Lab 2: Memory Management

Extremely Useful link for the Lab:
http://plafer.github.io/2017/07/22/jos_mem/

Introduction

In this lab, you will write the memory management code for your operating system. Memory management has two components.

The first component is a physical memory allocator for the kernel, so that the kernel can allocate memory and later free it. Your allocator will operate in units of 4096 bytes, called pages. Your task will be to maintain data structures that record which physical pages are free and which are allocated, and how many processes are sharing each allocated page. You will also write the routines to allocate and free pages of memory.

The second component of memory management is virtual memory, which maps the virtual addresses used by kernel and user software to addresses in physical memory. The x86 hardware's memory management unit (MMU) performs the mapping when instructions use memory, consulting a set of page tables.

*The **git checkout -b** command shown above actually does two things: it first creates a local branch lab2 that is based on the origin/lab2 branch provided by the course staff, and second, it changes the contents of your lab directory to reflect the files stored on the lab2 branch. Git allows switching between existing branches using **git checkout branch-name**, though you should commit any outstanding changes on one branch before switching to a different one.*

*In some cases, Git may not be able to figure out how to merge your changes with the new lab assignment (e.g. if you modified some of the code that is changed in the second lab assignment). In that case, the **git merge** command will tell you which files are conflicted, and you should first resolve the conflict (by editing the relevant files) and then commit the resulting files with **git commit -a**.*
Lab 2 contains the following new source files, which you should browse through:

- *inc/memlayout.h*
- *kern/pmap.c*
- *kern/pmap.h*
- *kern/kclock.h*
- *kern/kclock.c*

memlayout.h describes the layout of the virtual address space that you must implement by modifying pmap.c. memlayout.h and pmap.h define the PageInfo structure that you'll use to keep track of which pages of physical memory are free. kclock.c and kclock.h manipulate the PC's battery-backed clock and CMOS RAM hardware, in which the BIOS records the amount of physical memory the PC contains, among other things. The code in pmap.c needs to read this device hardware in order to figure out how much physical memory there is

Part 1: Physical Page Management

The operating system must keep track of which parts of physical RAM are free and which are currently in use. JOS manages the PC's physical memory with page granularity so that it can use the MMU to map and protect each piece of allocated memory.

You'll now write the physical page allocator. It keeps track of which pages are free with a linked list of struct PageInfo objects (which, unlike xv6, are not embedded in the free pages themselves), each corresponding to a physical page. You need to write the physical page allocator before you can write the rest of the virtual memory implementation, because your page table management code will need to allocate physical memory in which to store page tables.

OVERVIEW:

As we can see from entry_pgdir, we have the first 4 MB of the RAM available for us to use mapped starting from virtual address 0xf0000000 to 0xf03fffff. We'd like to start allocating memory for us to use for dynamically allocated data structures, but we have several problems:

1. *we don't have something like a malloc for us to use to "dynamically allocate stuff on the heap". This means we'll need to create our own allocator.*
2. *The kernel code and data is already present in RAM and we need to make sure not to overwrite this data by accident i.e. We need to keep track of which addresses have already been allocated, and which ones are free.*

All of those problem are addressed by the boot_alloc function. This will be our "temporary" memory allocator which we'll use to start allocating data structures dynamically. It is temporary because it can manage only the first 4MB of RAM.
Now that we have a memory allocator, we'll allocate an array of 1024 entries for our page directory kern_pgdir. This directory will be the new page directory that is meant to replace the temporary entry_pgdir provided to us in the first lab. This is because the x86 hardware mandates page alignment for the page directory, addresses returned by boot_alloc function must be page aligned.
*To manage the memory and avoid overwriting, we need to keep track of which (physical) page frames are allocated and which ones are free. Each page frame will have its own PageInfo struct that will also store some metadata, and those stucts will form a linked list. The amount of page frames we'll be managing depends on how much RAM we have in the computer, so we'll use boot_alloc to allocate an array of PageInfo proportional to the amount of RAM available, and will call this array struct PageInfo *pages. For example, if we have 131072K RAM, and each page is 4K, then we'll need an array of 32,768 PageInfo elements.*
*The head of the linked list is struct PageInfo *page_free_list and what we'll need to do next is to add all the free frames to this linked list. When adding the free page frames, it is important to first add the free page frames from the original 4MB region managed by boot_alloc.*
Once we are done with this, we will no longer need to use boot_alloc in order to allocate memory because page_free_list will now contain all the frames in the

original 4MB region that were not yet allocated by boot_alloc. In order to effectively use page_free_list, we'll create a function called page_alloc that "mark" a page frame as in use by popping it off the page_free_list.
The way we can access the data stored in a page frame directly is through the page2kva macro. This is how page2kva works:

1. *let's say we popped a struct PageInfo from the page_free_list (let's call it pp). How do we know the physical address that pp corresponds to? We can calculate it by finding what index pp is in the pages array and multiply it by the page size: (pp - pages)*4096.*
2. *But the processor can't address physical address directly, how knowing the physical address help us? Each page frame in the original 4MB region is mapped at virtual address located at an offset of 0xf0000000. So, for example, if we want to write to a page frame located in the physical address 0x01234000, we'll need to tell the processor to write to the virtual address 0xf1234000. Again, this trick works only for the page frames located in the original 4MB region of memory that has mapped by entry_pgdir.*

*Then we proceed with mapping the various memory regions as specified in the comments of mem_init. One interesting thing to note is that mem_init tells us to do is map the (physical) RAM addresses from 0x00000000 - 0x0fffffff into the virtual addresses 0xf0000000-0xffffffff (that's 256 MB). To calculate the memory overhead incurred by the 256 MB of memory: 268435456 bytes is 65536 page frames (divide by 4096). Each frame occupies 4 bytes in a page table entry. So this means we'll need 65536 * 4 = 262,144 bytes to store all the page tables of these mappings.*
Total Nu0mber of page tables needed: 65536 (total frames) / 1024 (frames mapped in each page table) = 64 page tables.

Lastly, load the physical address of kern_pgdir into cr3 register, and our new page directory is installed.

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.

```
// 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)
{
/*
Addresses lower than nextfree are considered allocated and the addresses equal to
and greater than next free are considered to be free. The end symbol which has
been utilized with keyword extern has actually been defined in the linker script
of the kernel kern/kernel.ld after the end of the .bss section of the kernel. As
such, it points to the first byte of memory which has not been allocated for
kernel code and data. The code as of such is self explanatory. We equate result
with nextfree and increment nextfree by n bytes, aligning it over PGSIZE i.e. 4096
bytes at the same time and return result as a pointer to the first block of
allocated memory.
*/
	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);
	}
	result = nextfree;

	// 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.

	nextfree = ROUNDUP ( result + n, PGSZIE);
	if (nextfree >= (KERNBASE + PTSIZE))
	{
		cprintf("OUT OF MEMORY");
		panic ("boot alloc Failed to allocate %d bytes", n);
	}
	return result;
}

void
mem_init(void)
{
 // 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(npages * sizeof (struct PageInfo));
memset (pages, 0, pages * sizeof(struct PageInfo));
}
```

The main purpose of the below function is to create and initialize data structures that are necessary to maintain metadata of all free pages in the memory.

```
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...)
    pages[0].pp_ref = 1;
    pages [0].pp_link = NULL;
    //  2) The rest of base memory, [PGSIZE, npages_basemem * PGSIZE)
    //     is free.
    for (int i = 1; i < npages_basemem; i++)
    {
        pages[i].pp_ref = 0;
        pages[i].pp_link = page_free_list;
        page_free_list = &pages [i];
    }
    //  3) Then comes the IO hole [IOPHYSMEM, EXTPHYSMEM), which must
    //     never be allocated.
    uint32_t free_pa = PADDR (boot_alloc (0));
    uint32_t free_pa_index = free_pa / PGSIZE;
    for (int i = npages_basemem; i < free_pa_index; i++)
    {
        pages[i].pp_ref = 1;
        pages[i].pp_link = NULL;
    }

    //  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!
/*
With boot_alloc function used above giving us the address of the first unallocated
i.e. free page in memory, PADDR macro which returns the physical address of the
```

```
kernel virtual address is being used to mark the pages not allocated by boot_alloc
to the page_free_list.

*/

     for (int i = free_pa_index; i < npages; i++)

     {

          pages[i].pp_ref = 0;

          pages[i].pp_link = free_page_list;

          free_page_list = &pages[i];

     }

     }

}


// 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. The former should be used as struct PageInfo wil
contain  the  physical  addresses  of  the  pages  which  need  to  be  converted  into
virtual addresses.

struct PageInfo *

page_alloc(int alloc_flags)

{

     // Fill this function in


     struct PageInfo* allocate_page = page_free_list;

     if (allocate_page == NULL)

     return NULL;
```

```
    page_free_list = allocate_page -> pp_link;
    allocate_page -> pp_link = NULL;

    if (alloc_flags & ALLOC_ZERO)
    memset (page2kva (allocate_page), 0, PGSIZE);

    return allocate_page;
}

//
// 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.

    assert (pp->pp_ref == 0);
    assert (pp->pp_link == NULL);

    pp->pp_ref = 0;
    pp->pp_link = page_free_list;
    page_free_list = pp;
}
```

Thus as is clear, there are two data structures being used. The array pages which contains metadata of all physical pages and page_free_list which is the linked list and contains all free pages within the address space. Of the two elements of the struct PageInfo, the pp_link will be set to NULL when a page is allocated and only the ref field will a role for the page. Conversely, adding a page to the page_free_list will make the pp_link point to the next free page in the list and the set the ref field to 0, thus utilizing only the pp_link field of the structure.

Part 2: Virtual Memory

Virtual, Linear, and Physical Addresses

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.

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

A C pointer is the "offset" component of the virtual address. In boot/boot.S, we installed a Global Descriptor Table (GDT) that effectively disabled segment translation by setting all segment base addresses to 0 and limits to 0xffffffff. Hence the "selector" has no effect and the linear address always equals the offset of the virtual address.

From code executing on the CPU, once we're in protected mode (which we entered first thing in boot/boot.S), there's no way to directly use a linear or physical address. All memory references are interpreted as virtual addresses and translated by the MMU, which means all pointers in C are virtual addresses.

The JOS kernel often needs to manipulate addresses as opaque values or as integers, without dereferencing them, for example in the physical memory allocator. Sometimes these are virtual addresses, and sometimes they are physical addresses. To help document the code, the JOS source distinguishes the two cases: the type uintptr_t represents opaque virtual addresses, and physaddr_t represents physical addresses. Both these types are really just synonyms for 32-bit integers (uint32_t), so the compiler won't stop you from assigning one type to another! Since they are integer types (not pointers), the compiler will complain if you try to dereference them.

The JOS kernel can dereference a uintptr_t by first casting it to a pointer type. In contrast, the kernel can't sensibly dereference a physical address, since the MMU translates all memory references. If you cast a physaddr_t to a pointer and dereference it, you may be able to load and store to the resulting address (the hardware will interpret it as a virtual address), but you probably won't get the memory location you intended.

To summarize:

C type	Address type

*T**	*Virtual*
uintptr_t	*Virtual*
physaddr_t	*Physical*

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;
```

Since the function return_a_pointer () returns a virtual address (all pointers are virtual addresses), it should be uintptr_t as x will contain a virtual address of value.

The JOS kernel sometimes needs to read or modify memory for which it knows only the physical address. For example, adding a mapping to a page table may require allocating physical memory to store a page directory and then initializing that memory. However, the kernel cannot bypass virtual address translation and thus cannot directly load and store to physical addresses. One reason JOS remaps all of physical memory starting from physical address 0 at virtual address 0xf0000000 is to help the kernel read and write memory for which it knows just the physical address. In order to translate a physical address into a virtual address that the kernel can actually read and write, the kernel must add 0xf0000000 to the physical address to find its corresponding virtual address in the remapped region. You should use KADDR(pa) to do that addition.

The JOS kernel also sometimes needs to be able to find a physical address given the virtual address of the memory in which a kernel data structure is stored. Kernel global variables and memory allocated by boot_alloc() are in the region where the kernel was loaded, starting at 0xf0000000, the very region where we mapped all of physical memory. Thus, to turn a virtual address in this region into a physical address, the kernel can simply subtract 0xf0000000. You should use PADDR(va) to do that subtraction.

Reference counting

In future labs you will often have the same physical page mapped at multiple virtual addresses simultaneously (or in the address spaces of multiple environments). You will keep a count of the number of references to each physical page in the pp_ref field of the struct PageInfo corresponding to the physical page. When this count goes to zero for a physical page, that page can be freed because it is no longer used. In general, this count should be equal to the number of times the physical page appears below UTOP in all page tables (the mappings above UTOP are mostly set up at boot time by the kernel and should never be freed, so there's no need to reference count them). We'll also use it to keep track of the number of pointers we keep to the page directory pages and, in turn, of the number of references the page directories have to page table pages.

Be careful when using page_alloc. The page it returns will always have a reference count of 0, so pp_ref should be incremented as soon as you've done something with the returned page (like inserting it into a page table). Sometimes this is handled by other functions (for example, page_insert) and sometimes the function calling page_alloc must do it directly.

Page Table Management

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.

```
// 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.
//
/*
Being given the virtual address, we use the same i.e. first 10 bits to offset into
the page directory for page table.
In case the age table does not exist, we use the page_alloc function to create a
page table page and increment its ref count. Using the page2pa macro to append
appropriate permissions to the page and obtain its physical address, we insert the
latter into the page directory and head over to the address of the page table use
the PTE_ADDR macro which returns the address of the page table or page directory
entry defined in inc/mmu.h. getting the a pointer to the virtual address of the
page table obtaiined from its physical address as returned by PTE_ADDR through
KADDR macro, we return the address held for the page table entry by offsetting
into the page table through PTX macro defined in inc/mmmu.h
*/
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
{
    // Fill this function in

    uintptr_t address = (uintptr_t) va;
    pde_t pde_offset = pgdir [PDX(address)];
    if (!(pde_offset & PTE_P) && create)
    {
        struct PageInfo* new_page = page_alloc (ALLOC_ZERO);
        if (!new_page) return NULL;

        new_page -> pp_ref ++;
        pde_offset = page2pa (new_page) | PTE_W | PTE_U | PTE_P;
        pgdir [PDX(address)] = pde_offset;
    } else if (!(pde_offset & PTE_P)) return NULL;

    physaddr_t pt_pa = PTE_ADDR(pde_offset);
    pte_t* pt_va = KADDR(pt_pa);
    return &pt_va [PTX(address)];
}

// 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
/*
Getting the actual number of pages that are to be mapped by dividing size by
PGSIZE, we use pgdir_walk to return pointer to the page table entry corresponding
to the va. Since reference count for the pages that are to be mapped doesn't
matter, we insert the physical address at the pointer returned by pgdir_walk with
granularity of pages, thus establishing the mapping.
*/
static void
boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int perm)
{
    // Fill this function in

    if (va % PGSIZE != 0 || pa % PGSIZE != 0 || size % PGSIZE != 0)
    panic ("boot_map_region cannot be executed \n");

    uint32_t no_pages = size / PGSIZE;
    for (int i = 0; i < no_pages; i ++)
    {
        pte_t* pte_entry = pgdir_walk (pgdir, (void*) (va + i*PGSIZE), 1);
        assert (pte_entry != NULL);
        *pte_entry = (pa + i *PGSIZE) | perm | PTE_P;
    }
}


//
// 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.
//
/*
The function is used to return the struct PageInfo for the physical page corresponding to the virtual address va. If no mapping for va exists the function returns NULL.
The pte_store is passed as double pointer since we want to access the value of pte_store outside the fucntion as well. In case of single pointer, it wont be possible as a single poointer will be passed by value to the function meaning a duplicate of th pte_store or its equivalent I.e pointer by some other name will be created and passed to function whenever pagee_lookup is called. Consequently, we wont be able to access it is value outside of the function in case a single pointer. Storint in the double pointer pte_store the value of the pte, we can access it outside the function as well.
The function finally, used the macro pa2page in kern/pmap.h. With pgdir_walk returning a pointer to the PTE, corresponding to va, the PTE_ADDR macro will convert the value pointed to by the returned pointer into physical address which in turn will be converted into the  virtual address of PageInfo struct corresponding to that page which is the desired return value.
*/
struct PageInfo *
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
    // Fill this function in
```

```
     pte_t* pte_entry = pgdir_walk (pgdir, va, 0);

     if (!pte_entry || !(*pte_entry & PTE_P))
     return NULL;

     if (pte_store)
     *pte_store = pte_entry;

     return pa2page(PTE_ADDR(*pte_entry));
}

// 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.
//
/*
Using the address of PTE stred in the pointer pte_address, the same is set to zero
and the two methods are called to fulfill the Details.
*/
void
page_remove(pde_t *pgdir, void *va)
{
     // Fill this function in
```

```
    pte_t* pte_address = NULL;
    struct PageInfo* pp = page_lookup (pgdir, va, &pte_address);
    if (!pp)
    return;

    *pte_address = 0;
    page_decref(pp);
    tlb_invalidate (pgdir, va);
}
// 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.
```

```
/*

The function is used to map the page at PA pp to va. Utilizing pgdir_walk, the
function will return a pointer to the appropriate PTE, creating one if necessary
as we have set the create flag to be true. In order to avoid the mentioned corner
case, check the physical address of the page provided through PagInfo struct with
the physical address at the PTE. IN case of the addresses being same, we check for
the the permissions of both. The 0x1FF which is anded with the PTE PA, is
essentially to just compare the permissions of the page at PTE with those passed
as function parameters. If the permissions too are same, the function plainly
exits. In case of not, the PA at pte will appended with the permissions provided
through the function, the tlb will be invalidated and the function returns a 0.
Page removl is not needed in this case.

In case of a mapping already existing at va, the same will be removed tghrough
page_remove function and the new mapping will  be established on incrementing the
reference count of the physical page represented by struct PageInfo* pp.

*/

int

page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int perm)

{

     // Fill this function in


     pte_t* pte_entry = pgdir_walk (pgdir, va, true);


     if (!pte_entry) return -E_NO_MEM;


     if (PTE_ADDR(*pte_entry) == page2pa (pp))

     {

          if ((*pte_entry & 0x1FF) == perm) return 0;


          *pte_entry = page2pa (pp) | perm | PTE_P;

          tlb_invalidate (pgdir, va);

          return 0;

     }


     if (*pte_entry & PTE_P)

     {

          page_remove (pgdir, va);
```

```
        assert (*pte_entry ==0);
    }


    pp -> pp_ref ++;
    *pte_entry = page2pa (pp) | perm | PTE_P;
    return 0;
    }
```

Part 3: Kernel Address Space

JOS divides the processor's 32-bit linear address space into two parts. User environments (processes), which we will begin loading and running in lab 3, will have control over the layout and contents of the lower part, while the kernel always maintains complete control over the upper part. The dividing line is defined somewhat arbitrarily by the symbol ULIM in inc/memlayout.h, reserving approximately 256MB of virtual address space for the kernel. This explains why we needed to give the kernel such a high link address in lab 1: otherwise there would not be enough room in the kernel's virtual address space to map in a user environment below it at the same time.

Permissions and Fault Isolation

Since kernel and user memory are both present in each environment's address space, we will have to use permission bits in our x86 page tables to allow user code access only to the user part of the address space. Otherwise bugs in user code might overwrite kernel data, causing a crash or more subtle malfunction; user code might also be able to steal other environments' private data. Note that the writable permission bit (PTE_W) affects both user and kernel code!

The user environment will have no permission to any of the memory above ULIM, while the kernel will be able to read and write this memory. For the address range [UTOP,ULIM), both the kernel and the user environment have the same permission: they can read but not write this address range. This range of address is used to expose certain kernel data structures read-only to the user environment. Lastly, the address space below UTOP is for the user environment to use; the user environment will set permissions for accessing this memory.

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.

```
 // 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 | PTE_P);

	//////////////////////////////////////////////////////////////////////
	// 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:
	uintptr_t address = KSTACKTOP - KSTKSIZE;
	  boot_map_region (kern_pgdir, address, KSTKSIZE, PADDR(bootstack), PTE_W |
PTE_P);

	//////////////////////////////////////////////////////////////////////
	// 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:
	uint32_t pa_range = 0xFFFFFFFF - KERNBASE +1;
	boot_map_region (kern_pgdir, KERNBASE, pa_range, 0, PTE_W | PTE_P);
```

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:

Entr	Base Virtual	Points to (logically):

y	Address	
1023	?	Page table for top 4MB of phys memory
1022	?	?
.	?	?
.	?	?
.	?	?
2	0x00800000	?
1	0x00400000	?
0	0x00000000	[see next question]

Ans:

Entry	***Base Virtual Address***	***Points to (logically):***
1023	*0xffc00000*	*Page table for top 4MB of phys memory*
...	*...*	*page addresses holding RAM*
960	*0xf0000000*	*the page table holding the mappings for the beginning of RAM (phyical address 0) (writable)*
959	*0xefc00000*	*kernel stack (writable)*
958	*0xef800000*	*unmapped*
957	*0xef400000*	*a virtual page table at virtual address UVPT.*
956	*0xef000000*	*page table that contains the pages struct (which is readonly)*
955	*0xeec00000*	*unmapped*
...	*...*	*unmapped*
0	*0x00000000*	*unmapped*

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?

Ans: The Kernel's memory will have its PTE_U not set as a result of which user programs cannot access kernel space. Trying to do so will result in a page fault and transfer control back to OS.

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

*Since JOS use 4MB UPAGES space to store all the PageInfo struct information, each struct is 8B, so we can store 512K PageInfo structs. The size of a page is 4KB, so we can have most 512K * 4KB = 2GB physical memory.*

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?

*268435456 (max RAM) / 4096 (bytes per page) = 65536 frames. Each frame occupies a page entry which is 4 bytes, so total 262144 bytes just for page tables. We'd need additional 4*1024 = 4096 bytes for page directory. Another overhead is the array of PageInfo structs where each struct occupies 8 bytes, and we have 1 struct per frame: 8*65536=524288 bytes. So if we sum everything up, we get 262144+4096+524288 = 790528 bytes of overhead.*

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?.

*After jmp *%eax finished.It is possible because entry_pgdir also maps va[0-4M) to pa[0-4M), it is necessary because later a kern_pgdir will be loaded and va[0, 4M) will be abandoned.*