
Sistemas Operativos

TP1 - Memoria virtual en JOS

Gabriel Robles – 95897

Ariel Windey – 97893

Repositorio: <https://github.com/gabyrobles93/TP1-SisOp>

page2pa

La función `page2pa` se encuentra definida en el archivo `pmap.h`:

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

Para entender su comportamiento es necesario comprender que representa cada tipo de dato:

- `typedef uint32_t physaddr_t`: el tipo de dato `physaddr_t` es un entero sin signo de 32 bits, utilizado para representar el valor de una dirección de memoria física.
- `struct PageInfo`: como se indica en el enunciado son estructuras con información asociada a las páginas de memoria física.
- `struct PageInfo *pages`: es el arreglo de páginas de memoria física, es decir es un puntero al primer `struct PageInfo` que está asociado a la primer página de memoria física

Entonces `(pp - pages)` es una cuenta que realiza aritmética de punteros en la que da el resultado del índice de la página de memoria física a la que corresponde la variable `pp`. Finalmente a este índice se le realiza un shift a izquierda de 12 posiciones (el valor de `PGSHIFT`), que es equivalente a multiplicar por la potencia 12 de 2, es decir 4096, que es precisamente el tamaño en bytes de cada página.

boot_alloc_pos

a) Partiendo desde `KERNBASE`, ubicado en `0xf0000000` (memoria virtual), el `kernel` es mapeado a partir del próximo MB desde esta posición, es decir en:

`KERNBASE + 0x00100000 = 0xf0000000 + 0x00100000 = 0xf0100000`

A partir de esta posición de memoria irá el `kernel`. Para determinar que posición de memoria devolverá el `boot_alloc(0)` antes de alocar memoria, es decir el valor con que se inicializó la variable `nextfree`, podemos averiguarlo de dos modos:

1. Corriendo el comando `size` sobre el binario, determinamos el tamaño del kernel:

```
→ TP1-SisOp git:(master) x size obj/kern/kernel
text      data      bss      dec      hex filename
34506     41728      1616    77850    1301a obj/kern/kernel
```

Como vemos ocupa 77850 bytes (0x0001301a en hexadecimal). Por lo tanto sumamos este valor a la posición memoria donde empieza el kernel:

$0xf0100000 + 0x0001301a = 0xf011301a$

Con lo cuál sabiendo que `nextfree` se inicializa en la dirección de memoria virtual de la página siguiente a la última página del kernel, deducimos que se inicializará en 0xf0114000.

2. Otro modo es ejecutar el comando `nm` sobre el binario, con la opción `-n` para obtener los símbolos del mismo ordenados según su posición en memoria:

```
→ TP1-SisOp git:(master) x nm -n obj/kern/kernel
0010000c T _start
f010000c T entry
...
f0113948 B kern_pgdir
f011394c B pages
f0113950 B end
```

Podemos ver que el último símbolo `end` (que de hecho es el que se utiliza para inicializar `nextfree`) se ubica en 0xf0113950, con lo cual volvemos a deducir que `nextfree` estará en 0xf0114000.

Para comprobar esto se agregó el siguiente código tras implementar la función `boot_alloc()`:

```
void
mem_init(void)
{
    uint32_t cr0;
    size_t n;

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

    // Remove this line when you're ready to test this function.
    cprintf("Nextfree, la página inmediata luego de que termina el kernel
en el AS: %p \n", boot_alloc(0));
    .
    .
    .
```

Ejecutando el kernel se obtiene lo siguiente:

```
QEMU
SeaBIOS (version 1.10.2-1ubuntu1)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDDD+07ECDDDD C980

Booting from Hard Disk...
6828 decimal is 15254 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
Nextfree, la pagina inmediata luego de que termina el kernel en el AS: 0xf011400
0
Npages cantidad de paginas fisicas: 32768
Sizeof PageInfo struct: 8>>>
>>> kernel panic at kern/pmap.c:113: boot_alloc: out of memory
>>>
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

b) A continuación se puede ver una sesión de gdb con lo que pide el enunciado y además comprueba lo formulado en el inciso anterior

```
→ TP1-SisOp git:(master) x make gdb
gdb -q -s obj/kern/kernel -ex 'target remote 127.0.0.1:26000' -n -x .gdbinit
Leyendo símbolos desde obj/kern/kernel...hecho.
Remote debugging using 127.0.0.1:26000
aviso: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.
0x0000ffff in ?? ()
.gdbinit: No existe el archivo o el directorio.
(gdb) b boot_alloc
Punto de interrupción 1 at 0xf0100995: file kern/pmap.c, line 98.
(gdb) c
Continuando.

Breakpoint 1, boot_alloc (n=0) at kern/pmap.c:98
98      if (!nextfree) {
(gdb) p nextfree
$1 = 0x0
(gdb) n
100      nextfree = ROUNDUP((char *) end, PGSIZE);
(gdb) p (char*)&end
$2 = 0xf0113950 "\020"
(gdb) n
112      if ((uintptr_t)ROUNDUP(nextfree + n, PGSIZE) > (KERNBASE + (4
<< 20))) {
(gdb) p nextfree
$3 = 0xf0114000 ""
(gdb) n
116      if (n > 0) {
(gdb) n
123  }
(gdb) p nextfree
$4 = 0xf0114000 ""
(gdb) n
mem_init () at kern/pmap.c:145
```

page_alloc

¿En qué se diferencia `page2pa()` de `page2kva()`?

El comportamiento de la función `page2pa()` fue descrito en la primer parte. La función `page2kva()`, hace uso de la misma pero en vez de retornar la dirección física asociada al `struct PageInfo` pasado por parámetro, a esta dirección física le aplica la macro `KADDR` que devuelve la dirección virtual correspondiente (siempre y cuando corresponde a una dirección del kernel, caso contrario ejecuta un `panic`). ...

map_region_large

¿Cuánta memoria se ahorró de este modo? ¿Es una cantidad fija, o depende de la memoria física de la computadora?

El uso de Large Pages tiene un ahorro de memoria. Lo podemos ver con el siguiente análisis:

Para referenciar 4 MiB de memoria física **sin** large pages necesitamos:

- Un PDE (Page Directory Entry) equivalente a 4 bytes.
- Una Page Table con todas sus entradas (1024 entradas de 4 bytes) las cuales referencian cada una a una página de memoria física de 4096 bytes. Entonces tenemos $1024 \times 4096 \text{ bytes} = 4 \text{ MiB}$.

Para referenciar 4MiB de memoria física **con** large pages necesitamos:

- Un PDE (Page Directory Entry) equivalente a 4 bytes. Con ese PDE ya referenciamos a los 4 MiB de memoria física.

Entonces el ahorro de memoria, **por cada large page**, es lo que ocupa la Page Table: 4096 bytes.

Con esto concluimos que dependiendo de cuánta memoria física tengamos distinta será la cantidad que tengamos que referenciar con estas estructuras. Utilizando la siguiente ecuación:

$$\text{Ahorrado} = (\text{Mem_fís_total} / 4 \text{ MiB}) * 4 \text{ KiB} = \text{MiB_fis_total} * \text{KiB}$$

Tenemos los siguientes resultados para distintas memorias físicas de máquinas virtuales en las que corremos JOS:

- 64 MiB: nos ahorramos 64 KiB.
- 128 MiB: nos ahorramos 128 KiB.
- 256 MiB: nos ahorramos 256 KiB.

Este cálculo es aproximado ya que despreciamos el tamaño que ocupa la Page Directory.

Git diff

```
diff --git a/kern/entry.S b/kern/entry.S
index 6c58826..025b11f 100644
--- a/kern/entry.S
+++ b/kern/entry.S
@@ -57,6 +57,13 @@ entry:
    # is defined in entrypgdir.c.
    movl $(RELOC(entry_pgdir)), %eax
    movl %eax, %cr3
+
+ # Configuro el registro cr4
+ # para large pages
+ movl %cr4, %eax
+ orl $(CR4_PSE), %eax
+ movl %eax, %cr4
+
+ # Turn on paging.
+ movl %cr0, %eax
+ orl $(CR0_PE|CR0_PG|CR0_WP), %eax
diff --git a/kern/entrypgdir.c b/kern/entrypgdir.c
index 4f324d1..3c8d81c 100644
--- a/kern/entrypgdir.c
+++ b/kern/entrypgdir.c
@@ -21,14 +21,15 @@ __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
+    = (0) + PTE_P + PTE_W + PTE_PS
};

// Entry 0 of the page table maps to physical page 0, entry 1 to
// physical page 1, etc.
+#if 0
__attribute__((__aligned__(PGSIZE)))
pte_t entry_pgtable[NPTENTRIES] = {
    0x0000000 | PTE_P | PTE_W,
@@ -1056,4 +1057,4 @@ pte_t entry_pgtable[NPTENTRIES] = {
    0x3fe000 | PTE_P | PTE_W,
    0x3ff000 | PTE_P | PTE_W,
};
-
-#endif
diff --git a/kern/pmap.c b/kern/pmap.c
index 88608e7..b215169 100644
--- a/kern/pmap.c
+++ b/kern/pmap.c
@@ -106,7 +106,20 @@ boot_alloc(uint32_t n)
    // LAB 2: Your code here.

-    return NULL;
+    // Están mapeados menos de 4 MB
+    // por lo que no podemos pedir
+    // más memoria que eso
+    if ((uintptr_t)ROUNDUP(nextfree + n, PGSIZE) > (KERNBASE + (4 << 20))) {
+        panic("boot_alloc: out of memory");
+    }
+    result = nextfree;
+    if (n > 0) {
+        nextfree = ROUNDUP(nextfree + n, PGSIZE);
+    }
+    return result;
}

// Set up a two-level page table:
@@ -127,9 +140,6 @@ mem_init(void)
    // 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);
```

```

@@ -154,7 +164,9 @@ mem_init(void)
    // 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, 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
@@ -165,7 +177,12 @@ mem_init(void)
    check_page_free_list(1);
    check_page_alloc();

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

+
    //////////////////////////////////////
    // Now we set up virtual memory
@@ -178,6 +195,13 @@ mem_init(void)
    // - pages itself -- kernel RW, user NONE
    // Your code goes here:

+    // Mapeo en kern_pgdir, UVPT - UPAGES direcciones virtuales a partir de
UPAGES
+    // a direcciones físicas a partir de donde comienza el struct page info
pages.
+
+    //page_insert (pde_t *pgdir, struct PageInfo *pp, void *va, int perm)
+    //boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa,
int perm)
+    boot_map_region(kern_pgdir, UPAGES, ROUNDUP(npages * sizeof(struct
PageInfo), PGSIZE), 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.
@@ -189,6 +213,7 @@ mem_init(void)
    // 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 | PTE_P);

    //////////////////////////////////////
    // Map all of physical memory at KERNBASE.
@@ -198,6 +223,7 @@ mem_init(void)
    // we just set up the mapping anyway.
    // Permissions: kernel RW, user NONE
    // Your code goes here:
+    boot_map_region(kern_pgdir, KERNBASE, 0xffffffff - KERNBASE + 1, 0, PTE_W
| PTE_P);

    // Check that the initial page directory has been set up correctly.
    check_kern_pgdir();
@@ -239,6 +265,31 @@ mem_init(void)
void
page_init(void)
{
+    // Hay paginas prohibidas y paginas libres.
+    // Las páginas prohibidas son todas las que ya estan ocupadas hasta este
punto.
+    // Mas las que se indiquen en los comentarios en inglés de abajo.
+    // Las paginas prohibidas se ponen en 0 y en NULL
+    // En 0 porque si se intentan liberar tirará kernel panic
+    // Y en null porque no forman parte de la lista enlazada
+    // Entonces hay que enlazar todas las páginas menos las prohibidas
+    // Poniendolas en 0 (pues son libres) y enlazando los punteros
+    // Las ocupadas que habrá en el futuro si van a tener su valor en 1
+    // Pero su puntero estará en NULL pues no formaran mas parte de la lista
libre
+    // Hasta que sean liberadas.
+
+    // Rocomienda dato: Que el for que viene ya hecho, poner if (condicion)
continue;
+    // y luego las lineas originales de la funcion. Esa condicion es la que
me dice
+    // si es una página prohibida, osea if(prohibida)
+    // Una manera muy facil es decir:
+    /*

```

```

+         physaddr_t addr = 0
+         if (i = 1; i < npages; i++) { // i empieza en 1 para saltar la
primera página
+             if (addr >= boot_alloc(0) || addr < io_phys_mem) {
+                 entonces no es prohibida
+             }
+             addr += PGSIZE;
+         }
+     */
+     // 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.
@@ -252,15 +303,21 @@ page_init(void)
+     // 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?
+     // Aca empieza el kernel
+     // Estan ocupadas todas las paginas
+     // desde EXTPHYSMEM hasta boot_alloc(0)
+     // 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];
+     physaddr_t paddr;
+     for (size_t i = 1; i < npages; i++) {
+         paddr = i * PGSIZE;
+         if (paddr >= PADDR(boot_alloc(0)) || paddr < IOPHYSMEM) { // Si no
es una dirección prohibida
+             // pages[i].pp_ref = 0; // Fue seteado con memset
+             pages[i].pp_link = page_free_list;
+             page_free_list = &pages[i];
+         }
+     }
+ }
@@ -280,7 +337,21 @@ struct PageInfo *
page_alloc(int alloc_flags)
{
    // Fill this function in
-    return 0;
+    if (page_free_list) {
+        struct PageInfo * page = page_free_list;
+        page_free_list = page->pp_link;
+        page->pp_link = NULL;
+
+        if (alloc_flags & ALLOC_ZERO) {
+            // Seteamos a cero la pagina fisica
+            // no el struct PageInfo
+            memset(page2kva(page), 0, PGSIZE);
+        }
+
+        return page;
+    }
+    return NULL; // No free pages
+}
+
+//
@@ -293,6 +364,16 @@ 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) {
+        panic("page_free: try to free page with pp_link set\n");
+    }
+    if (pp->pp_ref) {
+        panic("page_free: try to free page with pp_ref's\n");
+    }
+    pp->pp_link = page_free_list;
+    page_free_list = pp;
+}
+
+//
@@ -328,11 +409,65 @@ page_decref(struct PageInfo *pp)
+    // Hint 3: look at inc/mmu.h for useful macros that manipulate page
+    // table and page directory entries.
+    //
+

```

```

+/*
+ Recibe siempre como parámetro un pde_t * que es un puntero a una tira de
1024 words de 4 bytes.
+ pde_t * es accesible con corchetes [].
+ Es una estructura que sirve de Page Directory. Cada entrada tiene 32
bits. Los 20 bits mas altos
+ son una dirección física donde se ubica la Page Table en particular. Los
12 bits restantes son
+ bits de presencia.
+
+ De la casilla saco la dirección física, la convierto en virtual y con
eso referencio la Page Table
+ que quiero.
+
+ Esta funcion es una funcion de soporte que permite llegar a la página que
interesa.
+ Hay que chequear si el bit de presencia esta a cero (en ese caso la
entrada de la page
+ directory no tendra nada). Si esta en cero y flag de create, hay que
alocar un page table y asignarselo
+ en esa posición con la dirección física de la page table alocada y
ponerle los bits que
+ corresponda.
+ Si aloca una pagina, hay que hacer pp_ref++ a cada
+
+ Retorna un puntero (direccion virtual) a la page table
+*/
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
{
- // Fill this function in
- return NULL;
+ // Obtengo la entrada en la PD sumando a pgdir el indice de la VA
+ pde_t * pde = pgdir + PDX(va);
+
+ if ((*pde & PTE_P)) {
+ // Obtengo la direccion virtual del PT base register
+ pte_t * ptbr = KADDR(PTE_ADDR(*pde));
+
+ // Si ya existe retornamos el PTE correspondiente
+ return ptbr + PTX(va);
+ } else if (create) {
+ // Si la page table buscada no está presente y el flag de create
esta activado
+ struct PageInfo * new_pt_page = page_alloc(ALLOC_ZERO);
+
+ if (!new_pt_page) {
+ memoria
+ return NULL; // Fallo el page alloc porque no había mas
+
+ // Obtengo la direccion física de la entrada a la page table
alocada
+ physaddr_t pt_phyaddr = page2pa(new_pt_page);
+
+ // Escribimos la direccion fisica y los flags correspondientes
+ *pde = (pt_phyaddr | PTE_P | PTE_W | PTE_U);
+
+ // Marco como referenciado la page info asociada a la pagina fisica
alocada para la page table
+ new_pt_page->pp_ref++;
+
+ // Obtengo la direccion virtual del PT base register
+ pte_t * ptbr = KADDR(PTE_ADDR(*pde));
+
+ // Devolvemos el puntero a PTE
+ return ptbr + PTX(va);
+ } else {
+ // No está presente la page table
+ // buscada y el flag de create está desactivado
+ return NULL;
+ }
+ }
+
+//
@@ -345,11 +480,48 @@ pgdir_walk(pde_t *pgdir, const void *va, int create)
+// above UTOP. As such, it should *not* change the pp_ref field on the
+// mapped pages.
+
+//
+// boot_map_region(kern_pgdir, UPAGES, npages, PADDR(pages), PTE_U | PTE_P);
+
+// 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
+ #ifndef TP1_PSE
+     assert(va % PGSIZE == 0);
+     assert(pa % PGSIZE == 0);
+     assert(size % PGSIZE == 0);
+     assert(perm < (1 << PTXSHIFT));
+     for (size_t i = 0; i < size/PGSIZE; i++, va+=PGSIZE, pa+=PGSIZE) {
+         pte_t *pte = pgdir_walk(pgdir, (const void *) va, 1);
+         *pte |= pa | perm | PTE_P;
+     }
+ #else
+     if (va % PTSIZE == 0 && size % PTSIZE == 0 && pa % PTSIZE == 0) {
+         // Es una Large Page
+         for (size_t i = 0; i < size/PTSIZE; i++, va += PTSIZE, pa +=
PTSIZE) {
+             // Obtengo la PDE
+             pde_t *pde = pgdir + PDX(va);
+             // Escribo la dirección física de la página larga en la PDE,
+             // seteando los flags perm, PTE_PS (large page) y PTE_P
+             (present)
+             *pde = pa | perm | PTE_PS | PTE_P;
+         }
+     } else {
+         // Es una Short Page
+         assert(va % PGSIZE == 0);
+         assert(pa % PGSIZE == 0);
+         assert(size % PGSIZE == 0);
+         assert(perm < (1 << PTXSHIFT));
+         for (size_t i = 0; i < size/PGSIZE; i++, va+=PGSIZE, pa+=PGSIZE) {
+             pte_t *pte = pgdir_walk(pgdir, (const void *) va, 1);
+             *pte |= pa | perm | PTE_P;
+         }
+     }
+ #endif
+ }

//
@@ -380,7 +552,35 @@ boot_map_region(pde_t *pgdir, uintptr_t va, size_t size,
physaddr_t pa, int perm
int
page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int perm)
{
    // Fill this function in
+ pte_t *pte = pgdir_walk(pgdir, va, 1);
+ if (pte == NULL) {
+     // pgdir_walk pudo fallar por falta de memoria
+     return -E_NO_MEM;
+ }
+ // Actualizamos el estado de PageInfo
+ // Antes de page_remove ya que esta funcion
+ // puede llegar a liberar la pagina si es la ultima
+ // referencia. Esto evita el caso borde
+ pp->pp_ref++;
+ if (*pte & PTE_P) {
+     // Si ya estaba ocupada la removemos
+     page_remove(pgdir, va);
+ }
+ // Obtenemos la direccion fisica del struct PageInfo
+ physaddr_t padrr = page2pa(pp);
+ // No hace falta el shift porque los 12 bits de padrr son 0
+ // pues las paginas estan alineadas a multiplos de 4096
+ // seteamos la direccion fisica y los permisos
+ *pte = padrr | perm | PTE_P;
+ // pp_link ya fue puesto a null en la llamada
+ // correspondiente a page_alloc
+ return 0;
+ }

@@ -395,11 +595,32 @@ page_insert(pde_t *pgdir, struct PageInfo *pp, void *va,
int perm)
//
// Hint: the TA solution uses pgdir_walk and pa2page.

```

```

//
+
+/*
+ Dada una dirección virtual nos da un PageInfo
+ pgdir_walk(VA) = direccion virtual de la entrada a la página
+ pte_t * p = pgdir_walk(va)
+ phys_f = PTE_ADR(*p) // me da la dirección física
+ pa2page(f) -> Me retorna la página de la dirección física y retornamos
esto
+*/
+
+ struct PageInfo *
+ page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
+ {
+     // Fill this function in
+     return NULL;
+     pte_t * pte = pgdir_walk(pgdir, va, 0);
+     if (pte == NULL || !(*pte & PTE_P)) {
+         // No hay pagina mapeada para va
+         return NULL;
+     }
+     if (pte_store) {
+         // Guardamos en pte_store la direccion de PTE
+         *pte_store = pte;
+     }
+     physaddr_t page_paddr = PTE_ADDR(*pte);
+     return pa2page(page_paddr);
+ }
+
+//
+@@ -417,10 +638,29 @@ page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
+// Hint: The IA solution is implemented using page_lookup,
+// tlb_invalidate, and page_decref.
+//
+
+/*
+ Recibe un VA y hace dos cosas:
+ - decref(pagina) (es una función que ya esta implementada)
+   decrementa el pageref y si queda en cero llama a free de la pagina
+   automaticamente.
+ - limpiar PTE (Pone la page table entry a cero)
+*/
+ void
+ page_remove(pde_t *pgdir, void *va)
+ {
+     // Fill this function in
+     pte_t * pte;
+     // Conseguimos el struct PageInfo asociado y guardamos su PTE
+     struct PageInfo * page_to_remove = page_lookup(pgdir, va, &pte);
+     // Decrementamos pp_ref y liberamos si es necesario
+     page_decref(page_to_remove);
+     // Escribimos PTE en 0
+     *pte = 0;
+     // Realizamos la invalidacion de la entrada de la TLB
+     tlb_invalidate(pgdir, va);
+ }
+
+//
+@@ -671,7 +911,6 @@ check_page(void)
+ void *va;
+ int i;
+ extern pde_t entry_pgdir[];
+
+ // should be able to allocate three pages
+ pp0 = pp1 = pp2 = 0;
+ assert((pp0 = page_alloc(0)));

```