00400000	?
0	0x00000000	[see next question]

3. (From Lecture 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?

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

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?

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 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?

1) the first problem is to find the corresponding virtual address of the page and the corresponding physical address range of the map. So you have to go back to the structure of the virtual address, where the first 10 bits are pages, so you have 1024 entries, each of which is 2^10 pages, and you end up with 2^12 entries per page. A page can map 4MB of physical memory (PTSIZE = 4MB).

2) since the permission bit can be set in the page table, if PTE_U is not set to 0, the user has no permission to read and write.

3) all free physical pages are initially stored in an array such as pages, the basic unit of which is PageInfo, so the sizeof a PageInfo is sizeof(struct PageInfo) =8 Byte, that is to say, 512K pages at most, each page capacity 2 ^ 12 = 4K. So the 512 k * 4 k = 2 gb.

4) just from the previous question, the size of the array composed of all PageInfo

structures is 4MB, with a maximum of 512k pages, so there are 0.5m pages, a 32-bit address information, that is 4B, so the page information should be recorded with 4B* 0.5m =2MB, and the directory entry itself is 4K, so 6MB+4KB.

5) statement jmp *%eax that is to go to the eax address execution, here completed the jump. Relocated code is mainly set up the stack pointer and invoke the kern/init.c. Since virtual addresses of 0~4MB and KERNBASE ~ KERNBASE + 4MB are mapped to physical addresses of 0~4MB in kern/entrypgdir.c, eips can be executed at both high and low levels. This is necessary because if only the high address is mapped, then the next statement that opens the pagination mechanism will crash.

3.3 Challenge

Challenge! We consumed many physical pages to hold the page tables for the KERNBASE mapping. Do a more space-efficient job using the PTE_PS ("Page Size") bit in the page directory entries. This bit was *not* supported in the original 80386, but is supported on more recent x86 processors. You will therefore have to refer to Volume 3 of the current Intel manuals. Make sure you design the kernel to use this optimization only on processors that support it!

Challenge! Extend the JOS kernel monitor with commands to:

- 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.
- Explicitly set, clear, or change the permissions of any mapping in the current address space.
- 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!
- Do anything else that you think might be useful later for debugging the kernel. (There's a good chance it will be!)

First, the form is entered as a string, and we need to convert it to a numeric address.

```
uint32_t xtoi(char* buf) {
        uint32_t res = 0;
        buf += 2; //0x...
        while (*buf) {
                if (*buf >= 'a') *buf = *buf-'a'+'0'+10;//aha
                res = res*16 + *buf - '0';
                ++buf;
        }
        return res;
}
```

Write another function to format the output:

```
void pprint(pte_t *pte) {
        cprintf("PTE_P: %x, PTE_W: %x, PTE_U: %x\n",
                *pte&PTE_P, *pte&PTE_W, *pte&PTE_U);
}
```

Finally:

```
int
showmappings(int argc, char **argv, struct Trapframe *tf)
{
        if (argc == 1) {
                cprintf("Usage: showmappings 0xbegin_addr 0xend_addr\n");
                return 0;
        }
        uint32_t begin = xtoi(argv[1]), end = xtoi(argv[2]);
        cprintf("begin: %x, end: %x\n", begin, end);

        for (; begin <= end; begin += PGSIZE) {
                pte_t *pte = pgdir_walk(kern_pgdir, (void *) begin, 1); //create
                if (!pte) panic("boot_map_region panic, out of memory");
                if (*pte & PTE_P) {
                        cprintf("page %x with ", begin);
                        pprint(pte);
                } else cprintf("page not exist: %x\n", begin);
        }
        return 0;
}
```

and

```
static struct Command commands[] = {
        { "help", "Display this list of commands", mon_help },
        { "kerninfo", "Display information about the kernel", mon_kerninfo },
        {       "backtrace", "Backtrace calling stack", mon_backtrace },
        { "time", "Display running time of the command", mon_time},
        { "showmappings", "Show mappings between physical addr and virtual
addr", showmappings },
};
```

Run make qemu

```
K> showmappings
Usage: showmappings 0xbegin_addr 0xend_addr
K> showmappings 0xf011a000 0xf012a000
begin: f011a000, end: f012a000
page f011a000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f011b000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f011c000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f011d000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f011e000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f011f000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f0120000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f0121000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f0122000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f0123000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
```

```
page f0124000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f0125000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f0126000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f0127000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f0128000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f0129000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f012a000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
K> showmappings 0xef000000 0xef010000
begin: ef000000, end: ef010000
page ef000000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef001000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef002000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef003000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef004000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef005000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef006000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef007000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef008000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef009000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef00a000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef00b000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef00c000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef00d000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef00e000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef00f000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef010000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
K>
```

At this we have finished the lab. Thank TA.