Arranque y manejo de memoria

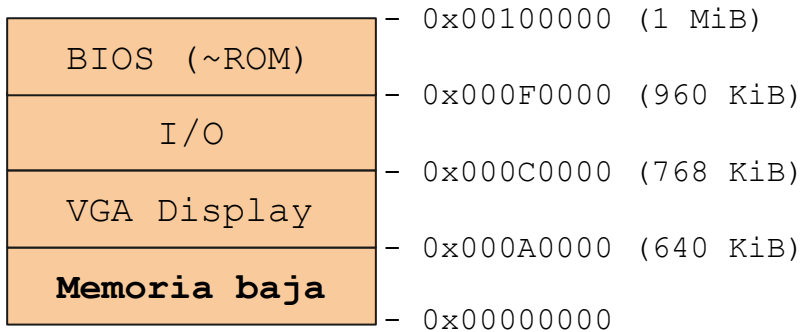
Caso de estudio: JOS

Sistemas Operativos - FIUBA

Introducción a arranque (en x86)

Arranque en x86 - modo real

- Por **compatibilidad hacia atrás**, todo x86 arranca en **modo real**
  - Igual que el 8086 original
- Registros de 16-bits
- 20-bits para direcciones de memoria
  - 1 MiB total!
  - Direccionado con *segmento + offset*
- El espacio de direcciones tenía reservado regiones para dispositivos

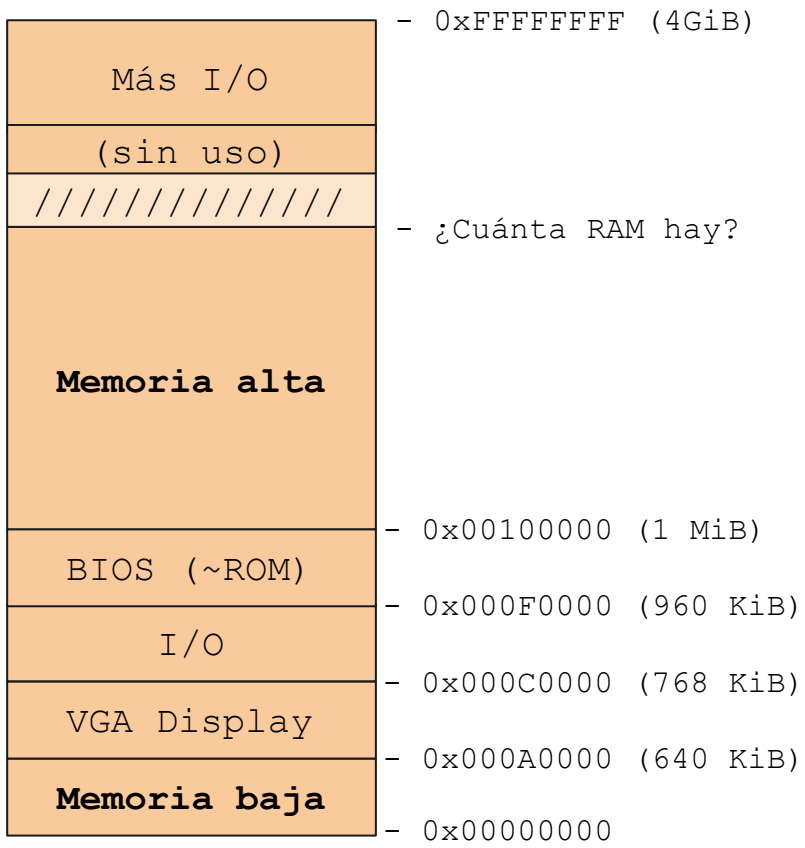


Arranque en x86 - modo protegido

- Aplica a procesadores con 32-bits (e incluso los modernos con 64-bits)
  - Nuevas formas de direccionamiento de memoria! (Segmentación, Paginación)
  - Más bits en los registros de propósito general
- Pero cuando arrancan lo hacen en **modo real**
- Es necesario **activar** las funcionalidades modernas mediante **registros de control**
  - Durante el arranque se pasa de **modo real** a **modo protegido**

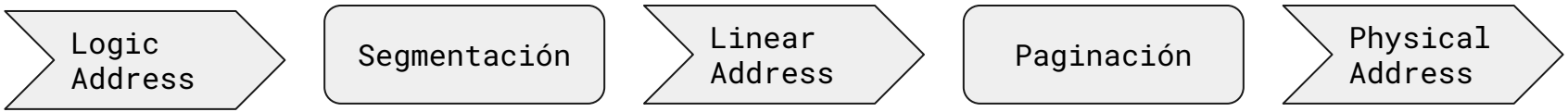
Arranque en x86 - memoria física

- Mapa para procesadores con 32-bits de **espacio de direcciones**
- “Hueco” alrededor de 1MiB por compatibilidad hacia atrás
- Bloques especiales
  - Mapeados a dispositivos
  - Mapeados a RAM real
  - Sin mapear (si tenemos menos que 4 GiB)



Arranque en x86 - direccionamiento

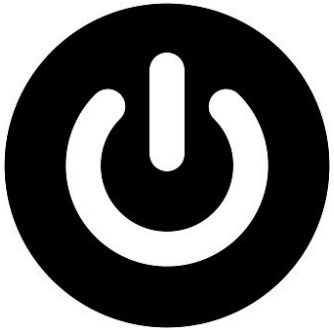
- El mecanismo de **segmentación** siempre está presente
- El mecanismo de **paginación** se habilita manualmente (**cr3**)
- Ambos se tienen que **configurar**



Arranque de un sistema operativo

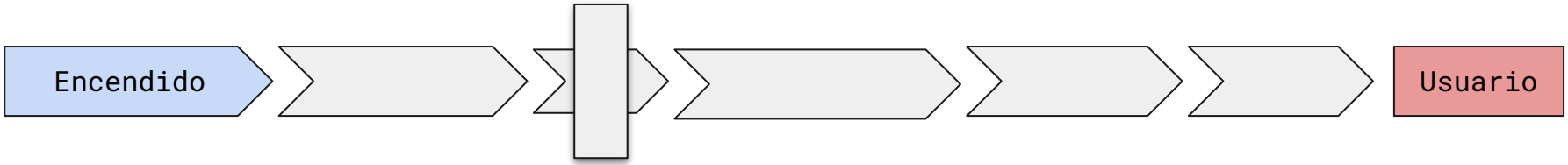
Encendido

- ¿Qué ocurre al encender la computadora?
- ¿Qué instrucciones se ejecutan?
- ¿Cómo se llega a ejecutar el OS (kernel)?
- ¿Cómo se llega a darle control al usuario?



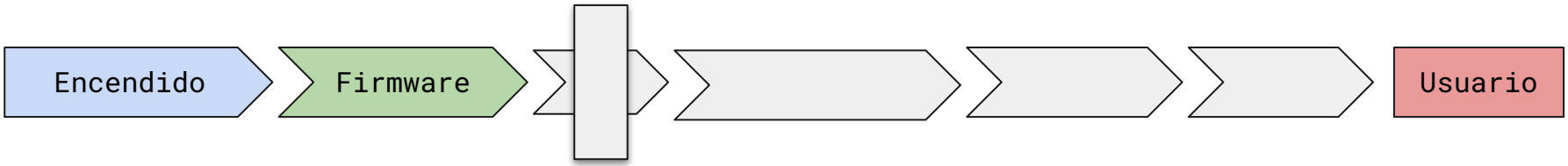
Proceso de arranque

- Tienen que ocurrir varias cosas hasta que la máquina enciende
- Comprobación de hardware (el pitido al encender)
- Carga del **kernel** en memoria
- Ejecución del proceso 1



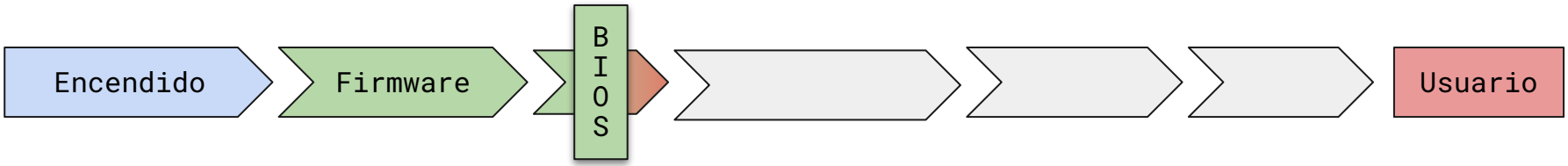
Firmware (aka “BIOS”)

- **Software embebido** en los dispositivos
  - Antes en ROM, ahora en memoria sobreescribible/mapeada en RAM
  - Lo primero que corre en cualquier dispositivo (EIP en una dirección fija, 0xffff0)
- Depende **del fabricante**, especificado en **hardware**
- Realiza distintas tareas
  - Inicializar hardware y chequear integridad de los periféricos (POST)
  - Ceder el control al OS



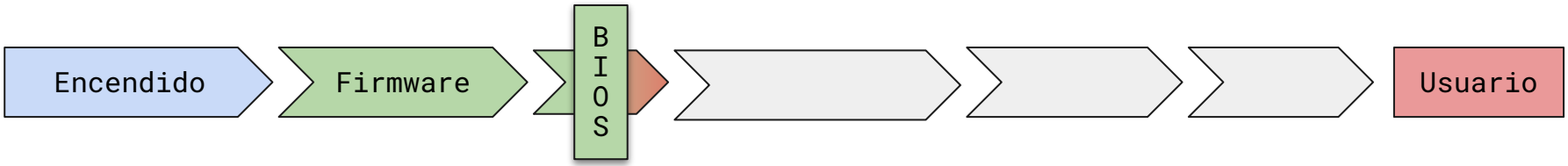
Del firmware al sistema operativo

- BIOS y UEFI son ejemplos de **interfaces** entre **firmware** y **sistema operativo**
- Estándar que define **cómo hace el firmware** para cargar el sistema operativo
    - Es el contrato entre los fabricantes de hardware y programadores de SO



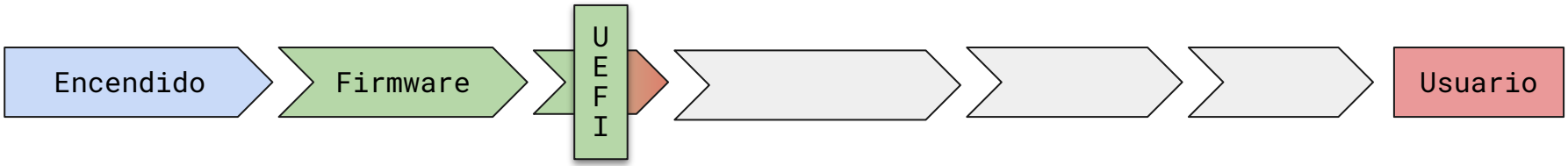
Basic Input Output System

- **Especificación de BIOS**
- La interfaz **original** (usada por primera vez por IBM)
- Es bastante "**ingenua**"
  - Itera sobre los dispositivos de almacenamiento detectados (e.g. HHD, CD-ROM, USB)
  - Cuando encuentra uno "booteable", carga en memoria **el primer sector** del dispositivo (**512 bytes**)
  - Solo carga 512 bytes, y en una región de memoria específica (0x7c00)
  - Sí, solo 512 bytes



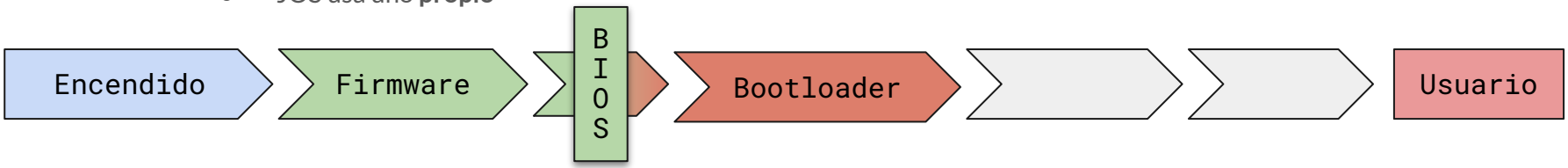
Unified Extensible Firmware Interface

- Especificación UEFI
- Un estándar moderno diseñado para sobrepasar las limitaciones de BIOS
- Es mucho más **compleja** y **flexible**
  - Capaz de reconocer particiones GPT
  - Puede reconocer y bootear directamente a cualquier OS que sea compatible
  - Muchas features más (secure boot, booteo por red PXE, fast boot)



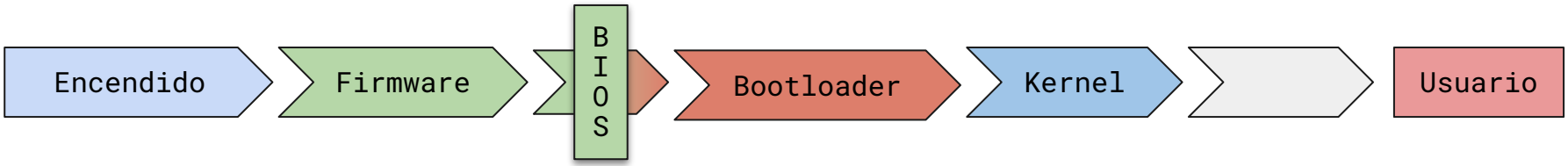
Bootloader (gestor de arranque)

- Es **software** que corre **antes que el kernel**
- La BIOS lo carga en memoria y su único propósito es **cargar y lanzar el kernel**
  - También se encarga de pasar de **modo real a modo protegido**
- Múltiples especificaciones (e.g. Multiboot)
- Múltiples implementaciones
  - GRand Unified Bootloader (a.k.a. GRUB)
  - JOS usa uno **propio**



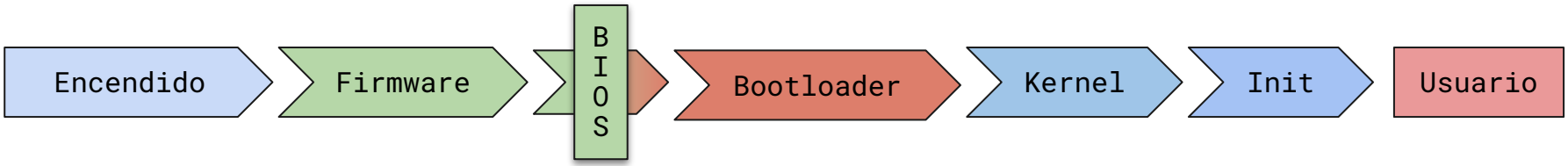
Kernel

- El **software** principal del **sistema operativo**
- Puede tener distintas fases de arranque
  - Inicialización de subsistemas (e.g. memoria, procesos, interrupciones, módulos)
- Al ser lo primero que corre tiene **control completo**
  - Dejar todo preparado para darle control al usuario
  - Ejecuta el proceso **inicial** o de arranque



Init

- Es el **primer proceso** de usuario
- Ejecuta el arranque de todos los servicios que corren en modo usuario
  - Interfaces gráficas, gestores de dispositivos, gestores de red, etc
- Cada **distribución** puede configurarla de forma distinta
  - Muchos sistemas Unix-like usan **systemd**
  - En JOS no existe el proceso **init**



TP1 - Caso de estudio: JOS

JOS - Introducción

- Sistema Operativo educativo
  - Implementado parcialmente
- El esqueleto del TP está publicado
  - Una rama para cada consigna, empezamos con el TP1
- Encuesta para conformar los grupos
  - 2 personas
  - Luego del alta en el formulario, les crearemos un repo conjunto
- Basado en curso MIT
  - La consigna original en inglés (puede ser de ayuda)
  - TP1 corresponde al Lab2 del MIT

¿Cómo usar el repositorio?

- La página de entregas explica integraciones
- Crear un *Pull Request* con todos los cambios

```
// 0) Clonar en un directorio local (e.g. mylabs)
$ git clone git@github.com:fiubatps/sisop_2021b_g10_juan_patri mylabs

// 1) Agregar el repositorio remoto con los esqueletos
$ cd mylabs
$ git remote add catedra  https://github.com/fisop/jos.git

// 2) Creación de la rama base para el tp1
$ git checkout -b base_tp1
$ git push -u origin base_tp1

// 3) Integración del esqueleto del tp1
$ git fetch --all
$ git checkout base_tp1
$ git merge catedra/tp1
$ git push origin base_tp1

// 4) Creación de la rama entrega para el tp1
$ git checkout -b entrega_tp1
$ git push -u origin entrega_tp1

// Asegurarse de  siempre commitear y pushear los cambios
// en el branch  entrega_tp1
```

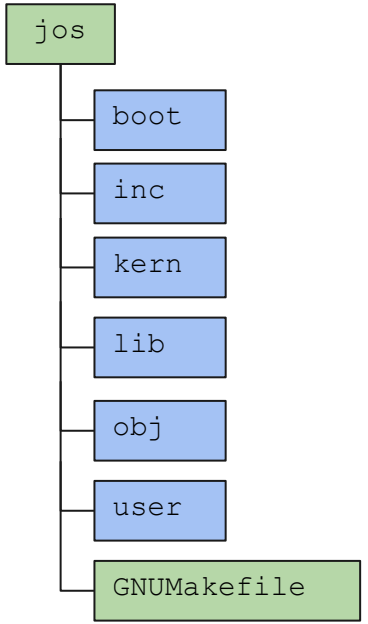
¿Cómo crear el PR?

- Se debe crear un PR
  - base_tp1 como base
  - entrega_tp1 como target
- **IMPORTANTE**
  - Sólo se deberían mostrar sus cambios

```
// Para ver el branch actual
$ git branch

// 5) Trabajar en los ejercicios
$ git add kern/pmap.c
$ git commit -m “Resuelvo TP1”
$ git push origin entrega_tp1

// 6) Ejecutar las pruebas
$ make grade

// 7) Cuando estén todos los cambios, crear el PR desde la UI
```

JOS - Navegando el repositorio

- Código en **C** con algunas secciones en **asm**
- El repo está dividido en varias carpetas
  - **boot**: el bootloader de JOS
  - **inc**: encabezados comunes (**IMPORTANTE**!)
  - **kern**: el código del kernel!
  - **lib**: código de librerías de usuario (**TP3**)
  - **user**: código de programas de usuario (**TP2**)
  - **obj**: directorio de los binarios compilados



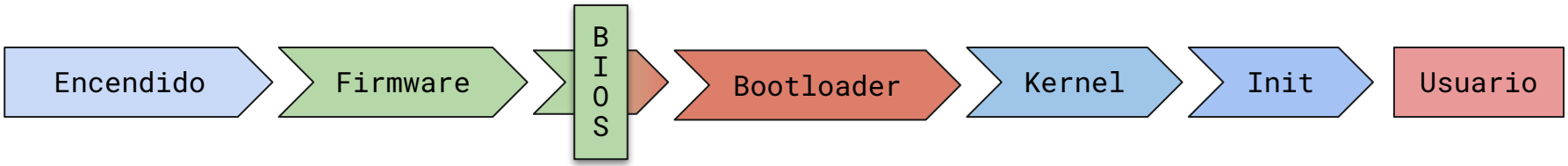
JOS - Corriendo JOS

- **make qemu**: corre JOS con modo gráfico
  - IMPORTANTE: **C-a x** para salir!
- **make qemu-nox**: corre JOS sin modo gráfico (redirige salida a la terminal)
- **make grade**: corre las pruebas automatizadas
- **make qemu-nox-gdb**: inicializa qemu pero espera una conexión de gdb para debuggear
- **make gdb**: inicia una sesión de gdb para debuggear JOS

Arranque en JOS

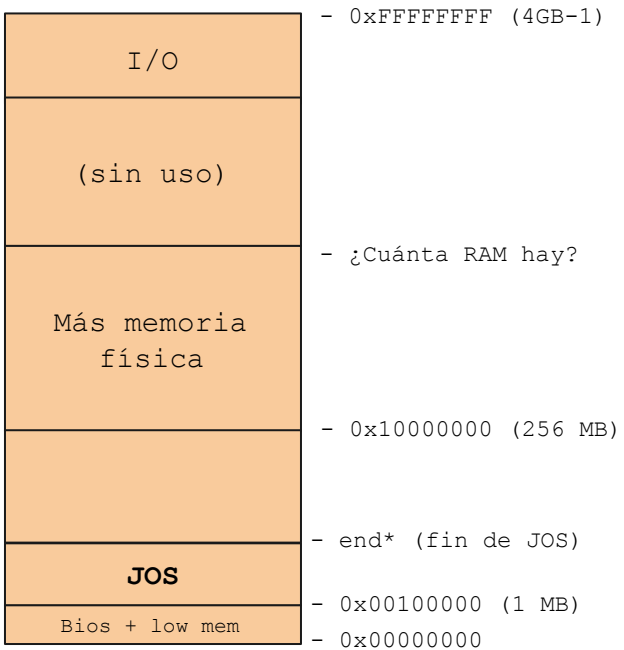
- JOS tiene **su propio bootloader**
  - Carga tipo BIOS, delegada en qemu
- En el repo se genera 2 imágenes distintas
  - El **bootloader de JOS**: imágen que se hace pasar por MBR
  - El **kernel:** genera una imágen formato **ELF**
- No hay proceso init (aún)

- ¿Dónde se encuentra el código del bootloader?
- ¿En qué dirección se enlaza? ¿Qué hace exactamente?
- ¿Donde arranca el kernel de JOS? ¿En qué direcciones está enlazado?



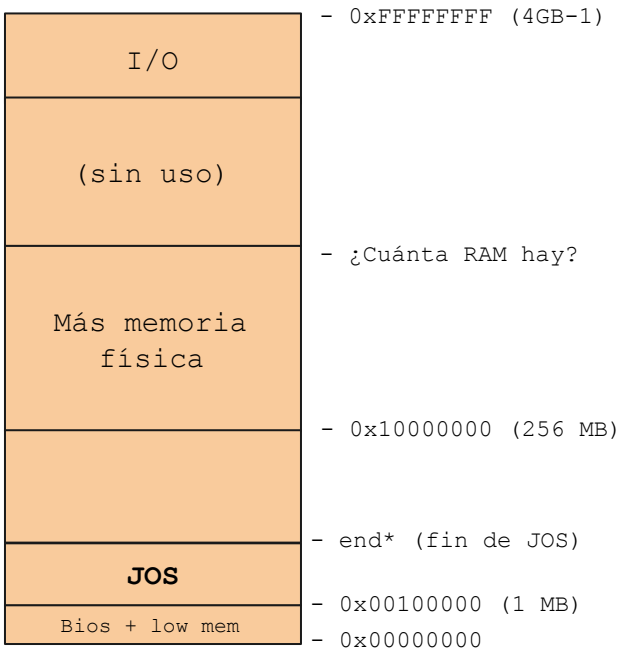
Arranque en JOS - El bootloader

- El **bootloader** carga a la imagen del **kernel** en en la **dirección física** 1MiB
  - ¿Dónde comienza su ejecución?
  - ¿De dónde obtiene la imágen?
- La imágen de JOS tiene cierto tamaño
  - El fin de esa imagen queda guardado en la variable **end**
- ¿Estamos en modo real? ¿Cómo se está direccionando la memoria?



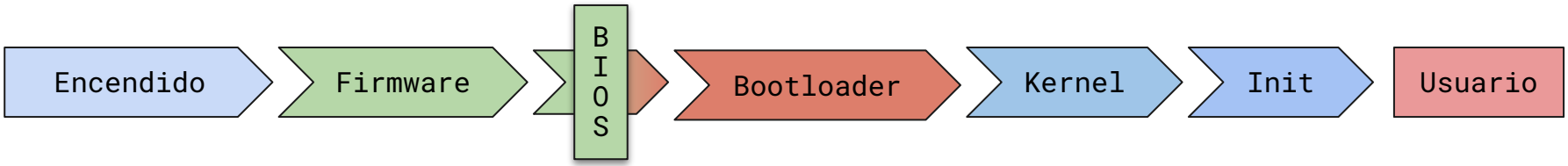
Arranque en JOS - El bootloader

- El **bootloader** carga a la imagen del **kernel** en en la **dirección física** 1MiB
  - Está enlazado para que inicie en la dirección 0x7c00
  - Lee la imágen de un disco emulado por qemu, y la carga en la dirección 0x100000 de memoria (1MiB)
- Realiza el paso de **modo real** a **modo protegido**
- Como la imagen del kernel es un ELF, usa el símbolo de **entry** para decidir dónde comienza el kernel
- Es un bootloader compatible con **multiboot**!



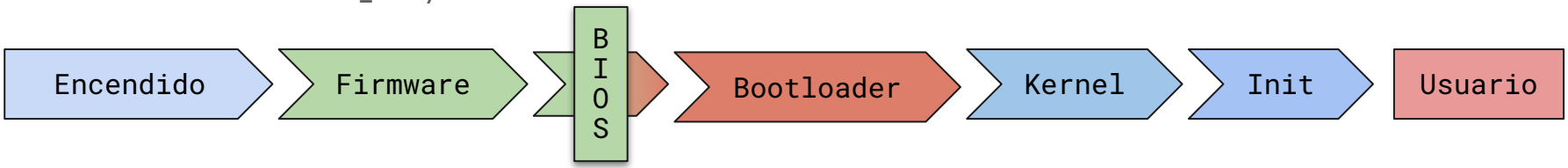
Arranque en JOS - Entrypoint del kernel

- El punto de entrada del kernel es **entry.S**
  - ¿Dónde está enlazado? ¿Por qué?
- Realiza una tarea muy específica ¿cuál?
- ¿Qué función llama luego de que termina?



Arranque en JOS - Entrypoint del kernel

- El punto de entrada del kernel es **entry.S**
  - Está enlazado en direcciones **altas**
  - Excepto por el símbolo **_start** !!
- Hace el cambio de direcciones físicas bajas a direcciones virtuales altas
  - Configuración de paginación primitiva (próxima clase)
- Llama a **i386_init** ya en direcciones **altas**



Arranque en JOS - Estamos en el kernel

- Una vez que estamos en **i386_init** ya llegamos al kernel “normal”
  - La completitud del kernel está cargada en memoria
  - Tenemos **paginación** activada (i.e. corremos en direcciones **altas**)
  - Tenemos **control total** del CPU (ring 0)
- El kernel realiza una seguidilla de llamadas de inicialización
  - Hoy nos compete **mem_init**
  - Notar la definición del símbolo **end**

Parte 1 - inicialización de la memoria

Inicializar las estructuras de memoria

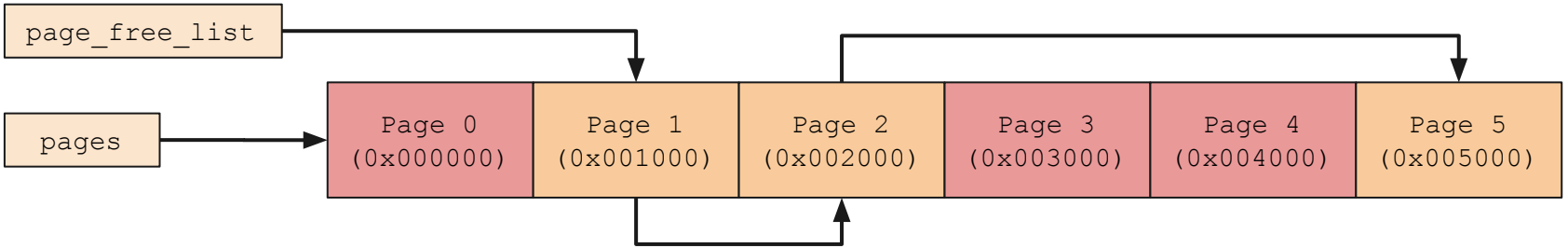
- La memoria se divide **conceptualmente** en **páginas**
  - Bloques contiguos de 4KiB
  - Alineadas a 4KiB
- Se representan con **struct PageInfo**
  - en *inc/memlayout.h*
- Se almacenan en un **array *pages***
  - Cada elemento del array es parte de una **lista enlazada**
  - Siempre son **páginas físicas**

```
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;
};

struct PageInfo *pages; // Physical page state array
```

El array pages

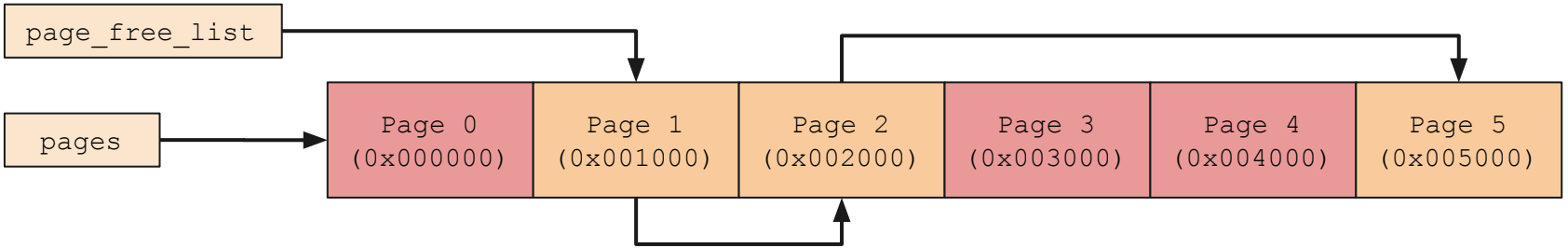
- La **lista enlazada** contiene a las páginas libres (sin uso)
- Las páginas usadas tienen su *next_free* en *NULL*
- *pp_ref* cuenta referencias en **paginación**

```
- pages[N] siempre es la página N
```



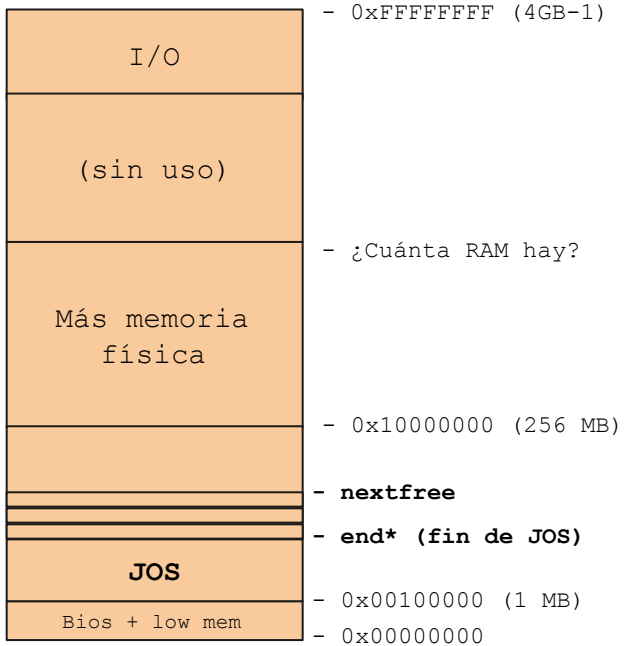
El array pages

- El **array pages** tiene que estar en memoria
- Se usa para alocar páginas de memoria física... pero
- ¿Cómo alocamos la memoria para el **array pages**?



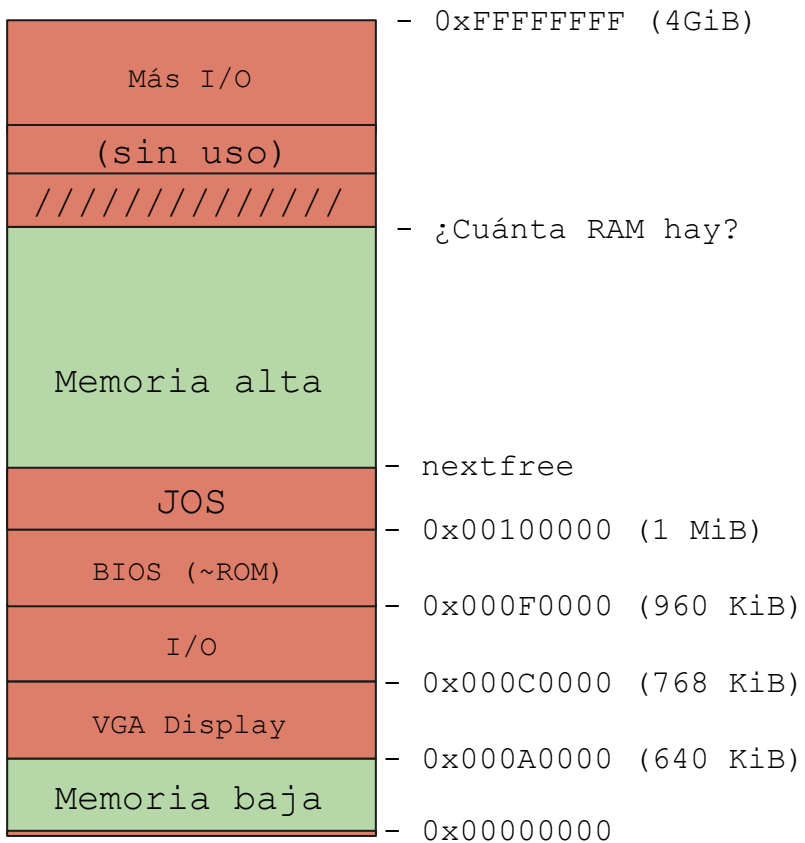
boot_alloc()

- Forma primitiva de **alocar memoria**
- Sólo para el **kernel** y se usa antes de que se haya configurado la estructura de páginas
- Reserva memoria **inmediatamente posterior** a donde termina el kernel (*end*)
- Mantiene un puntero ***nextfree*** a la siguiente dirección libre
- Devuelve una dirección **virtual** (usar *KADDR*)



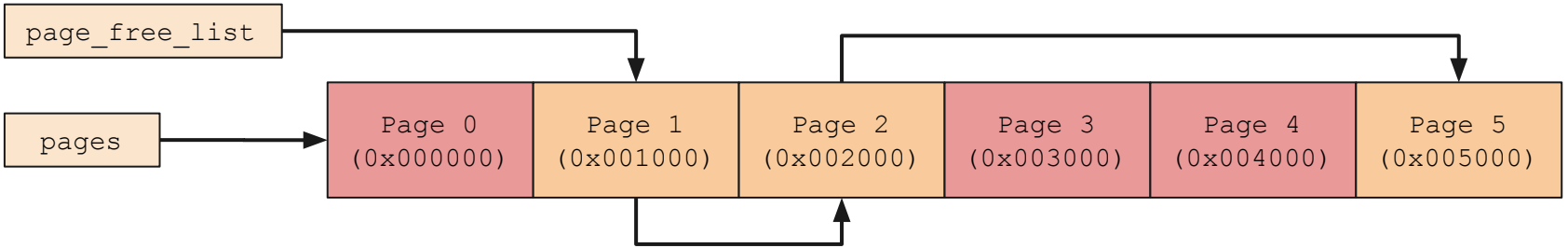
page_init()

- Inicializa el **array pages**
- ¿Cómo saber si una página está ocupada?
- Usar constantes en *inc/memlayout.h*



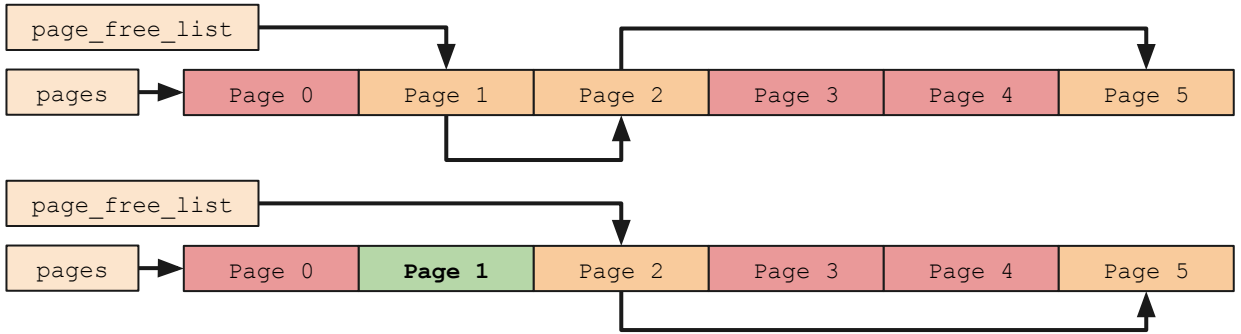
page_alloc() y page_free()

- Formas de **operar** con el **array pages**
- ¿Cómo se obtiene una página libre?
- ¿Cómo se libera una página?



page_alloc() - ejemplo

- *page_alloc* devuelve la página 1
- Avanza *page_free_list* al siguiente elemento de la lista enlazada



page_free() - ejemplo

- Se libera la página 3
- Se introduce la página liberada como cabecera de la lista