Multiprocesos en JOS

Sistemas Operativos - FIUBA

TP3 JOS - Introducción

- El esqueleto para este TP está en la rama ***tp3*** del repo de jos de la materia
- Para integrarlo deben:
  - desde su branch *entrega_tp2* crear una nueva rama **base_tp3**
  - mergear los cambios de tp3
  - pushear base_tp3 y crear entrega_tp3
- No olvidar traer sus cambios a entrega_tp3 de las correcciones del TP2

```
// Pararse en la rama entrega_tp2
$ git checkout entrega_tp2

// Crear una nueva rama base_tp3 (y pararse en ella)
$ git checkout -b base_tp3

// Integración del "esqueleto" del tp3
// No debería haber conflictos, si los hubiera,
// resolverlos
$ git fetch --all
$ git merge catedra tp3

// Probar que se corren las pruebas nuevas
$ make grade

// Pushear la rama base_tp3
$ git push -u origin base_tp3

// Creación de la rama entrega para el tp3 (y pararse
// en ella)
$ git checkout -b entrega_tp3
$ git push -u origin entrega_tp3
```

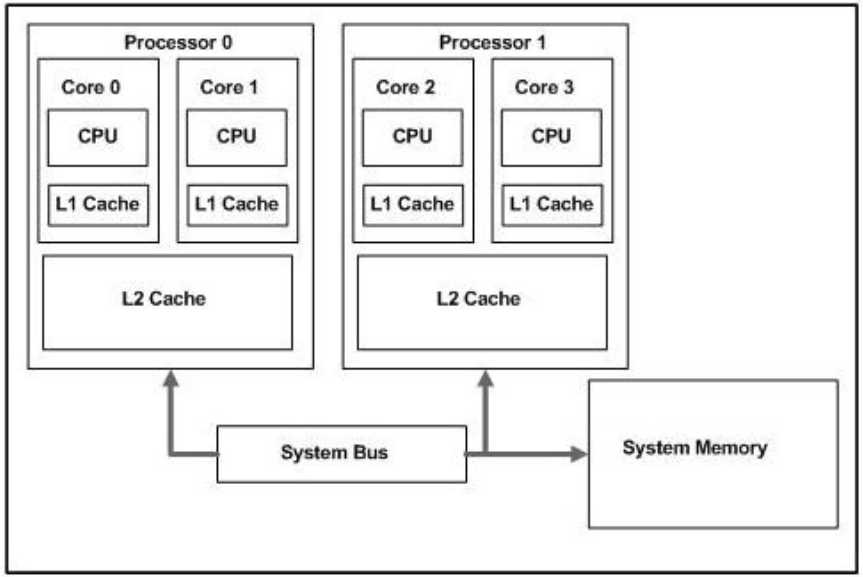
Hasta ahora...

- JOS permite lanzar un único proceso
  - ¿De dónde sale? ¿Cómo inicia?
  - ¿Qué pasa cuando termina?
- Soporte de algunas syscalls básicas
  - ¿Cuáles?
- ¿Cuántos procesadores usa JOS?

Multiprocesador (en JOS)

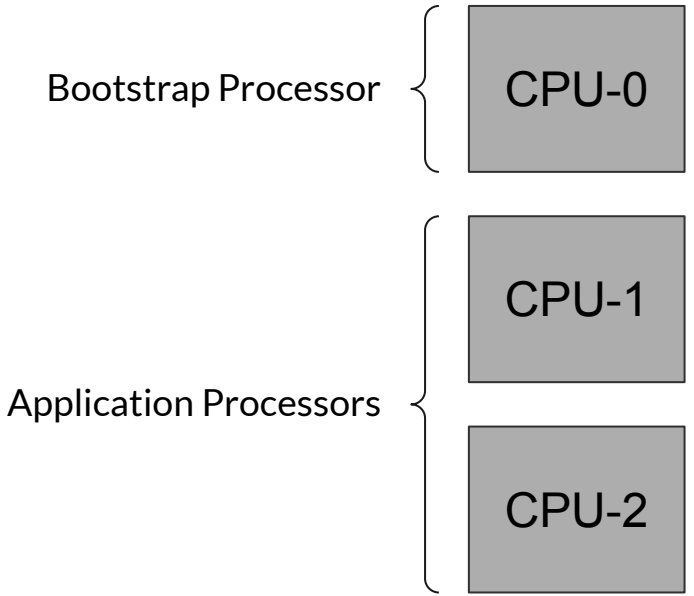
Symmetric Multi-Processing

- Todo procesador es **idéntico** en cuanto a acceso de recursos
- Cada procesador:
  - tiene su **propio set de registros**
  - tiene su propia cache (e.g. L1/L2)
  - todos arrancan en modo real
  - “controlador de interrupciones” (LAPIC)
- Todos comparten
  - acceso a memoria (**MMU, bus de datos**)
- ¿Y el stack?
- *¡Excepto durante el arranque!*



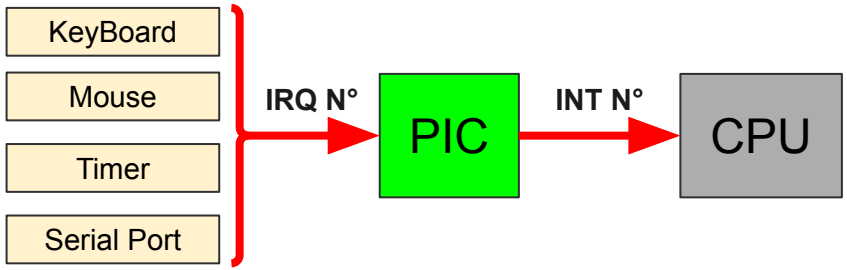
¿Cómo cambia el arranque?

- Durante el encendido **siempre** el *bootstrap processor* (BPS)
  - Determinado por el hardware (y firmware)
- El resto de los procesadores son *application processors* (APs)
- Todos los APs arrancan en estado *idle* (i.e. *halted*)
- El BPS **detecta** e **inicializa** a los APs usando LAPIC
- Los APs también arrancan en **modo real**



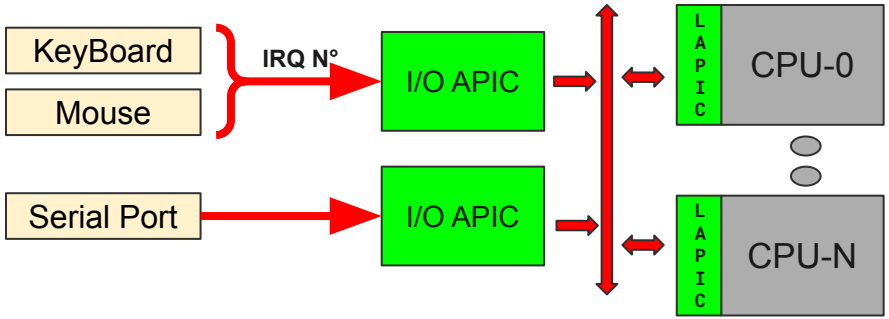
PIC (*Programmable Interrupt Controller*)

- En los primeros x86, controla las interrupciones
- Mapea señales del bus de interrupciones (*Interrupt Requests*, o IRQ) a interrupts del procesador
- Generan interrupciones del CPU según IRQ base + offset

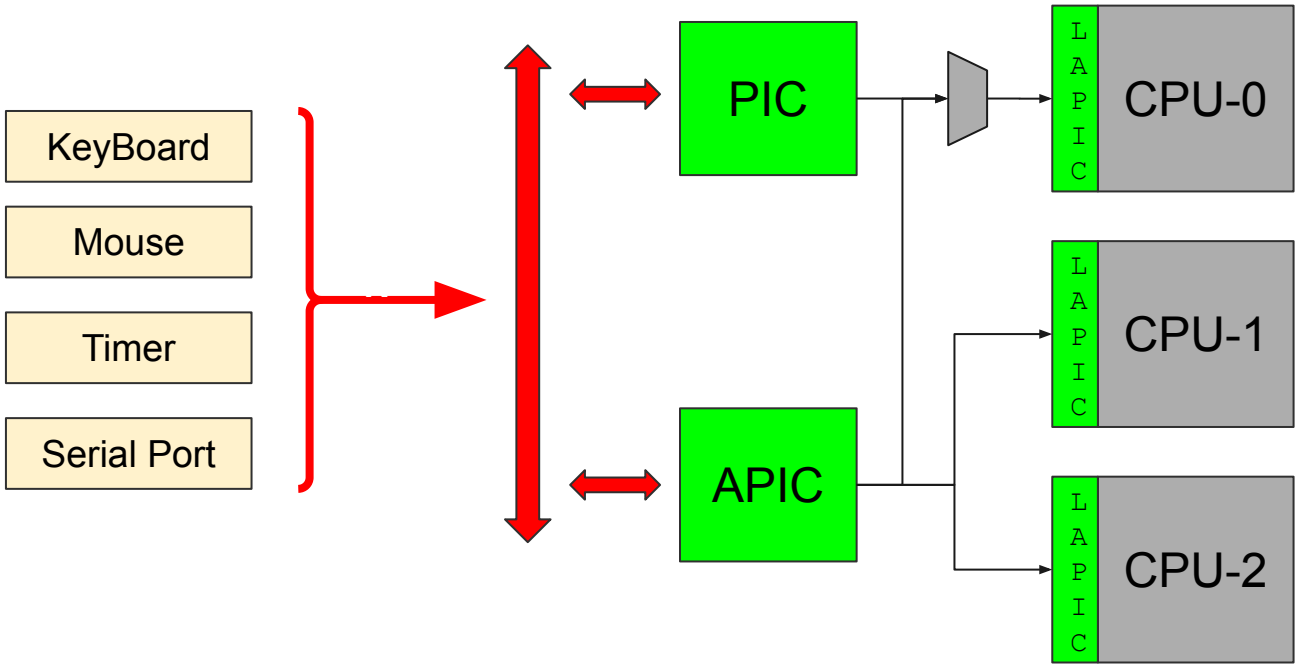


APIC (*Advanced PIC*)

- El sucesor de PIC
- Dos tipos:
  - I/O APIC: uno por cada bus de periféricos
  - Local APIC: manejo de interrupciones por CPU
- Por compatibilidad hacia atrás, conviven con el PIC del CPU-0
- Permiten comunicación **entre CPUs**

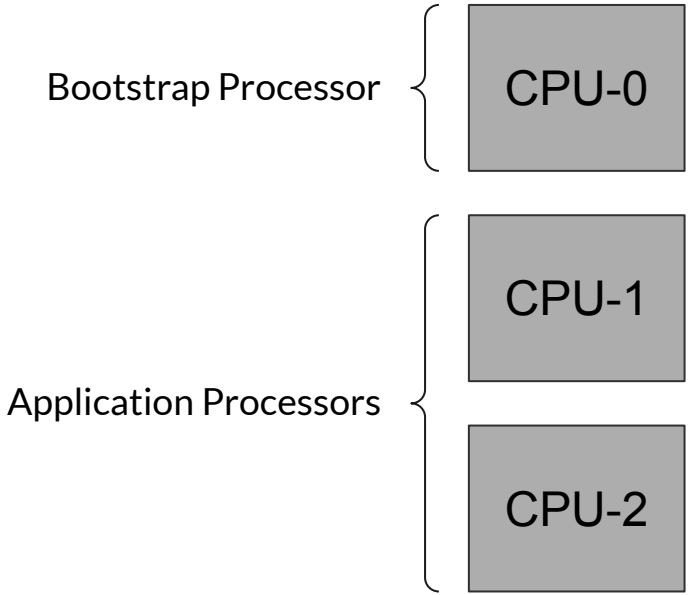


PIC y APIC combinados



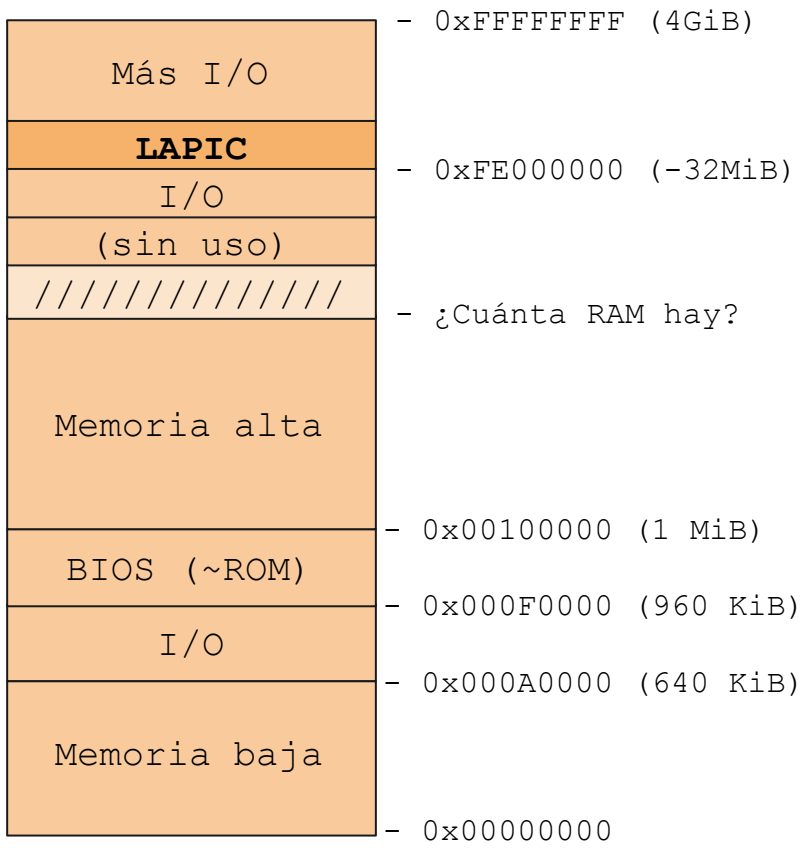
¿Cómo cambia el arranque?

- El BPS **arranca** de forma normal, pasa a modo protegido y activa paginación
- El BPS **configura** tanto PIC como LAPIC
- El BPS **utiliza LAPIC** para enviar una **señal de encendido** a cada APs
- Cada APs arranca y pasa a modo protegido a su tiempo
    - ¿Qué código ejecuta?
- El BPS ejecuta procesos, el resto se queda *idle* en el monitor



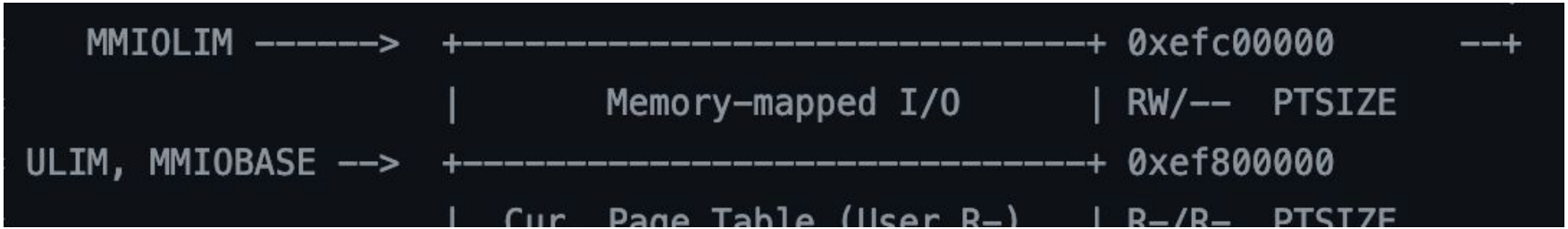
Configurando LAPIC

- LAPIC está mapeado a direcciones de memoria física
- Son direcciones **físicas** altas
  - ¿Cómo podemos acceder?
- Se configura en **lapic_init**()
- Algo similar ocurre con el PIC del BPS en **pic_init**()



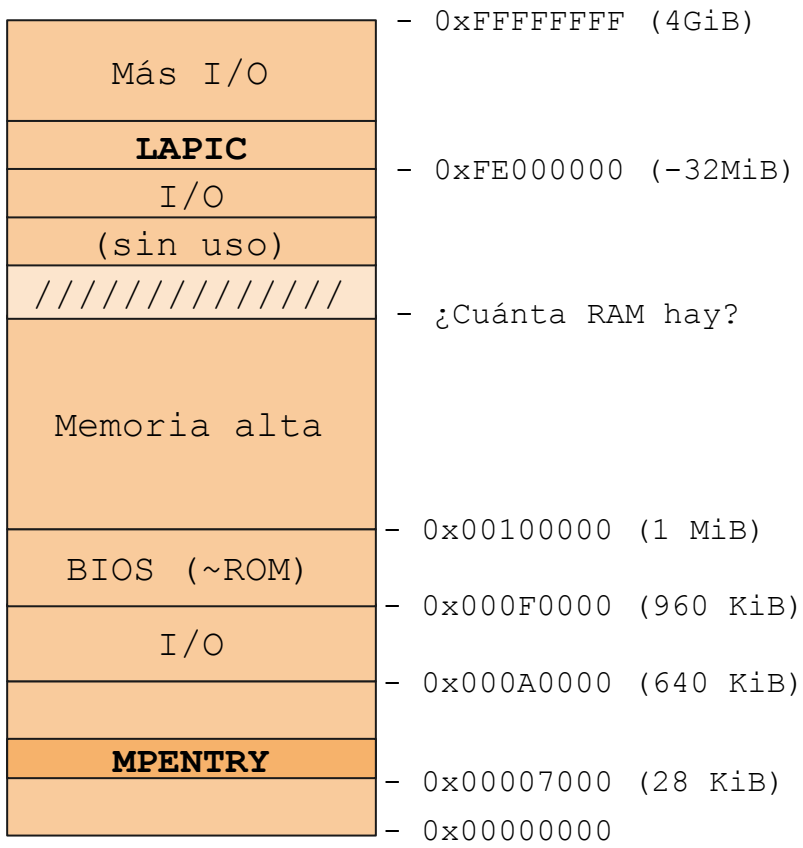
Configurando LAPIC

- ¡No se puede acceder directamente!
- JOS reserva *MMIOLIM* para mapear la región de LAPIC en su pgdir y así poder accederlo
- ¿Podemos mapearlo normalmente?
  - No! Hay que utilizar flags especiales para indicar que es MMIO
  - *Cache Disable* (PTE_CD) y *Write Through* (PTE_WT)



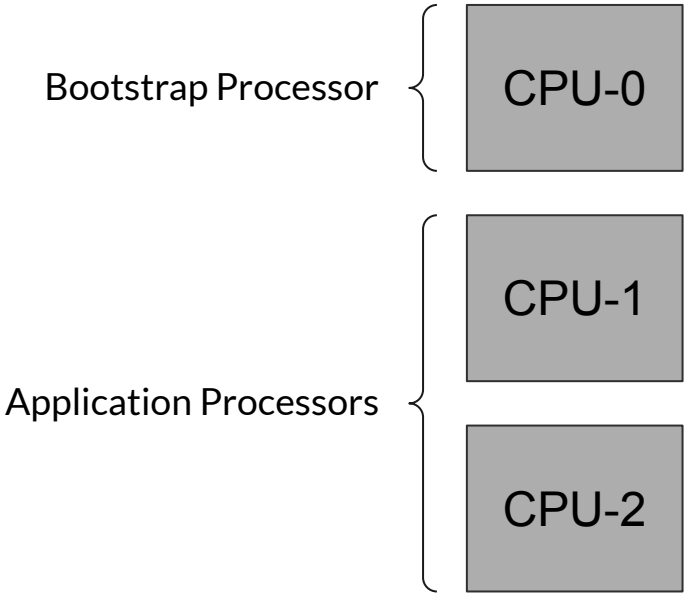
¿Dónde arrancan los APS?

- Tienen que arrancar en **memoria física baja**
  - No tienen paginación activada
- **No** pueden arrancar en el mismo lado que el BPS
- Tienen su propio código en **mpentry.S**
- Hay una **página física** reservada para hacer de punto de entrada.
- ¿Se puede usar esa página? ¿Por qué?
- ¿Está el código ahí originalmente?
- ¿Y el stack? Hay que inicializar uno para cada CPU.



¿Cómo cambia el arranque? - Resumen

- BPS
  - arranque normal (entry.S)
  - *mp_init*() -> detecta número de CPUs, inicializa arreglo global *cpus*
  - *lapic_init*() -> configuración del LAPIC
  - *pic_init*() -> configuración de I/O APIC/PIC
  - *boot_aps*() -> prepara y “despierta” los APS
  - *env_run*() -> primer proceso
- APS
  - arranque multiprocesador (mpentry.S)
  - queda *idle*



Problemas de SMP

- ¡Más de un procesador corriendo código a la vez!
- Posibles accesos concurrentes a las estructuras del kernel
- La solución de JOS:
  - Un lock (mutex) para entrar al kernel
  - ¿En qué puntos se debería tomar/liberar?
- ¿Cómo implementamos un lock en el kernel?

Tarea 0 - Configuración SMP

Tarea 0 - Preparación SMP

- **mem_init_mp()**
  - Realiza los mapeos necesarios para el stack de cada procesador (ver inc/memlayout.h)
- **mpentry_addr()**
  - Actualizar page_init para reflejar el uso de mpentry
- **mmio_map_region()**
  - Similar a boot_map_region, pero específica para mappear regiones de MMIO
- **mpentry_cr4**()
  - Activar *large pages* en los APS

Tarea 3 - Arrancando SMP

- **multicore_init()** [*ya implementado*]
  - Repasar el arranque de un AP
- **trap_init_percpu()**
  - Actualizar trap_init para reflejar los stacks por CPU (¿Por qué es necesario? ¿Qué stack se usa al volver al kernel?)
- **kernel_lock()**
  - Usar el Kernel Big Lock donde corresponda

Scheduling en JOS

Hasta ahora...

- JOS reconoce y despierta a todos los CPUs disponibles
- JOS permite correr únicamente un sólo proceso
  - ¿Qué ocurre cuando termina?
  - ¿Qué pasa si ese proceso decide no terminar nunca y acapara el CPU?
  - ¿Cómo garantizamos que todos los procesos puedan correr?
- ¿Cómo hacemos uso de estos nuevos CPUs?
- ¿Cómo ***creamos*** nuevos procesos?
  - ¿Permitimos que el usuario los cree?

Round Robin Scheduler

- En JOS implementaremos un scheduler de tipo *round robin*.
  - Siempre schedulea “al siguiente” enviroment capaz de correr
  - Si no hay más procesos, pone el CPU a dormir
- Tendrá dos fases
  - Non-preemptive (sin desalojo)
  - Preemptive (con desalojo)
- Se implementará en la función **sched_yield**()

sys_yield()

- En la versión sin desalojo
- Permite a un proceso ceder voluntariamente el CPU
- Scheduling cooperativo
- ¿Cómo se implementaría? ¿Qué debe hacer el kernel si alguien llama a sys_yield?

Round Robin Scheduler - esperando

- ¿Qué diferencia hay entre las siguientes formas de “esperar” a una condición?
- ¿Alguna es mejor? ¿Hay otras?
- ¿Cuál preferirían para un scheduler?

```
while (!condition) {
    ;
}

while (!condition) {
    sleep(1);
}
```

Round Robin Scheduler - esperando

- La instrucción **hlt** (halt) pone a dormir al CPU hasta que llegue una interrupción.
- Durante ese tiempo, el CPU está apagado y consume energía mínima.
- Está implementada en **sched_halt**()
  - ¿Hace algo más?
  - ¿Por qué?

```
while (!condition) {
    hlt(1);
}
```

Scheduler con desalojo

- Se necesita la utilización de un **timer** (temporizador)
- Generación de interrupciones periódicas que le devuelven el control al kernel
- En JOS, se configura un timer en **pic_init()**
- Es necesario soportar las interrupciones
- ¿Cómo nos aseguramos que estemos aceptando interrupciones?

Tarea 1 - Scheduler

- **sched_yield()**
  - Implementación del scheduler *round robin*
- **sys_yield()**
  - Implementación de la *syscall yield*
- **timer_irq() y timer_preempt()**
  - Soporte para scheduler con desalojo a través de la interrupción del timer
  - Se tienen que habilitar las interrupciones en procesos de usuario!