Trabajo Práctico 1: Memoria virtual en JOS

Integrantes:

Cotarelo, Rodrigo 98577

Longo, Nicolás 98271

75.08 - Sistemas Operativos – Cátedra Méndez

2do cuatrimestre 2018

05/10/2018

static void \*

boot\_alloc(uint32\_t n)

{

static char \*nextfree; // virtual address of next byte of free memory

char \*result;

// Initialize nextfree if this is the first time.

// 'end' is a magic symbol automatically generated by the linker,

// which points to the end of the kernel's bss segment:

// the first virtual address that the linker did \*not\* assign

// to any kernel code or global variables.

if (!nextfree) {

extern char end[];

nextfree = ROUNDUP((char \*) end, PGSIZE);

}

// Allocate a chunk large enough to hold 'n' bytes, then update

// nextfree. Make sure nextfree is kept aligned

// to a multiple of PGSIZE.

//

// LAB 2: Your code here.

cprintf("boot\_alloc memory at %x\n", nextfree);

cprintf("Next memory at %x\n", ROUNDUP((char \*) (nextfree+n), PGSIZE));

if (n != 0) {

char \*next = nextfree;

nextfree = ROUNDUP((char \*) (nextfree+n), PGSIZE);

return next;

}

else

{

return nextfree;

}

return NULL;

}

void

mem\_init(void)

{

uint32\_t cr0;

size\_t n;

// Find out how much memory the machine has (npages & npages\_basemem).

i386\_detect\_memory();

// Remove this line when you're ready to test this function.

//panic("mem\_init: This function is not finished\n");

//////////////////////////////////////////////////////////////////////

// create initial page directory.

kern\_pgdir = (pde\_t \*) boot\_alloc(PGSIZE);

memset(kern\_pgdir, 0, PGSIZE);

//////////////////////////////////////////////////////////////////////

// Recursively insert PD in itself as a page table, to form

// a virtual page table at virtual address UVPT.

// (For now, you don't have understand the greater purpose of the

// following line.)

// Permissions: kernel R, user R

kern\_pgdir[PDX(UVPT)] = PADDR(kern\_pgdir) | PTE\_U | PTE\_P;

//////////////////////////////////////////////////////////////////////

// Allocate an array of npages 'struct PageInfo's and store it in

// 'pages'.

// The kernel uses this array to keep track of physical pages: for

// each physical page, there is a corresponding struct PageInfo in this

// array. 'npages' is the number of physical pages in memory. Use

// memset

// to initialize all fields of each struct PageInfo to 0.

// Your code goes here:

pages = (struct PageInfo \*) boot\_alloc(sizeof(struct PageInfo) \* npages);

cprintf("npages: %d\n", npages);

cprintf("npages\_basemem: %d\n", npages\_basemem);

cprintf("pages: %x\n", pages);

//////////////////////////////////////////////////////////////////////

// Now that we've allocated the initial kernel data structures, we set

// up the list of free physical pages. Once we've done so, all further

// memory management will go through the page\_\* functions. In

// particular, we can now map memory using boot\_map\_region

// or page\_insert

page\_init();

check\_page\_free\_list(1);

check\_page\_alloc();

check\_page();

//////////////////////////////////////////////////////////////////////

// Now we set up virtual memory

//////////////////////////////////////////////////////////////////////

// Map 'pages' read-only by the user at linear address UPAGES

// Permissions:

// - the new image at UPAGES -- kernel R, user R

// (ie. perm = PTE\_U | PTE\_P)

// - pages itself -- kernel RW, user NONE

// Your code goes here:

boot\_map\_region(kern\_pgdir, UPAGES, PTSIZE, PADDR(pages) , PTE\_U | 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:

boot\_map\_region(kern\_pgdir, KSTACKTOP-KSTKSIZE, 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:

boot\_map\_region(kern\_pgdir, KERNBASE, ROUNDDOWN( (2^32) - KERNBASE, PGSIZE),

0x0, PTE\_W | PTE\_P);

// Check that the initial page directory has been set up correctly.

check\_kern\_pgdir();

// Switch from the minimal entry page directory to the full kern\_pgdir

// page table we just created. Our instruction pointer should be

// somewhere between KERNBASE and KERNBASE+4MB right now, which is

// mapped the same way by both page tables.

//

// If the machine reboots at this point, you've probably set up your

// kern\_pgdir wrong.

lcr3(PADDR(kern\_pgdir));

check\_page\_free\_list(0);

// entry.S set the really important flags in cr0 (including enabling

// paging). Here we configure the rest of the flags that we care about.

cr0 = rcr0();

cr0 |= CR0\_PE | CR0\_PG | CR0\_AM | CR0\_WP | CR0\_NE | CR0\_MP;

cr0 &= ~(CR0\_TS | CR0\_EM);

lcr0(cr0);

// Some more checks, only possible after kern\_pgdir is installed.

check\_page\_installed\_pgdir();

}

// --------------------------------------------------------------

// Tracking of physical pages.

// The 'pages' array has one 'struct PageInfo' entry per physical page.

// Pages are reference counted, and free pages are kept on a linked list.

// --------------------------------------------------------------

void

page\_init(void)

{

size\_t i;

for (i = 1; i < npages\_basemem; i++) { //Este for lo que hace es generar la lista enlazada de las paginas. pages[i].pp\_link guarda la direccion de la pagina anterior (pages[i-1])

pages[i].pp\_ref = 0;

pages[i].pp\_link = page\_free\_list;

page\_free\_list = &pages[i];

}

int med = PGNUM(PADDR(boot\_alloc(0)));

cprintf("pageinfo size: %d\n", sizeof(struct PageInfo));

cprintf("%x\n", ((char\*)pages) + (sizeof(struct PageInfo) \* npages));

cprintf("med=%d\n", med);

for (i = med; i < npages; i++) { //Este for continua linkeando en la lista enlazada.

pages[i].pp\_ref = 0; //Se setea pp\_ref en 0 para indificar que la pagina esta libre

pages[i].pp\_link = page\_free\_list;

page\_free\_list = &pages[i];

}

}

struct PageInfo \*

page\_alloc(int alloc\_flags)

{

// Fill this function in

if (page\_free\_list) {

struct PageInfo \*ret = page\_free\_list; //Linea 310 y 311 sacan la pagina que esta como cabeza de la lista enlazada y dejan a la anterior como cabeza.

page\_free\_list = page\_free\_list->pp\_link; //Si pone los flags correctos, seteo la pagina que quiero alocar (la cabeza de la lista) toda en cero.

if (alloc\_flags & ALLOC\_ZERO)

memset(page2kva(ret), 0, PGSIZE);

return ret;

}

return NULL;

}

void

page\_free(struct PageInfo \*pp)

{

pp->pp\_link = page\_free\_list; //Lo que hago es poner esta pagina como cabeza de la lista enlazada. Pp pasa a ser la nueva cabeza de la lista, por lo que apunta a la que era la ultima pagina.

page\_free\_list = pp; //Ahora pp es la ultima pagina de la lista enlazada, por eso la guardo en page\_free\_list.

}

pte\_t \*

pgdir\_walk(pde\_t \*pgdir, const void \*va, int create)

{

pte\_t\* virtual\_pte;

pde\_t pde = pgdir[PDX(va)];

if( (!pde) || !(pde & PTE\_P) )

{

if(!create) // no crearlo

return NULL;

struct PageInfo\* pp = page\_alloc(ALLOC\_ZERO);

if(pp == NULL) // falló page alloc

return NULL;

pp->pp\_ref++;

// PTE\_ADDR se queda con los 20 bits de la direccion fisica

// de la page table entry

pde = PTE\_ADDR( page2pa(pp) ) | PTE\_P | PTE\_W | PTE\_U;

pgdir[PDX(va)] = pde;

}

// Kaddr( PTE\_ADDR ..) me devuelve la direccion virtual del comienzo

// de la pte.

virtual\_pte = KADDR( PTE\_ADDR(pde) );

return virtual\_pte + PTX(va);

}

boot\_map\_region(pde\_t \*pgdir, uintptr\_t va, size\_t size, physaddr\_t pa, int perm)

{

// Fill this function in

int cant\_paginas = size/PGSIZE;

for(int i = 0; i < size; i+= PGSIZE)

{

uintptr\_t virtual = va + i;

physaddr\_t physical = pa + i;

pte\_t \*pag = pgdir\_walk(pgdir, (const void\*) virtual , 1);

\*pag = physical | perm | PTE\_P;

}

}

int

page\_insert(pde\_t \*pgdir, struct PageInfo \*pp, void \*va, int perm)

{

pte\_t\* pte = pgdir\_walk(pgdir, va, 1); // la creo si es necesario

if(pte == NULL) // me quedé sin memoria

return -E\_NO\_MEM;

pp->pp\_ref++;

// aumento el contador ahora, por si en el próximo paso resulta

// que estoy borrando la misma página que quiero insertar

// (de ésta manera no se le hará free a la página)

if(\*pte & PTE\_P) // VA previamente mappeada

page\_remove(pgdir, va);

physaddr\_t pa\_page = PTE\_ADDR( page2pa(pp) );

\*pte = pa\_page | perm | PTE\_P;

return 0;

}

struct PageInfo \*

page\_lookup(pde\_t \*pgdir, void \*va, pte\_t \*\*pte\_store)

{

pte\_t\* pte = pgdir\_walk(pgdir, va, 0);

if( (pte == NULL) || !( \*pte & PTE\_P) ) // verificar bits de Presente y eso

return NULL;

if(pte\_store != NULL)

\*pte\_store = pte;

physaddr\_t pa = PTE\_ADDR(\*pte);

return pa2page(pa);

}

void

page\_remove(pde\_t \*pgdir, void \*va)

{

pte\_t\* pte;

struct PageInfo\* pp = page\_lookup(pgdir, va, &pte);

if(pp == NULL) // "silently does nothing"

return;

pp->pp\_link = page\_free\_list; // para el caso de la reinserción!

page\_decref(pp);

\*pte = 0;

tlb\_invalidate(pgdir, va);

}

TP1.md:

TP1: Memoria virtual en JOS

===========================

page2pa

-------

La función recibe un puntero a un struct PageInfo. En Jos, se utiliza ésta dirección de memoria en conjunto con la del comienzo del arreglo pages para deducir la dirección física.

En primer lugar, es necesario mencionar que el arreglo pages posee un puntero al comienzo de la lista de páginas. Si a una página cualquiera le restamos el comienzo del arreglo, obtenemos la dirección relativa.

En segundo lugar, este arreglo está almacenado en memoria física cuyo mapeo en memoria virtual es correlativo (1 a 1). Esto quiere decir que si restamos una página determinada con el inicio de la lista obtenemos la posición en memoria física de dicha página.

boot\_alloc\_pos

--------------

a)

$ nm kernel

f0115970 B end

Por lo tanto, la primera dirección que boot\_alloc devolverá será 0xf0116000

b)

(gdb) b boot\_alloc

Punto de interrupción 1 at 0xf01009a6: file kern/pmap.c, line 89.

(gdb) c

Continuando.

Se asume que la arquitectura objetivo es i386

Breakpoint 1, boot\_alloc (n=65684) at kern/pmap.c:89

89 {

=> 0xf01009af <boot\_alloc+9>: cmpl $0x0,0xf0115538

98 if (!nextfree) {

(gdb) p nextfree

$1 = 0x0

=> 0xf01009fc <boot\_alloc+86>: mov $0xf011696f,%eax

100 nextfree = ROUNDUP((char \*) end, PGSIZE);

(gdb) p (char\*) &end

$9 = 0xf0115970 "\022"

(gdb) p nextfree

$4 = 0xf0116000 ""

page\_alloc

----------

La función page2pa recibe un puntero a un struct PageInfo y devuelve su dirección física. La función page2kva devuelve la dirección virtual mapeada (por encima del Kernel, usando la macro KADDR) a la dirección física que contiene a dicha página.