Links

http://csapp.cs.cmu.edu/2e/samples.html Capitulo 8.5
https://www.kea.nu/files/textbooks/ospp/osppv2.pdf Capitulo 8

Introducción a arranque

Los SO tienen que estar implementados (compilados para correr) sobre un hardware. Muchas de las cosas que hace el SO tiene que ver con la funcionalidad del hardware por lo que depende del mismo de las funcionalidades que va a tener el SO. El SO va a depender de la arquitectura.

x86 es un procesador Intel de 32 bits.
x86_64 es la misma arquitectura extendida a 64 bits.

Cuando arranca un procesador x86 lo hace con una cantidad de features mínima que es lo que se conoce como modo real. Arranca con registros de 16 bits, con 1 MB de direccionamiento de memoria virtual, con features de paginación y segmentación limitadas porque es como lo hacía uno de los primeros procesadores. Justamente, tiene que hacer esto porque queremos que sea compatible hacia atrás.

Modo real

Por compatibilidad hacia atrás, todo x86 arranca en modo real.
Proceso de arranque:

1) Arranca en una etapa de firmware que le dice al sistema como debe arrancar.
    a) Se pone el instruction pointer en ROM donde se guarda el firmware.
    b) Se hacen algunos chequeos cómo inicializar hardware y la integridad de los periféricos. También cede el control al OS, pero hasta el momento no hay nada que ceder.
2) Para ceder el control al OS se usan interfaces, como BIOS y UEFI. Es una especificación y su algoritmo es ejecutado por firmware.
3) Siguiendo con BIOS, 512 bytes es muy poco para cargar lo que sea. Se carga una función cuya responsabilidad es cargar memoria. Este proceso se conoce como **bootloader**.

Jos tiene su propio bootloader

qemu tiene su propia bios

Preguntas del tp:

1) ¿Dónde se encuentra el código del bootloader?
    a) El código del bootloader se encuentra en la carpeta /boot
    b) Físicamente está en el primer sector del disco para simular BIOS (1 MiB)
2) Donde empieza a cargar el kernel?
    a) A partir de la dirección 1MB
3) Si preguntan la dirección de alguna cosa relacionada al kernel, se puede correr nm ./obj/kern/kernel

Modo protegido

1) La funcionalidad del bootloader justamente es cargar un kernel en memoria y pasar de modo real a modo protegido. Este proceso puede ser segmentado en distintas etapas cargando otros bootloaders, hasta que recursivamente cargará al kernel.
2) Una vez que finaliza pone un Instruction pointer en alguna posición que permita al kernel funcionar. Tiene múltiples especificaciones, entonces el bootloader sabe encontrar como bootear el propio sistema en cuestión.
    a) Usaremos un bootloader de JOS.
3) Kernel: El software principal del OS.
    a) Puede tener diferentes fases de arranque (inicialización de subsistemas).
    b) Tiene completo control de lo que haga de ahora en más.
    c) Lanza el proceso init
4) Init: empieza a levantar servicios, interfaces gráficas, gestores de dispositivos. Pero siempre en modo usuario.

Bios

1) Itera todos los dispositivos reconocidos en etapa post y los detecta.
2) Encuentra un booteable y lo carga en el primer sector del dispositivo (512 bytes).
3) Pone el instruction pointer en la dirección 0x7c00, siempre en la misma.

entry.S

1. Carga el entry pgdir en CR3
2. Habilitamos la paginación
    a. Largas en CR4
    b. Chicas en CR0
3. Hace un jump del kernel, al instruction pointer del kernel en memoria alta para seguir ejecutando.

Inicialización de la memoria

Una vez de haber booteado, ¿qué vamos a querer hacer con la memoria antes que nada? En el TP1 vamos a terminar con un kernel que opera en modo kernel y no crea ningún proceso pero inicializa toda la memoria.
Hay 2 cosas que queremos hacer con la memoria. La primera es saber cuánta memoria hay, qué parte de esa memoria está ocupada y qué parte no. Recordar que el kernel está mapeado en direcciones físicas por lo que una parte de la memoria es ocupada por el kernel.

JOS usa paginación por lo que necesitamos soporte de hardware. En x86 las páginas son de 4 KiB. Es como si agarraras los 4 GB del espacio de direcciones y lo dividieras en bloques de 4 KiB. Alguno de esos bloques son páginas. Tenemos que saber cuáles de esas páginas están libre y cuáles están ocupadas. Para esto, JOS va a tener una estructura en memoria que es una arreglo de páginas. Un array de C en el que cada elemento representa una página física del espacio de direcciones físico. Este arreglo es dinámico ya que no

conocemos la memoria disponible que tenemos por lo que tiene un asterisco, caso contrario tendría corchetes.

Entonces tenemos un arreglo de páginas struct PageInfo *pages compuesto por elementos de tipo struct PageInfo que representa a cada página del arreglo.
Notar que página física equivale a decir pageframe.

Variables

npages

Es la cantidad de páginas detectadas en el sistema, son asignadas por i386_detect_memory. Luego se usa para: alocar la memoria y después para mapearla. La memoria alocada se usa para guardar el array de PageInfo. Para el mapeo de las páginas se usa el npages * sizeof(PageInfo)

npages_basemem

Es la dirección virtual donde arranca la lista de páginas a alocar.

UPAGES

Esta variable es usada para darle un espacio accesible solamente de lectura al usuario de las páginas previamente detectadas y booteadas.

UTOP

PageInfo

Entonces tenemos un arreglo de páginas struct PageInfo *pages compuesto por elementos de tipo struct PageInfo que representa a cada página del arreglo.

Y el struct PageInfo *pp_link es como si fuera una lista enlazada porque apunta a otro elemento del struct.

page_free_list

Array de páginas libres

end

end → variable global que guarda la dirección en donde termina el kernel

KERNBASE

KERNBASE → es la base del kernel. Si bien el kernel no comienza estrictamente en KERNBASE, resulta ser el límite entre el kernel con lo que está debajo del mismo (vamos a ver que hay regiones debajo del kernel que pertenecen al usuario).

Macros

PADDR(kva)

Convierte la dirección virtual kva a dirección física.

KADDR(pa)

Convierte la dirección física pa a dirección virtual.

Funciones

mem_init()

Lo primero que hace es llamar a i386_detect_memory() que a partir de interfaces de hardware obtiene cuántas páginas de memoria física hay en el sistema. Es decir, tengo 4 GB que son direccionables, algunas de esas páginas están tomadas por entrada y salida pero del resto no todas están mapeadas a memoria. Con QEMU vamos a correr simulando una máquina con 128 MB de RAM. Por lo que esto va a detectar los 128 MB y nos va a devolver npages y npages_basemem nos devuelve cuánta memoria total tenemos. Estas 2 variables son globales.

El struct contiene un puntero a otro struct y un contador de referencias. Si este es el struct que vamos a guardar para representar a la página, ¿dónde está la dirección física de la página? Es su índice en el arreglo, no necesitamos guardarlo porque si calculo el índice podemos saber qué página es. La página 0 va a estar en el 4k byte * índice 0, la página 1 en el 4k byte * índice 1 y así sucesivamente.

Entonces el contador de referencia indica cuántos mapeos hacen referencia a esta página física. Y el struct PageInfo *pp_link es como si fuera una lista enlazada porque apunta a otro elemento del struct, que está libre (ver struct PageInfo linea 176) y difiere de la lista de pages, que es un array.

En el kernel hay estructuras de datos que tienen doble significado. En el caso del arreglo pages es ambas cosas. Es un arreglo porque los elementos están guardados uno detrás de otro indexados. Y además sobre ese arreglo cada posición es un nodo en una lista enlazada.

Es decir, tenemos el arreglo pages que apunta a un arreglo de páginas y además sobre ese arreglo vamos a tener una lista enlazada que contiene únicamente páginas libres. Hay una variable global (page_free_list) que va a apuntar siempre a la cabecera de esa lista (que contiene únicamente paginas libres). Las páginas ocupadas apuntan a NULL en el campo

de la lista enlazada. Esto es la definición de la implementación. Por lo tanto, siempre va a devolver la primera página libre que encuentre. Es más eficiente devolver la cabecera de la lista que recorrer todo el arreglo.

Al comienzo de mem_init() se reserva una página para el page directory del kernel, y se guarda en la variable global kern_pgdir. Al final de la función, se carga en %cr3 en sustitución del usado en el proceso de arranque.
En esta función, también se llama 3 veces a boot_map_region() para configurar:

- el stack del kernel en KSTACKTOP
- el arreglo pages en UPAGES
- los primeros 256 MiB de memoria física en KERNBASE

boot_alloc()

Se va a usar únicamente en esta etapa(inicialización de memoria): un kernel muy temprano que todavía no configuró memoria. Solo lo vamos a utilizar para guardar memoria para estructuras vitales como por ejemplo el arreglo de pages.
Por lo que para alocar memoria para una estructura corremos la línea tantos bytes como necesitemos y devolvemos esa región de memoria como un puntero que podemos utilizar.

Recibe una cantidad de bytes y reserva esa cantidad de bytes después de la imagen del kernel. Mantiene un puntero a la última dirección ocupada o a la primera libre, corre ese límite una cierta cantidad de bytes, lo actualiza y devuelve un puntero a la dirección que quedó en el medio.

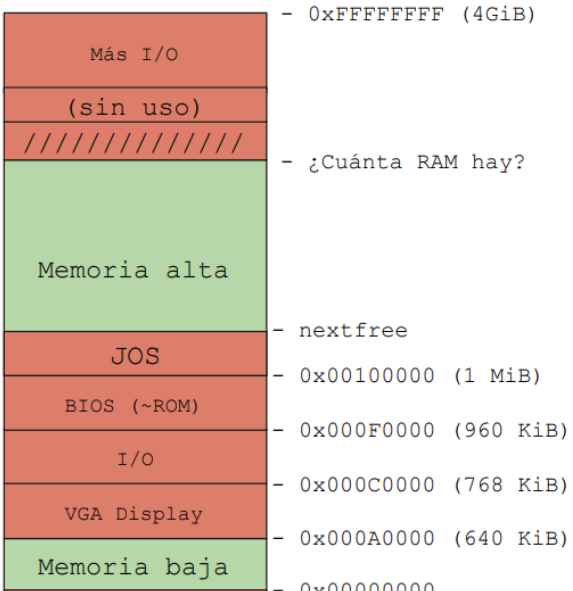
Una de las cosas que pide esta función es chequear si hay memoria, si no la hay podemos usar la función panic que sirve para crashear el sistema porque si no hay memoria física para usar este struct estamos en serios problemas.

Conocemos la cantidad de memoria física porque sabemos dónde estamos mapeados, cuánta memoria física total hay (está en npages). Tenemos que hacer la cuenta para convertir de la dirección virtual a la física, fijarnos si se pasa de npages y si no podemos devolver esa memoria.
Entonces boot_alloc() es un reservador de memoria primitiva.

page_init()

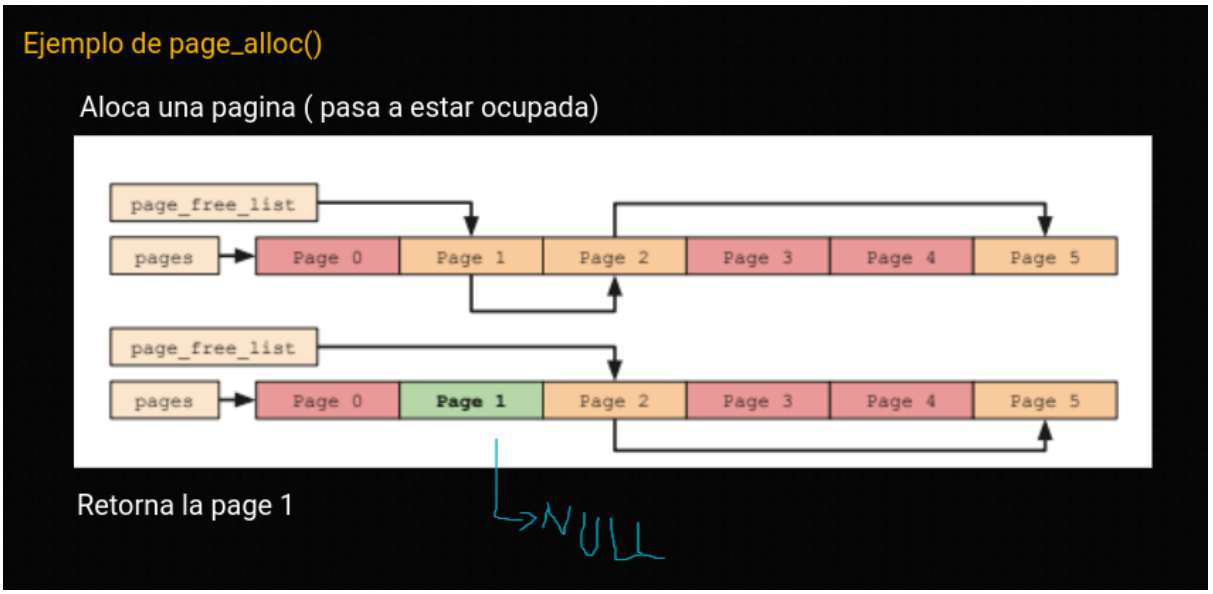
Cada struct page está alocado. Recorre el struct pages* entero y lo marca como todo libre.

Esto es memoria física. Queremos ver cuál está ocupada y cuál no. La primera página física (página 0) tiene a nuestro bootloader, es decir, código que ya está mapeado por lo



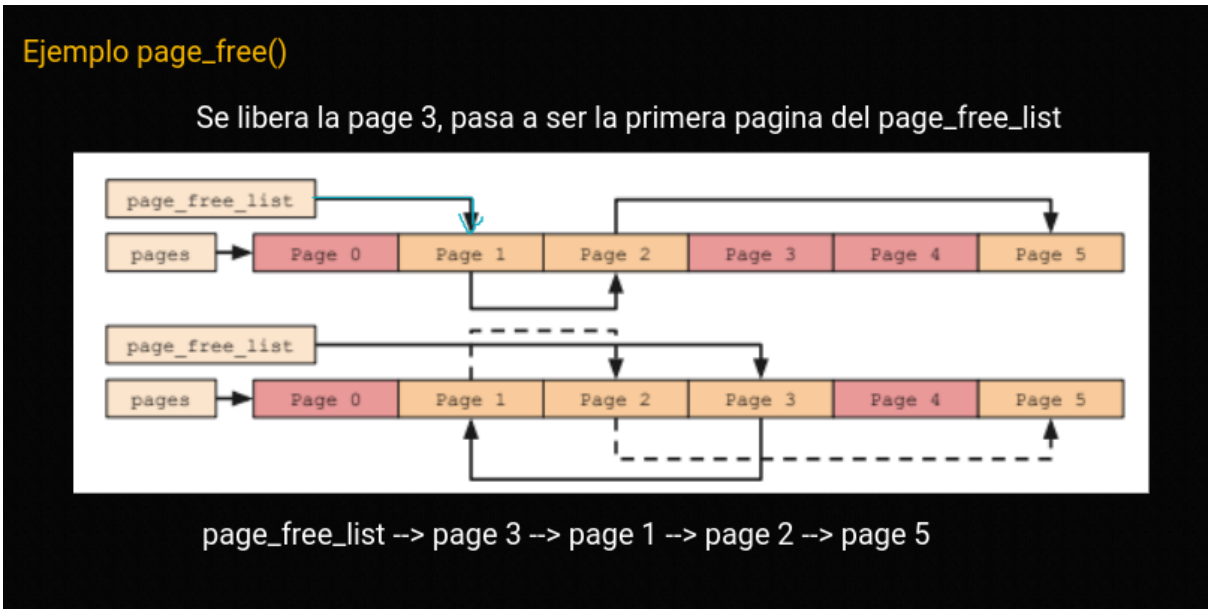
cual está ocupada. Después hay una región de memoria baja que en principio está libre. No hay nada porque el kernel está mapeado en la dirección 1 MB. Entre la memoria alta y la baja hay una región de páginas usadas. Arriba tenemos páginas libres. Para saber cuánta RAM hay tenemos la variable npages y así obtener este valor. A partir del diagrama podemos ver qué páginas están ocupadas.

page_alloc()



Devuelve un puntero a un elemento del array pages que es una página libre. No nos dice para qué se va a usar sino que está libre y se puede usar. Con los flags podemos controlar si la limpia con ceros o la devuelve sin tocar.

page_free()



Recibe una página y la marca como libre.

¿Cómo se obtiene y libera una página libre?
Insertar y remover de la page_free_list.
Si se libera una página, la insertamos al inicio de la lista.
Si se aloca la primera página libre, la lista va a apuntar a la siguiente dirección (que apunta la primera página).

Solo se libera si no tiene referencias a la misma.

page_insert()

Crea un mapeo entre una dirección virtual y una física. La variable pp representa una página física. Tiene que llegar a la posición de la page table en donde uno tiene que guardar la dirección física a la cual va a hacer referencia la dirección virtual. Queremos obtener la dirección física que representa al puntero pp que no es la dirección física en la que está dicho puntero.

¿Qué sucede si ya había una página mapeada a esa posición? Puede pasar que para una cierta dirección virtual ya haya un espacio físico.
Esta función también tiene que invalidar a la TLB. La TLB es una caché que guarda traducciones (mapeos entre direcciones física y virtuales). Si se hace un nuevo mapeo es posible que la TLB quede en estado inválido. Esto se deja para la función page_remove().
Con el valor que devuelve pgdir_walk(), esta función termina haciendo el mapeo que corresponde.

—
versión Nacho:

Mapea una physical page a una virtual address. Hace lo mismo que boot_map_region, pero para una sola página común. Contempla el caso en el que dependiendo si la página ya estaba mapeada, la remueve y la vuelve a mapear.

page_lookup()

Se encarga de ir a ver qué tiene la page directory en la dirección va. Es muy parecida a pgdir_walk() solo que le agrega cositas (no crea nada) y devuelve un puntero a un struct PageInfo.

La variable pte_store (si es !=0) devuelve la información que hay en la entrada a la page table a la cual hace referencia el PageInfo. Es decir, deja la información que te devuelve pgdir_walk(). Esto es utilizado por page_remove y se puede usar para verificar los permisos de la página para los argumentos de syscalls, pero no debe ser utilizado por la mayoría de las personas que llaman.

—

page_lookup(): encuentra y devuelve el entry de la tabla de una página. Como una especie de pgdir_walk() pero sin crear nada.

```
PTE_ADDR(*pte) devuelve la dirección física
return pa2page(PTE_ADDR(*pte)); devuelve el contenido de la página de la dirección física
```

page_remove()

Hace un unmap de una dirección física de la dirección virtual que llega por parámetro. Si no hay una página física en esa dirección, no hace nada.
Lo busca con page_lookup, lo sobreescribe todo con 0 y decrementa el contador de pages.
La TLB debe invalidarse si se elimina una entrada de la tabla de páginas.
Finalmente, la función no devuelve nada.

La CPU genera una dirección lógica, la segmentación la transforma en una dirección lineal y la paginación la transforma en la dirección física.

boot_map_region()

Recibe un page directory, va y pa son direcciones de comienzo, el size de la región a mapear y los permisos.
Genera un mapeo estático de una región x.

Es similar a page_insert() en tanto que escribe en el PTE correspondiente, pero:

- no incrementa el contador de referencias de las páginas
- puede ser llamada con regiones mucho mayores de 4KiB

Esta función sólo está destinada a configurar las asignaciones ``estáticas'' por encima de UTOP. Como tal, **no** debe cambiar el campo pp_ref en las páginas asignadas.

Usar *large pages* oportunisticamente:

El uso de large pages en x86 requiere que, tanto dirección física como virtual estén alineadas a 4MiB. Es posible que los valores iniciales de pa y va en la función no estén alineados a 4MiB (y solo a 4KiB). Pero en ese caso, y si el parámetro size es lo suficientemente grande, llegará un momento en que sí lo estén. En ese momento, si la cantidad restante de memoria es mayor o igual a 4MiB, se debe usar una large page.

pgdir_walk()

Navega el page directory. Es decir, va a hacer lo que hace la MMU (estructura del computador). Va a ir al primer nivel → indexar con el offset → arma la dirección → ir a la page table correspondiente → indexarla → buscar la dirección física.

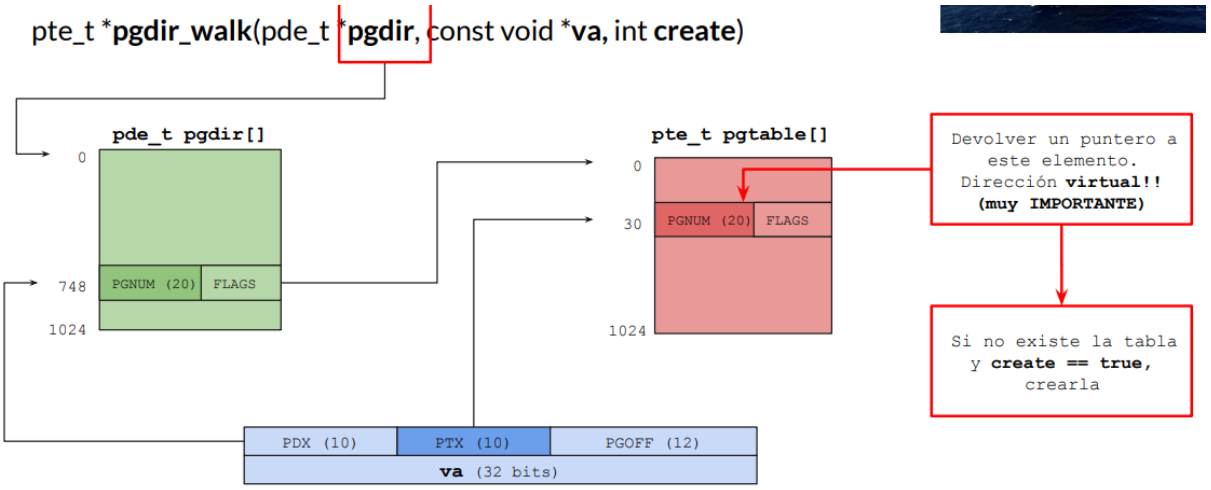
Agarra cualquier page directory. El contenido de la entrada del page directory es fisica. Esta dirección física apunta a la page table correspondiente. Pero nosotros queremos acceder a esta page table desde código C por lo que vamos a tener que convertir la dirección física en

virtual. No podríamos acceder a la page table a partir de la dirección física porque está la MMU y va a querer realizar la traducción igual.

Si no existe la tabla intermedia, es decir, la page table se debería crear. Si la variable create es true. Esto es porque podría no estar una page table porque no se accede a una dirección contenida en la misma. Lo que siempre va a existir es el page directory. Inicialmente lo que hay es una page directory, a medida que se van haciendo las traducciones se van creando las page tables.
Esta es la ventaja de este tipo de paginación, no se tiene todo el array de traducciones. Podemos determinar que hay o no algo en la dirección que contiene la page directory usamos los flags. Siempre devuelve un puntero a donde debería estar la dirección de la page table aunque no exista.

Esta función no hace un mapeo. Solo crea una page table en caso de que no exista. Es una función de acceso, búsqueda. Le paso una dirección virtual, vas a la dirección de la page table, me devuelve un puntero a dónde está la página física que uno quiere traducir. Si no existe la page table se devuelve el puntero a donde debería estar la posible página física.

Ayuda del enunciado: tanto page directories cómo page tables almacenan direcciones físicas (ya que es la MMU quien las procesa). La función debe devolver una dirección virtual.



Los flags de las entradas en la page directory solo se pueden tocar en pgdir_walk(). Para ser lo más permisivo debería validar todos los flags.

___

Páginas grandes:

Si consumo muchas páginas de 4k, tengo que hacer muchos accesos a memoria y cuesta todo el trabajo de la MMU.

Podríamos tener una página de 4MB. 4MB es lo que mapea una page table. Con esto nos podríamos saltear la página intermedia siendo mucho más eficientes, aunque con más coste en memoria.

Cuando la página es grande damos los primeros 10 bits que indexan el page directory y los otros 22 bits llevan directamente a la dirección física. Devuelta, **sin intermediario**. Esta página física, será de 4MB y por eso el offset es de 22 bits. 2^22 = 4MB.

Hay un nuevo flag en la primera tabla que es PTE_ES, cuando es 1, se debe aplicar el indexado de una página grande. Si está en 0, se hará una traducción normal.