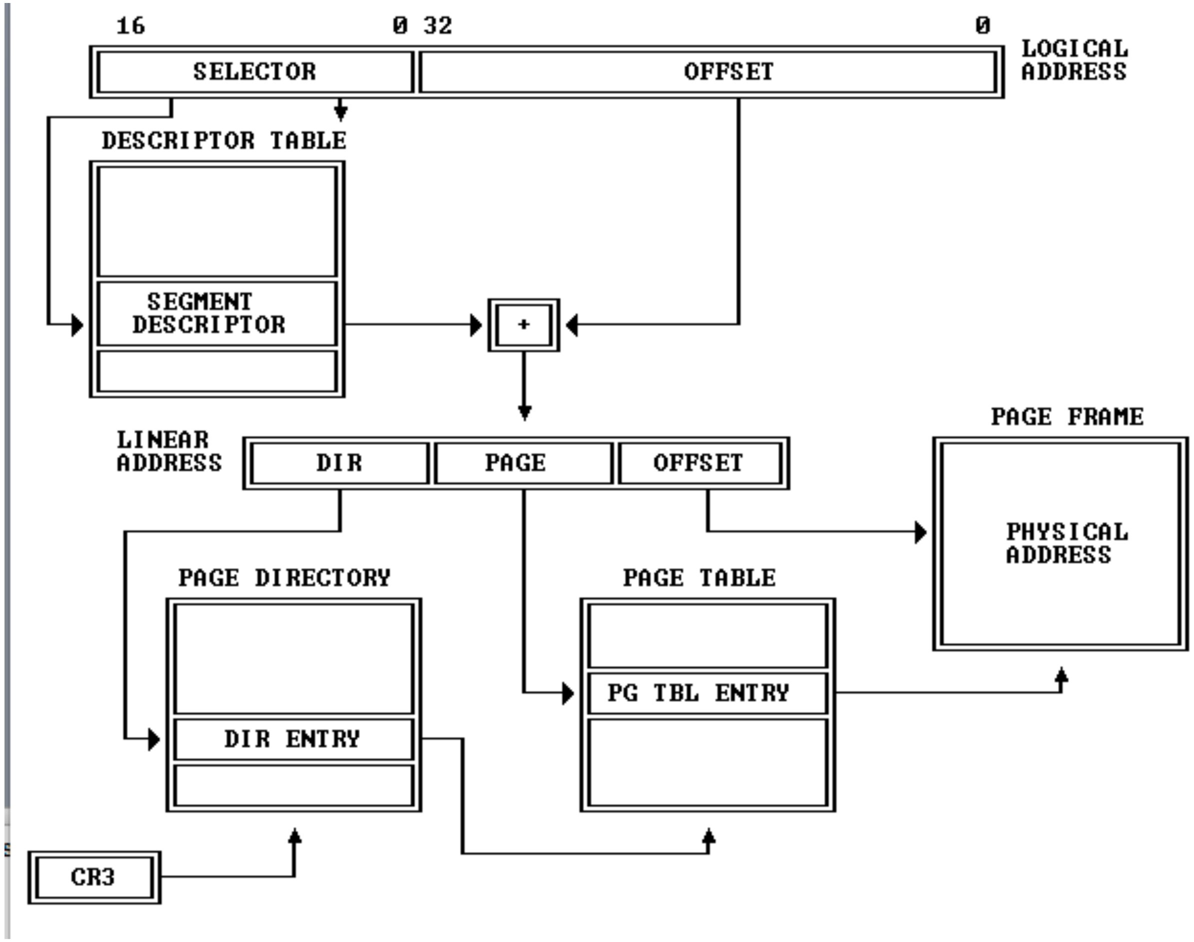
Sistemi Operativi

Lez. 19/20/21 :
Memoria Virtuale in JOS

I386- addressing mechanism



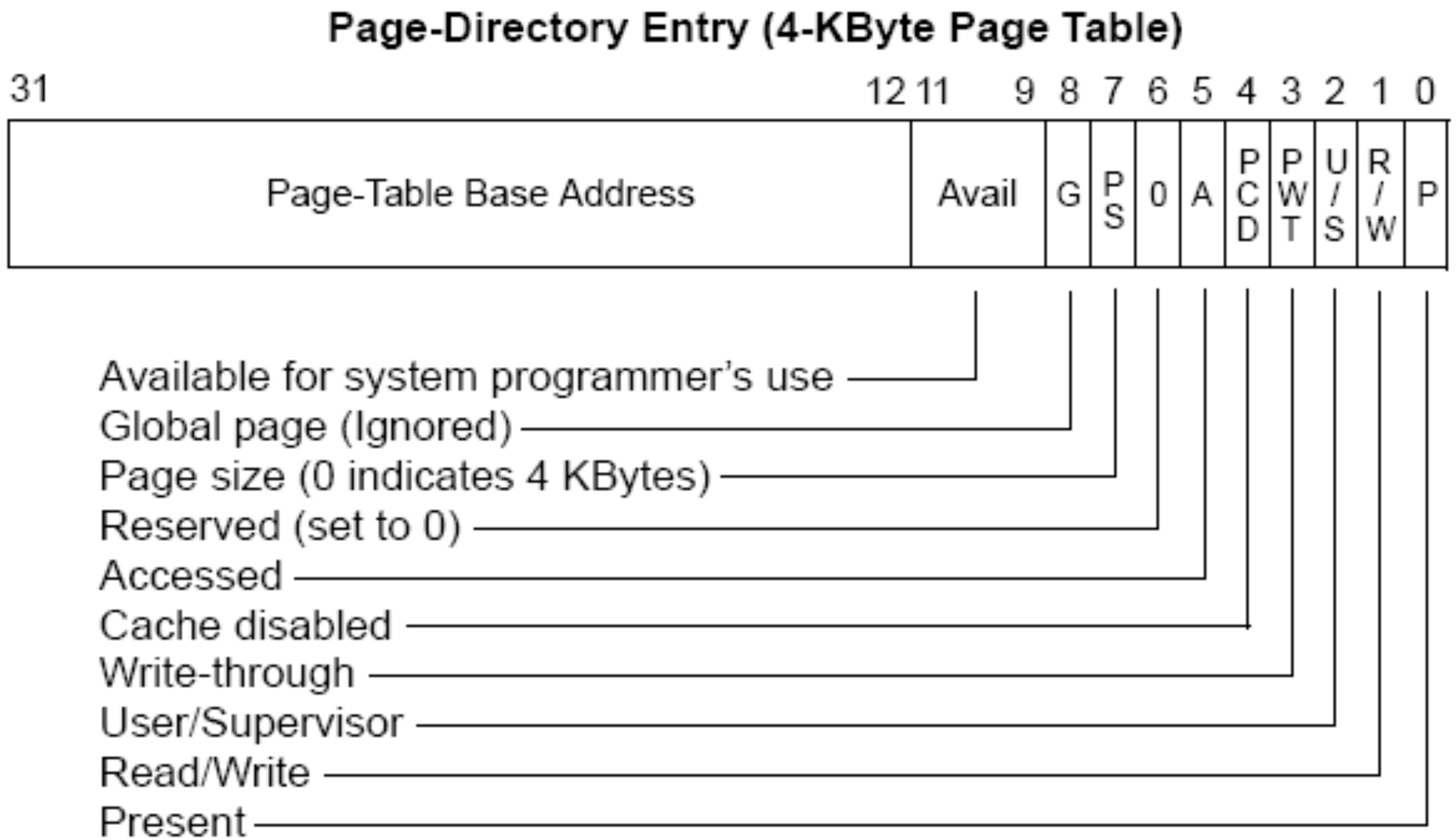
x86 terminology

- *virtual address* consists of a segment selector and an offset within the segment
- *linear address* is what you get after segment translation but before page translation
- *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

PD

- The physical address of the current page directory is stored in the CPU register CR3, also called the page directory base register (PDBR)
- Memory management software has the option of using one page directory for all tasks, one page directory for each task, or some combination of the two
- Entries in either level of page tables have the same format

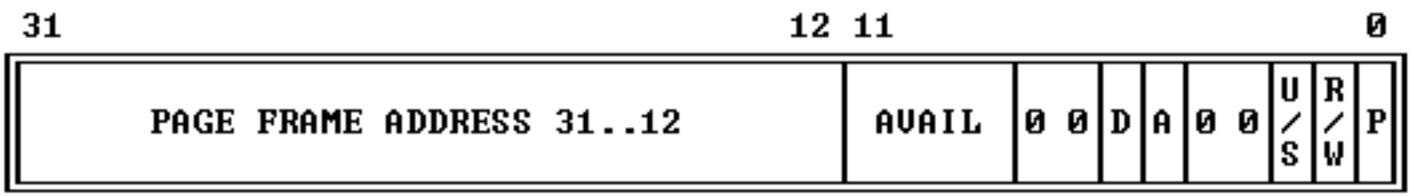
Page Directory entry



PT

- A page table is simply an array of 32-bit page specifiers. A page table is itself a page, and therefore contains 4 Kilobytes of memory or at most 1K 32-bit entries.
- Two levels of tables are used to address a page of memory. At the higher level is a page directory. The page directory addresses up to 1K page tables of the second level. A page table of the second level addresses up to 1K pages.
- All the tables addressed by one page directory, therefore, can address 1M pages (2^(20)).

Page table entry (PTE)




```
P      - PRESENT
R/W    - READ/WRITE
U/S    - USER/SUPERVISOR
D      - DIRTY
AVAIL  - AVAILABLE FOR SYSTEMS PROGRAMMER USE

NOTE: 0 INDICATES INTEL RESERVED. DO NOT DEFINE.
```

Address

- The page frame address specifies the physical starting address of a page
- In a page directory, the page frame address is the address of a page table
- In a second-level page table, the page frame address is the address of the page frame that contains the desired memory operand

Present bit

- The Present bit indicates whether a page table entry can be used in address translation
- P=1 indicates that the entry can be used. If P=0 in either level of page tables when an attempt is made to use a page-table entry for address translation, the processor signals a page exception
- There is no present bit for the page directory itself. The page directory may be not-present while the associated task is suspended, but the operating system must ensure that the page directory indicated by the CR3 image in the TSS is present in physical memory before the task is dispatched

R/W & U/S bits

- These bits are not used for address translation, but are used for page-level protection, which the processor performs at the same time as address translation
- For greatest efficiency in address translation, the processor stores the most recently used page-table data in an on-chip cache

Cache Flush

- The existence of the page-translation cache is invisible to applications programmers but not to systems programmers
- operating-system programmers must flush the cache whenever the page tables are changed.
- The page-translation cache can be flushed by reloading CR3 with a `mov` instruction
  - `mov %eax, %cr3`

Linear & physical address

- 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
- 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. Hence the "selector" has no effect and the linear address always equals the offset of the virtual address

THE FIRST SETTING OF VM DURING KERNEL INITIALIZATION

VM

- At the beginning the kernel maps the first 4MB of physical memory by using the hand-written, statically-initialized page directory and page table in kern/entrypgdir.c.
- Up until kern/entry.S sets the CR0_PG flag, memory references are treated as physical addresses (strictly speaking, they're linear addresses, but boot/boot.S set up an identity mapping from linear addresses to physical addresses and we're never going to change that).

VM

- Once CR0_PG is set, memory references are virtual addresses that get translated by the virtual memory hardware to physical addresses.
- `entry_pgdir` translates virtual addresses in the range 0xf0000000 through 0xf0400000 to physical addresses 0x00000000 through 0x00400000, as well as virtual addresses 0x00000000 through 0x00400000 to physical addresses 0x00000000 through 0x00400000.
- Any virtual address that is not in one of these two ranges will cause a hardware exception

Kernel (.code)

```
.globl entry
entry:
    movw    $0x1234,0x472            # warm boot
    movl    $(RELOC(entry_pgdir)), %eax
    movl    %eax, %cr3
    # Turn on paging.
    movl    %cr0, %eax
    orl     $(CR0_PE|CR0_PG|CR0_WP), %eax
    movl    %eax, %cr0
    # Now paging is enabled, but we're still running at a low EIP
    # (why is this okay?).  Jump up above KERNBASE before entering C code
    mov     $relocated, %eax
    jmp     *%eax
relocated:
    movl    $0x0,%ebp               # nuke frame pointer
    movl    $(bootstacktop),%esp
    call    i386_init
spin:
    jmp spin
```

entrypgdir.c

```
// The entry.S page directory maps the first 4MB of physical memory
// starting at virtual address KERNBASE (that is, it maps virtual
// addresses [KERNBASE, KERNBASE+4MB) to physical addresses [0, 4MB)).

// We choose 4MB because that's how much we can map with one page
// table and it's enough to get us through early boot.

// We also map
// virtual addresses [0, 4MB) to physical addresses [0, 4MB); this
// region is critical for a few instructions in entry.S and then we
// never use it again.
```

First Paging structure

- We must build the VM data structures that will enable the following mapping among virtual addresses and physical addresses:
  - [0xf0000000, 0xf0400000) → [0x00000000, 0x00400000)
  - [0x0000000, 0x00400000) → [0x00000000, 0x00400000)

Some constants

```
#define PTSIZE        (PGSIZE*NPTENTRIES) // bytes mapped by a page
                                          // directory entry
#define PTSHIFT       22       // log2(PTSIZE)

#define PTXSHIFT 12      // offset of PTX in a linear address
#define PDXSHIFT 22      // offset of PDX in a linear address

// 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
```

… and types

```
typedef uint32_t pte_t;
typedef uint32_t pde_t;

pte_t entry_pgtable[NPTENTRIES];
pde_t entry_pgdir[NPDENTRIES];
```

Page directory

```
// Page directories (and page tables), must start on a page boundary,
// hence the "__aligned__" attribute.  Also, because of restrictions
// related to linking and static initializers, we use "x + PTE_P"
// here, rather than the more standard "x | PTE_P".  Everywhere else
// you should use "|" to combine flags.

__attribute__((__aligned__(PGSIZE)))

pde_t entry_pgdir[NPDENTRIES] = {

        // Map VA's [0, 4MB) to PA's [0, 4MB)
        [0] = ((uintptr_t)entry_pgtable - KERNBASE) + PTE_P,

        // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
        [KERNBASE>>PDXSHIFT]
                = ((uintptr_t)entry_pgtable - KERNBASE) + PTE_P + PTE_W
};
```

Page table

```
// Entry 0 of the page table maps to physical page 0, entry 1 to
// physical page 1, etc.
__attribute__((__aligned__(PGSIZE)))
pte_t entry_pgtable[NPTENTRIES] = {
    0x000000 | PTE_P | PTE_W,
    0x001000 | PTE_P | PTE_W,
    0x002000 | PTE_P | PTE_W,
    0x003000 | PTE_P | PTE_W,
    0x004000 | PTE_P | PTE_W,
    0x005000 | PTE_P | PTE_W,
    0x006000 | PTE_P | PTE_W,
    0x007000 | PTE_P | PTE_W,
    0x008000 | PTE_P | PTE_W,
    0x009000 | PTE_P | PTE_W,
    0x00a000 | PTE_P | PTE_W,
    0x00b000 | PTE_P | PTE_W,
    0x00c000 | PTE_P | PTE_W,
    0x00d000 | PTE_P | PTE_W,
    0x00e000 | PTE_P | PTE_W,
```

Page table

```
    0x3ef000 | PTE_P | PTE_W,
    0x3f0000 | PTE_P | PTE_W,
    0x3f1000 | PTE_P | PTE_W,
    0x3f2000 | PTE_P | PTE_W,
    0x3f3000 | PTE_P | PTE_W,
    0x3f4000 | PTE_P | PTE_W,
    0x3f5000 | PTE_P | PTE_W,
    0x3f6000 | PTE_P | PTE_W,
    0x3f7000 | PTE_P | PTE_W,
    0x3f8000 | PTE_P | PTE_W,
    0x3f9000 | PTE_P | PTE_W,
    0x3fa000 | PTE_P | PTE_W,
    0x3fb000 | PTE_P | PTE_W,
    0x3fc000 | PTE_P | PTE_W,
    0x3fd000 | PTE_P | PTE_W,
    0x3fe000 | PTE_P | PTE_W,
    0x3ff000 | PTE_P | PTE_W,
};
```

kernel/entry.s

```
###############################################################
# The kernel (this code) is linked at address ~(KERNBASE + 1 Meg),
# but the bootloader loads it at address ~1 Meg.
#
# RELOC(x) maps a symbol x from its link address to its actual
# location in physical memory (its load address).
##############################################################
#define  KERNBASE 0xF0000000
#define  RELOC(x) ((x) - KERNBASE)
# '_start' specifies the ELF entry point.  Since we haven't set up
# virtual memory when the bootloader enters this code, we need the
# bootloader to jump to the *physical* address of the entry point.

.globl      _start
_start = RELOC(entry)

.globl entry

entry:
   movw  $0x1234,0x472            # warm boot
```

```
    # We haven't set up virtual memory yet, so we're running from
    # the physical address the boot loader loaded the kernel at: 1MB
    # (plus a few bytes).  However, the C code is linked to run at
    # KERNBASE+1MB.  Hence, we set up a trivial page directory that
    # translates virtual addresses [KERNBASE, KERNBASE+4MB) to
    # physical addresses [0, 4MB).  This 4MB region will be suffice
    # until we set up our real page table in mem_init in lab 2.

    # Load the physical address of entry_pgdir into cr3.  entry_pgdir
    # is defined in entrypgdir.c.
    movl  $(RELOC(entry_pgdir)), %eax
    movl  %eax, %cr3
    # Turn on paging
    movl  %cr0, %eax
    orl   $(CR0_PE|CR0_PG|CR0_WP), %eax
    movl  %eax, %cr0

    # Now paging is enabled, but we're still running at a low EIP
    # (why is this okay?).  Jump up above KERNBASE before entering
    # C code.
    mov   $relocated, %eax
    jmp   *%eax       #after this jump paging starts working
relocated:
```

kern/entry.s

- The processor is still executing instructions at low addresses after paging is enabled, which works since entrypgdir maps low addresses. If we had omitted entry 0 from entrypgdir, the computer would have crashed when trying to execute the instruction after the one that enabled paging
- The indirect jump is needed because the assembler would generate a PC-relative direct jump, which would execute the low-memory version of kernel

kern/entry.s

- Now entry needs to transfer to the kernel's C code, and run it in high memory.
- First it must make the stack pointer, %esp, point to a stack so that C code will work. All symbols have high addresses, including stack, so the stack will still be valid even when the low mappings are removed. Finally entry jumps to kernel, which is also a high address
- entry cannot return, since there's no return PC on the stack

kernel/entry.s

```
# Clear the frame pointer register (EBP)
# so that once we get into debugging C code,
# stack backtraces will be terminated properly.

movl $0x0,%ebp           # nuke frame pointer

# Set the stack pointer
movl $(bootstacktop),%esp
# now to C code
call i386_init

# Should never get here, but in case we do, just spin.
spin:     jmp  spin

.data
###################################################################
# boot stack
###################################################################
    .p2align    PGSHIFT         # force page alignment
    .globl      bootstack
bootstack:
    .space      KSTKSIZE
    .globl      bootstacktop
bootstacktop:
```

kern/init.c

```
/* See COPYRIGHT for copyright information. */

#include <inc/stdio.h>
#include <inc/string.h>
#include <inc/assert.h>
#include <kern/monitor.h>
#include <kern/console.h>
#include <kern/pmap.h>
#include <kern/kclock.h>
#include <kern/env.h>
#include <kern/trap.h>

void
i386_init(void)
{
    extern char edata[], end[];
```

kern/init.c

```
// Before doing anything else, complete the ELF loading process.

    // Clear the uninitialized global data (BSS) section of our program.
    // This ensures that all static/global variables start out zero.
    memset(edata, 0, end - edata);

    // Initialize the console.
    // Can't call cprintf until after we do this!
    cons_init();

    cprintf("6828 decimal is %o octal!\n", 6828);

    // Lab 2 memory management initialization functions
    mem_init();

    // Lab 3 user environment initialization functions
    env_init();
    trap_init();
```

SETTING THE KERNEL MEMORY LAYOUT

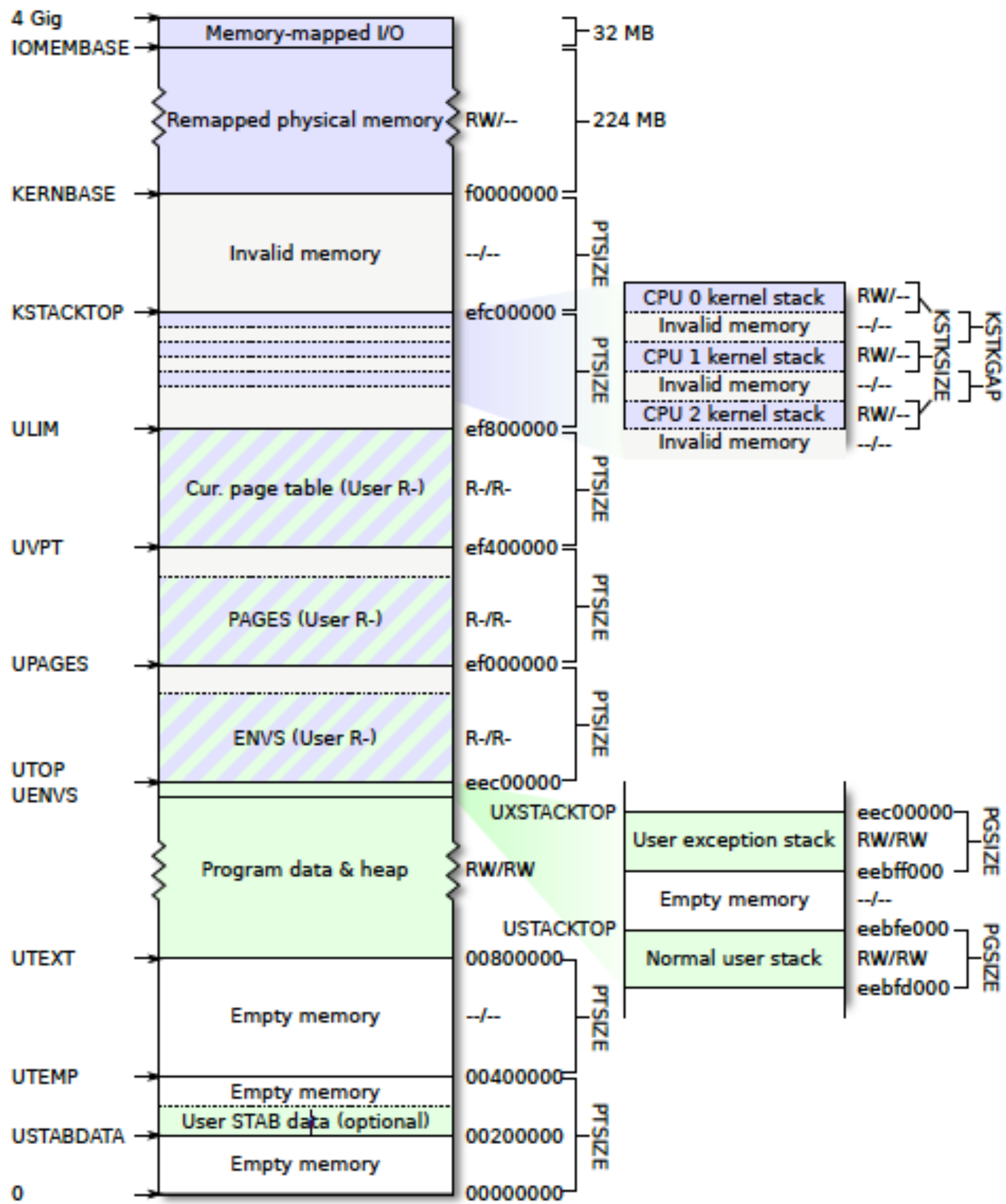
init.c richiamato da entry.s

```
void
i386_init(void)
{
	extern char edata[], end[];

	// Before doing anything else, complete the ELF loading process.
	// Clear the uninitialized global data (BSS) section of our program.
	// This ensures that all static/global variables start out zero.
	memset(edata, 0, end - edata);

	// Initialize the console.
	// Can't call cprintf until after we do this!
	cons_init();

	cprintf("6828 decimal is %o octal!\n", 6828);

	// Lab 2 memory management initialization functions
	mem_init();
```

`mem_init()`

- The purposes of `mem_init()` are:
  - to build a new page directory and the related page tables which map the JOS memory layout, these data structures will substitute those initially set up by the kernel
  - To build the data structures for managing the physical frame

JOS Virtual Memory Map



mem_init (file pmap.c)

```
// 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 setup 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();
```

mem_init (crea una PD vuota)

```
////////////////////////////////////////////////////
// create initial page directory.

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

boot_alloc (kern/pmap.c)

```
// If n>0, allocates enough pages of contiguous physical
// memory to hold 'n’  bytes.  Doesn't initialize the memory.

// Returns the kernel virtual address of the
// allocated memory chunk..

// If n==0, returns the address of the next free page without
// allocating anything.
//
// If we're out of memory, boot_alloc should panic.
```

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);
    }
```

boot_alloc

```
    // Allocate a chunk large enough to hold 'n' bytes, then update
    // nextfree.  Make sure nextfree is kept aligned
    // to a multiple of PGSIZE.
    if (n > 0) {
        // Round-up alloc_size promises round-up nextfree.
        uint32_t alloc_size = ROUNDUP(n, PGSIZE);
        result = nextfree;
        nextfree += alloc_size;
        // only 4MB of physical memory is mapped so far,
        // memory allocation cannot exceeds the limit.
        if ((uintptr_t)nextfree >= 0xf0400000)
           panic("boot_alloc: out of memory");
                }
        else {result = nextfree; }
    return result;
}
```

MANAGING FRAMES

```
void
mem_init(void)
{
    uint32_t cr0; size_t n; char * dummy;
// Find out how much memory the machine has (npages &
// npages_basemem).
    i386_detect_memory();
// create initial page directory.
    kern_pgdir = (pde_t *) boot_alloc(PGSIZE);
    memset(kern_pgdir, 0, PGSIZE);
// Allocate an array of npages 'struct PageInfo's and store it
//in 'pages'.
    pages = boot_alloc (npages * sizeof(struct PageInfo));
    envs = boot_alloc (NENV * sizeof(struct Env));
// Now that we've allocated the initial kernel data
structures, we set  up the list of free physical pages.
    page_init();
    check_page_free_list(1);
    check_page_alloc();
    check_page();
```

mem_init (2)

```
//////////////////////////////////////////////////////////////////////
// Allocate an array of npages 'struct Page'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 Page in this
// array.  'npages' is the number of physical pages in memory.
    pages = (struct Page *)boot_alloc(npages * sizeof(struct Page));

//////////////////////////////////////////////////////////////////////
// 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();
```

Frame 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
- It keeps track of which pages are free with a linked list of struct PageInfo objects, 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

Frame management

- The frame management module consists of three major components:
  - Initialization module
  - Frame allocation module
  - Frame de-allocation module

struct page

```
 * Each struct Page stores metadata for one physical page.
 * Is it NOT the physical page itself, but there is a one-to-one
 * correspondence between physical pages and struct Page's.
 * You can map a Page * to the corresponding physical address
 * with page2pa() in kern/pmap.h.
 */
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;
};
```

Setup pages data structure

```
/////////////////////////////////////////////////////////////
// Allocate an array of npages 'struct Page'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 Page in this  array.  'npages' is
//  the number of physical pages in memory.

size = npages * (sizeof(struct PageInfo));
pages = (struct Page *) boot_alloc(size);
```

page_init (initializes pages data structure)

```
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?
    //
```

Variables declaration

```
// These variables are set by i386_detect_memory()
size_t npages;         // Amount of physical memory (in pages)
static size_t npages_basemem; // Amount of base memory (in
                              // pages)

// These variables are set in mem_init()
pde_t *kern_pgdir;        // Kernel's initial page directory
struct Page *pages;       // Physical page state array
static struct Page *page_free_list;  // Free list of physical
                                     // pages
```

page_init

```
size_t i,j;
physaddr_t physadd;
page_free_list = NULL;
for (i = 1; i < npages_basemem; i++) {
    pages[i].pp_ref = 0;
    pages[i].pp_link = page_free_list;
    page_free_list = &pages[i];
}
physadd = PADDR (nextfree);
j = (int) (physadd) / PGSIZE;
for (i = j; i< npages; i++) {
    pages[i].pp_ref = 0;
    pages[i].pp_link = page_free_list;
    page_free_list = &pages[i];
}
```

```
// 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)
```

Alcune costanti

```
// Page directory and page table constants.
#define NPDENTRIES   1024    // page directory entries per page directory
#define NPTENTRIES   1024    // page table entries per page table

#define PGSIZE       4096    // bytes mapped by a page
#define PGSHIFT      12      // log2(PGSIZE)

#define PTSIZE       (PGSIZE*NPTENTRIES) // bytes mapped by a page directory entry
#define PTSHIFT      22      // log2(PTSIZE)

#define PTXSHIFT 12      // offset of PTX in a linear address
#define PDXSHIFT 22      // offset of PDX in a linear address
```

KADDR(pa)

- JOS kernel sometimes needs to read or modify memory for which it only knows the physical address and the kernel, like any other software, cannot bypass virtual memory translation and thus cannot directly load and store to physical addresses
- One reason JOS remaps of 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

KADDR(pa)

- 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
- PADDR (ka) will do the reverse operation

KADDR(pa)

```
/* 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);
}
```

PADDR(kva)

```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;
}
```

PA ←→ KA

```
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)); }
```

Exercise

```
// 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).
//
// Returns NULL if out of free memory.
//
// Hint: use page2kva and memset

struct PageInfo *
page_alloc(int alloc_flags)
```

page_alloc

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

Exercise

```
//
// 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)
```

page_free

```
{
   if (pp->pp_ref == 0) {
      memset(page2kva(pp), 0x00, PGSIZE);
      pp->pp_link = page_free_list;
      page_free_list = pp; }
      }
```

MAPPING VM

Some constants

```
#define PTSIZE        (PGSIZE*NPTENTRIES) // bytes mapped by a page directory
                                          // entry
#define PTSHIFT       22       // log2(PTSIZE)

#define PTXSHIFT 12        // offset of PTX in a linear address
#define PDXSHIFT 22        // offset of PDX in a linear address

// 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
```

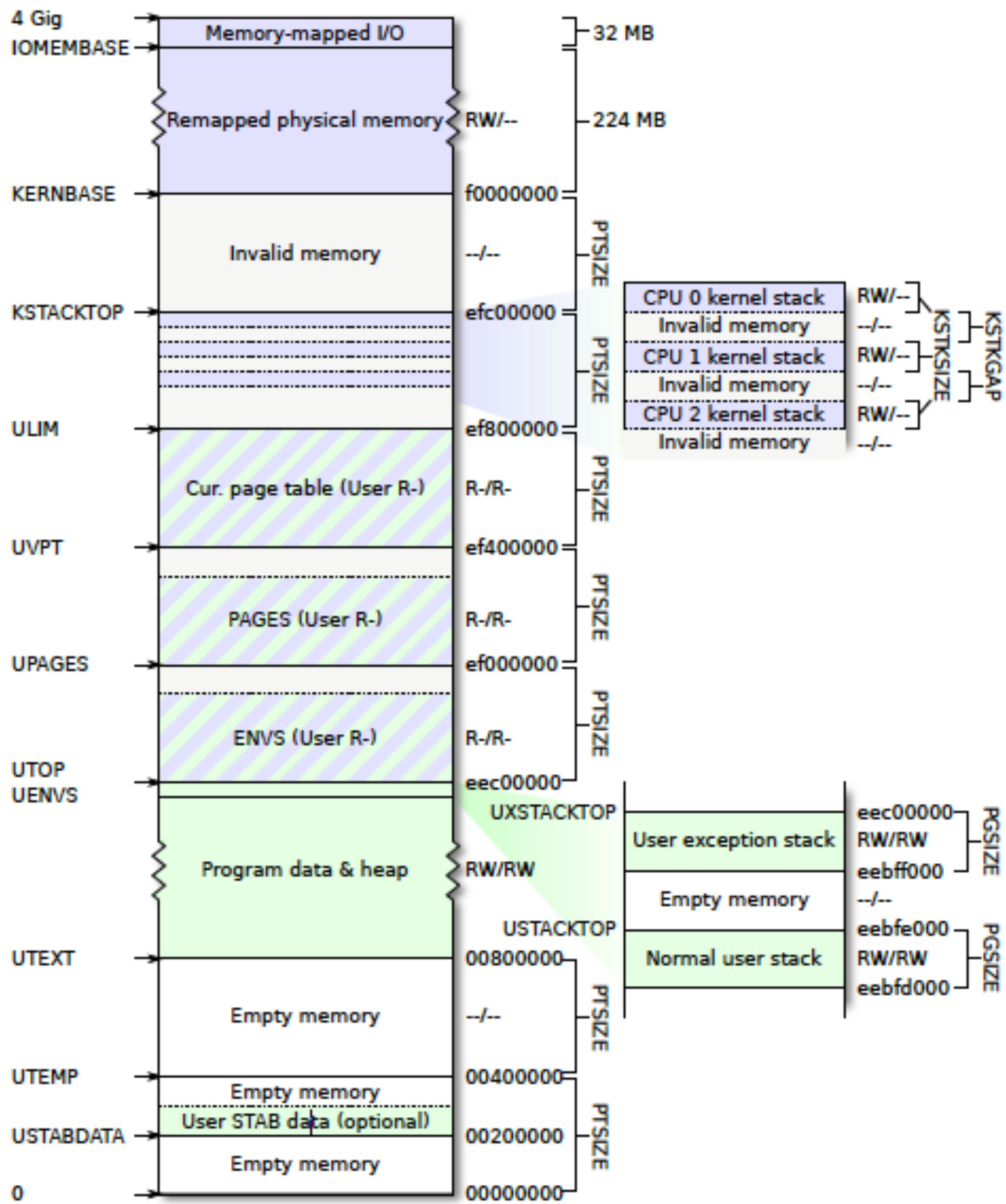
memlayout.h

```
// All physical memory mapped at this address
#define KERNBASE    0xF0000000

// At IOPHYSMEM (640K) there is a 384K hole for I/O.  From the kernel,
// IOPHYSMEM can be addressed at KERNBASE + IOPHYSMEM.  The hole ends
// at physical address EXTPHYSMEM.
#define IOPHYSMEM   0x0A0000
#define EXTPHYSMEM  0x100000

// Kernel stack.
#define KSTACKTOP   (KERNBASE - PTSIZE)
#define KSTKSIZE    (8*PGSIZE)          // size of a kernel stack
#define KSTKGAP     (8*PGSIZE)          // size of a kernel stack guard
#define ULIM        (KSTACKTOP - PTSIZE)
```

Memory Layout

- 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

JOS Virtual Memory Map



A.A. 2014/2015

Memory Layout Protection

- 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.

Memory Layout

- 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

SETTING OUT KERNEL PGDIR

Kernel Page Directory

```
//////////////////////////////////////////////////
// create initial page directory.

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

KERNBASE mapping

```
//////////////////////////////////////////////////////////////////////
// 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
```

KERNBASE mapping

```
k=0;
for (i= KERNBASE; i<(2^32); i = i + PTSIZE
{
    dummy = page_alloc(0); // alloca pagina per Page T.
    pgdir0 = &pgdir[PDX((uint32_t *)i)];
    *pgdir0 = page2pa(dummy)| PTE_P | PTE_W ;
    pt1 = (uint32_t *)page2kva(dummy);
    for (j= 0; j<NPTENTRIES; j = j+1)
        {
            pt1[j] = k | PTE_P |PTE_W; //abilita pagina a
                                      // lettura/scrittura
            k = k + PGSIZE;
        }
}
```

A MORE GENERAL METHOD FOR MAPPING

pgdir_walk(pde_t *pgdir, const void *va, int create)

```
// 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.
```

```
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
{
    struct PageInfo *new_page;
    if ((pgdir[PDX(va)] & PTE_P) != 0)
        return &(((pte_t *)KADDR(PTE_ADDR(pgdir[PDX(va)])))[PTX(va)]);
    else
       {
        if (create == 0)
            return NULL;
        else
        {   new_page = page_alloc(ALLOC_ZERO);
            if (new_page == NULL)
                return NULL;
            new_page->pp_ref  = 1;
            pgdir[PDX(va)] = page2pa (new_page) |PTE_P | PTE_W | PTE_U;
        return &(((pte_t *)KADDR(PTE_ADDR(pgdir[PDX(va)])))[PTX(va)]);
        }
     }
}
```

boot_map_region

```
// 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.
// 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.
//
```

boot_map_region

```
static void
boot_map_region(pde_t *pgdir, uintptr_t va, size_t
size, physaddr_t pa, int perm)
{
   size_t index;
   if (size & 0xFFF) panic ("boot_map_region: size is
                              not a multiple of PGSIZE");
   for (index = 0; index < size; index = index + PGSIZE)
   *(pgdir_walk (pgdir, (void *)(va + index), 1)) =
                           (pa + index) | perm | PTE_P;

}
```

Map KERNBASE

```
//////////////////////////////////////////////////////////////////////
// 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


boot_map_region(kern_pgdir, KERNBASE,
                            (0xFFFFFFFF)-KERNBASE +
                             1, 0, PTE_W);
```

Map pages

```
/////////////////////////////////////////////////////////////////
// Map the 'pages' data structure at linear address UPAGES in
// such a way that it is read-only by the user
// Permissions:
//     - the new image at UPAGES -- kernel R, user R
//       (ie. perm = PTE_U | PTE_P)
//     - pages itself -- kernel RW, user NONE

boot_map_region (kern_pgdir, UPAGES,
ROUNDUP(npages*sizeof(struct PageInfo), PGSIZE), PADDR(pages),
PTE_U);
```

Exercise

```
////////////////////////////////
// Map the 'envs' array read-only by the user at linear
// address UENVS
// (ie. perm = PTE_U | PTE_P).
// Permissions:
//    - the new image at UENVS  -- kernel R, user R
//    - envs itself -- kernel RW, user NONE
```

Map envs

```
//////////////////////////////////////////////////////
// Map the 'envs' array read-only by the user at linear
// address UENVS
// (ie. perm = PTE_U | PTE_P).
// Permissions:
//    - the new image at UENVS  -- kernel R, user R
//    - envs itself -- kernel RW, user NONE
boot_map_region (kern_pgdir, UENVS,
ROUNDUP(NENV*sizeof(struct Env), PGSIZE), PADDR(envs),
PTE_U);
```

MAP Kernel Stack

```
//////////////////////////////////////////////////
// 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

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

When the PT is set up …

```
// 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);
```

page_insert

```
// 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
```

page_insert

```
int
page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int perm)
{
    pte_t *pte;
    pte = pgdir_walk(pgdir, va, 1);
    perm &= 0xfff; // truncate 'perm' to the right bits
    if (pte == NULL)
    return -E_NO_MEM;

    if (*pte & PTE_P) {
        if (PTE_ADDR(*pte) != page2pa(pp)) {
        // already a page mapped at 'va'
        page_remove(pgdir, va);
        *pte = page2pa(pp) | perm | PTE_P;
        // the ref is incremented, because it is used in the mapping.
        pp->pp_ref++;
        tlb_invalidate(pgdir, va); }
    else {
```

page_insert

```
else {
        // the same page mapped at 'va'
        // the permission may change
        *pte = (*pte & 0xfffff000) | perm | PTE_P;
        }
    } else {
    *pte = page2pa(pp) | perm | PTE_P;
// the ref is incremented, because it is used in the
mapping.
    pp->pp_ref++;
    }
    return 0;
}
```

page_remove

```
// 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 pg dir/pg table.
//
// Hint: The TA solution is implemented using page_lookup,
//      tlb_invalidate, and page_decref.
//
```

page_remove

```
void
page_remove(pde_t *pgdir, void *va)
{
    struct PageInfo *page_to_remove;
    pte_t  *ptelement;

    page_to_remove = page_lookup (pgdir, va, &ptelement);
    if (page_to_remove == NULL) return;

    *ptelement  = 0;
    page_decref (page_to_remove);
    tlb_invalidate (pgdir, va);
    return;
}
```

page_free

```
void
page_free(struct Page *pp)
{
     if(pp->pp_ref != 0) {
          panic("a page should only be freed when its ref reaches 0.\n");
          }
           pp->pp_link = page_free_list;
           page_free_list = pp; }
           }
```

page_free

// Return a page to the free list

// This function should only be called when
// pp- >pp_ref reaches 0